UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Hybrides Ray Tracing mit RTX-Technologie in Vulkan

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Maximilian Mader

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

Koblenz, im Juni 2019

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 der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....
(Ort, Datum) (Unterschrift)

Zusammenfassung

Im Rahmen dieser Masterarbeit wird das Prinzip des hybriden Ray Tracing, einer Kombination einer Rasterisierungs-Pipeline mit Ray Tracing-Verfahren für einzelne Effekte, vorgestellt und eine Anwendung implementiert, welche innerhalb einer hybriden Ray Tracing-Pipeline Schatten, Umgebungsverdeckung und Reflexionen berechnet und diese Effekte mit der direkten Beleuchtung kombiniert. Das hybride Ray Tracing basiert auf der Idee, die Performance und Flexibilität von Rasterisierungs-Pipelines mit Ray Tracing zu kombinieren, um die Limitation der Rasterisierung, nicht auf die gesamte Umgebungsgeometrie an jedem Punkt zugreifen zu können, aufzuheben. Im Rahmen der Implementation wird in die verwendete RTX-API sowie die Grafikschnittstelle Vulkan eingeführt und diese anhand der Implementation erklärt. Auf Grundlage der Ergebnisse und der Erkenntnisse bei der Nutzung der API wird diese, ihre Einsatzzwecke und Ausgereiftheit belangend, eingeschätzt.

Abstract

In this master's thesis the principle of hybrid ray tracing, consisting of a rasterization pipeline which includes ray tracing techniques for certain effects, is explained and the implementation of an application which uses a hybrid approach in which ray tracing is used to calculate shadows, ambient occlusion, and reflections and combines those with direct lighting is documented and explained. Hybrid ray tracing is based on the idea of combining the performance and flexibility of rasterization-based approaches with ray tracing to overcome the limitation of not being able to access the complete surrounding geometry at any point in the scene. While describing the implementation of said application, the RTX API which is being used for ray tracing is explained as well Vulkan, the graphics API used. Based on the results and the insights gained while using the RTX API, it is assessed in regards of its usage scenarios and technical sophistication.

Inhaltsverzeichnis

1 Einleitung

Hybrides Ray Tracing ist ein Ansatz, welcher klassische Rasterisierungs-Techniken in einer Rendering-Pipeline mit Ray Tracing-Techniken kombiniert. So wird aus einer Standard-Pipeline für Echtzeitrendering eine Kombination, bei der sämtliche Informationen über die gesamte Szenengeometrie an einer beliebigen Stelle im Raum bereitstehen. Dies erlaubt eine visuelle Qualität jenseits der durch die fehlenden Umgebungsinformationen begrenzten Rasterisierungs-Ansätze und ermöglicht das Eingliedern der intuitiveren Ray Tracing-Denkweise in Echtzeitapplikationen, welche von Sichtstrahlen statt vom Rastern von Geometrie ausgeht.

Ziel dieser Arbeit ist es, die Möglichkeiten der neuartigen, für Echtzeit-Ray Tracing gedachten Programmierschnittstellen durch die Implementierung von zuvor durch Rasterisierungs-Techniken gelösten Verfahren zum Rendering von Schatten, Reflexionen und Umgebungsverdeckung in Kombination mit einer Rasterisierungs-Pipeline zu erproben.

1.1 Motivation

Rasterisierungs-basierte Ansätze in der Echtzeit-Computergrafik haben trotz kontinuierlicher Weiterentwicklung auf eine inhaltliche Art und Weise einen Fixpunkt erreicht. Während die vorhandenen Methoden weiter mit der Hardware beispielsweise in ihrer Auflösung skalieren und auf immer mehr Geräten verfügbar werden, ist die grundlegende Limitierung von Rasterisierungs-basierten Ansätzen, die Nicht-Verfügbarkeit des Zugriffs auf die gesamte Umgebungs-Geometrie, doch immer vorhanden. So bestehen Rasterisierungs-Techniken hauptsächlich aus Ansätzen, diese Limitation zu umgehen, indem eine Art von bereits generierten Ressourcen genutzt wird.

Durch die Verfügbarkeit von nativer Hardware-Unterstützung und von vorab eingebauten Ray Tracing-Funktionen in Grafikprogrammierschnittstellen rücken nun Techniken in Reichweite, welche mit vollem Zugriff auf die Szenengeometrie arbeiten können. Ray Tracing kann nun, im Rahmen der verfügbaren Performance, als zusätzliches Werkzeug in der Echtzeit-Computergrafik eingesetzt werden.

Das durch die Leistung der Hardware vorgegebene Budget an Strahlen bringt neue Herausforderungen in die Echtzeit-Computergrafik. So wird der Fokus vom Umgehen des Problems der fehlenden Information der Umgebungsgeometrie gerichtet auf die Probleme von Ray Tracing mit einem eingeschränkten Strahlen-Budget: Das Behandeln von Rauschen, das sinnvolle Verteilen der begrenzten Anzahl von verschickbaren Strahlen pro Bild und vor allem das Bilden neuer Abläufe, Pipelines für Anwendungen, welche Ray Tracing als Zusatz zur Rasterisierung einsetzen.

2 Grundlagen

In den folgenden Abschnitten werden die Grundlagen dieser Arbeit, sowohl Hardwareseitig, als auch mit Bezug auf die verwendeten Softwareschnittstellen und die theoretischen Grundlagen, vorgestellt.

2.1 NVIDIAs Turing-Grafikprozessorgeneration

Die sogenannte *Turing*-Grafikkartengeneration von *NVIDIA* beinhaltet erstmals direkt in eine standardmäßige Verbraucher-Grafikkarte (im Gegensatz zu Grafikkarten für professionelle Anwendungen) eingebaute Hardware, welche zur Beschleunigung von Ray Tracing-Anwendungen dient.

2.1.1 Aufbau der Grafik-Chips

Der eigentliche Grafikchip einer Grafikkarte, auch *GPU* (*Graphics Processing Unit*) genannt, auf dem die Prozessorlogik ausgeführt wird, ist aus sogenannten *Streaming Multiprocessors* (*SMs*) aufgebaut. Diese SMs beinhalten die eigentlichen „Kerne“, welche die Instruktionen, die beispielsweise in Shader-Programmen enthalten sind, ausführen. Wie in Abbildung 1 zu sehen ist, besteht jeder SM physikalisch hauptsächlich aus vier Blöcken mit jeweils 16 Integer-Kernen und 16 Floating-Point-Kernen. Die physikalische Implementation von Integer-Kernen auf *NVIDIA*-Grafikkarten ist neu und verhindert das Stillstehen der Floating-Point-Berechnungen wenn eine Integer-Operation ausgeführt wird, wie von NVIDIA [NVI18] beschrieben. Vor dem Einführen dieser Integer-Kerne wurden Integer-Operationen von dem Floating-Point-Kernen übernommen.

Die in Abbildung 1 sichtbaren *Tensor Cores*, welche Matrix-Multiplikationen und -Additionen mit gemischter, niedriger Präzision erlauben, sind ebenfalls neu für Verbraucher-Grafikkarten, für professionelle Anwender schon in einer anderen Form in der vorausgegangen *Volta*-Generation verfügbar.

Die für diese Arbeit relevanten *Ray Tracing Cores*, in Abbildung 1 als *RT Core* bezeichnet, sind ebenfalls neu und werden von *NVIDIA* als die signifikanteste Neuerung der Turing-Generation angesehen[1]. Diese Ray Tracing Cores ermöglichen eine hardware-beschleunigte Traversierung der von *NVIDIA* implementierten Ray Tracing-Beschleunigungsdatenstruktur, einer *bounding volume hierarchy*, siehe auch Abschnitt 2.1.3, und die hardware-beschleunigte Berechnung des Schnittpunkts von einem Strahl und einem Dreieck, wie es bei Ray Tracing-Anwendungen benötigt wird[1]. Die jeweiligen Ray Tracing Cores auf dem SM werden von den Threads benutzt, welche auf den „normalen“ Kernen desselben SMs ausgeführt werden.

[1] `https://www.nvidia.com/de-de/geforce/20-series/`, letzter Abruf: 5. Juni 2019

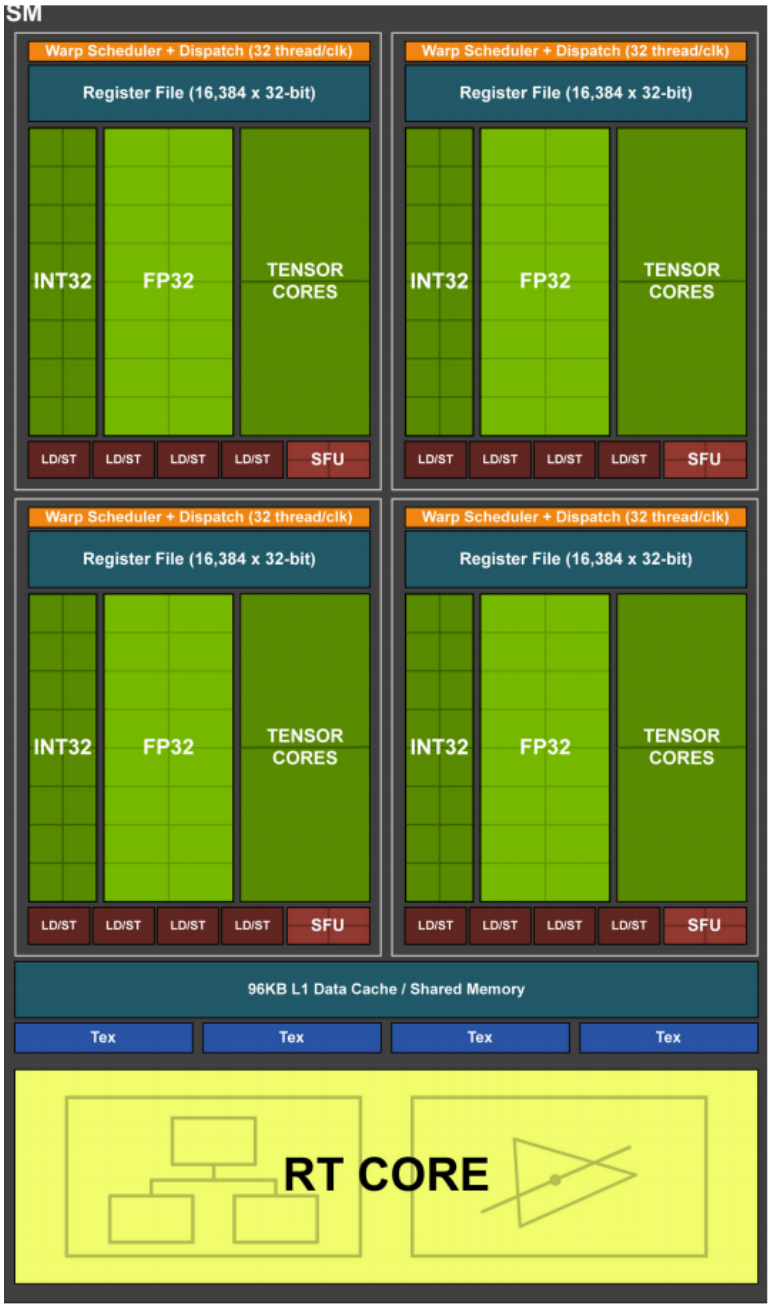


Abbildung 1: Aufbau eines Turing-SMs. Quelle: NVIDIA [NVI18]

An dieser Stelle ist anzumerken, dass die Verfügbarkeit der RTX-API im Laufe der Arbeit auf ausgewählte *NVIDIA*-Grafikkarten ausgeweitet wurde, auf denen keine Ray Tracing Cores verbaut sind[2]. Dabei wird die Aufgabe der Ray Tracing Cores von den standardmäßig vorhandenen Kernen übernommen, wobei die Performance dementsprechend schlechter ausfällt.

2.1.2 Die RTX-API

Zur Nutzung der zuvor vorgestellten Ray Tracing Cores und deren Möglichkeiten für die Beschleunigung für Ray Tracing-Anwendungen, wird von *NVIDIA* das sogenannte *RTX-API* (*RTX*-Programmierschnittstelle) bereitgestellt. Diese ist verfügbar in der Grafik-Programmierschnittstelle *DirectX 12* von Microsoft und wird dort auch als *DXR*[3] (*DirectX Ray Tracing*) bezeichnet, in *NVIDIA*s *OptiX*[4], einer Programmierschnittstelle für Ray Tracing-Anwendungen, auf deren von Parker et al. [Par+10] vorgestelltem Design die RTX-Schnittstelle basiert, und in der Grafik-Programmierschnittstelle *Vulkan*[5] der Khronos Group als *NVIDIA*-exklusive Erweiterung mit dem Namen *NV_ray_tracing*[6]. Mehr zu Vulkan, Vulkan-Erweiterungen und der Umsetzung der RTX-API in Vulkan ist in Abschnitt 2.2 beschrieben.

Die Funktionsweise der RTX-Programmierschnittstelle, hier beschrieben mit Bezug auf Vulkan (bei gleichwertigem Funktionsumfang und vergleichbarer Funktionsweise in den anderen zuvor erwähnten Programmierschnittstellen) ist grundlegend in Abbildung 2 abgebildet. In dem Diagramm sind grün eingefärbte Kästen programmierbare Shader und graue Kästen eine fixe, vom Grafiktreiber ausgeführte Funktion, vergleichbar mit den nichtprogrammierbaren Stufen in einer klassischen Rasterisierungs-Grafikpipeline. Der Ablauf ist wie folgt: Strahlen werden wie in einem *Ray Generation Shader* beschrieben generiert und das Verschicken der Strahlen mit einem *Trace-Command* ausgelöst (oberer Teil in Abbildung 2).

Danach wird vom Treiber mithilfe der in Abschnitt 2.1.1 beschriebenen Ray Tracing Cores die Beschleunigungsdatenstruktur traversiert, um zu ermitteln, welche Geometrie für einen Schnitttest in Frage kommt. Dieser Schnitttest wird bei dem Standardfall „Strahl-Dreieck“ ebenfalls von den Ray Tracing Cores berechnet, kann aber auch vom Nutzer der Programmierschnittstelle mit einem *Intersection Shader* berechnet werden. Gibt es einen von den Ray Tracing Cores oder von dem Ausführen des Intersection Shaders berechneten Schnittpunkt, wird, falls bereitgestellt, ein *Any Hit Shader* ausgeführt.

[2] `https://www.nvidia.com/en-us/geforce/news/geforce-gtx-dxr-ray-tracing-available-now/` 11. April 2019, letzter Abruf: 5. Juni 2019

[3] `https://blogs.msdn.microsoft.com/directx/2018/03/19/announcing-microsoft-directx-raytracing/`, letzter Abruf: 5. Juni 2019

[4] `https://developer.nvidia.com/optix`, letzter Abruf: 5. Juni 2019

[5] `https://www.khronos.org/vulkan/`, letzter Abruf: 5. Juni 2019

[6] `https://github.com/KhronosGroup/Vulkan-Docs/blob/master/appendices/VK_NV_ray_tracing.txt`, letzter Abruf: 5. Juni 2019

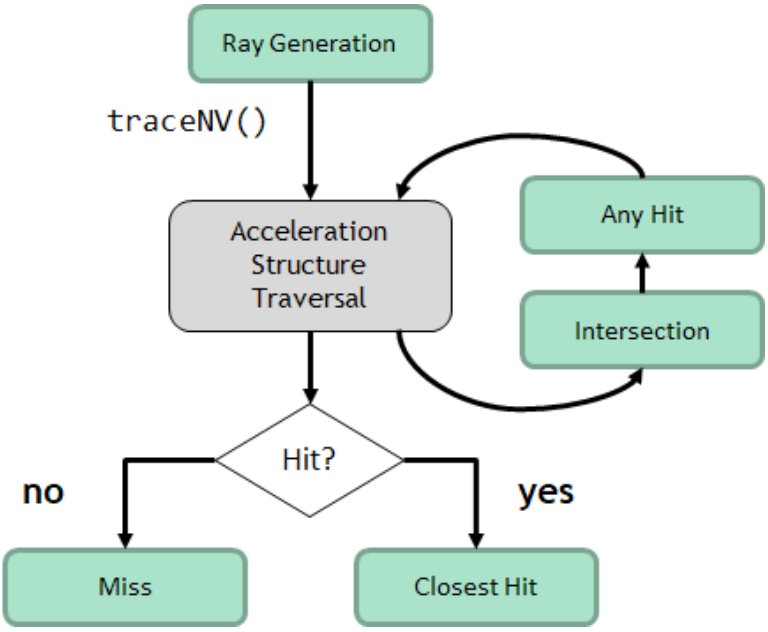


Abbildung 2: Beziehungen der Ray Tracing-Shader zueinander. Quelle: `https://devblogs.nvidia.com/vulkan-raytracing/`

In diesem können nun beispielsweise Beleuchtungsinformationen berechnet werden, es kann aber auch das weitere Nachverfolgen des aktuellen Strahls abgebrochen werden. Zusätzlich zum *Any Hit Shader* oder auch stattdessen kann ein *Closest Hit Shader* ausgeführt werden. Dieser wird nur bei dem vordersten Schnittpunkt des Strahls mit der Geometrie ausgeführt und ist für Standard-Ray Tracing-Applikationen somit besonders wichtig.

Wird kein Schnittpunkt festgestellt, wird ein *Miss Shader* ausgeführt. Closest Hit Shader und Miss Shader sind in der Lage, weitere Strahlen zu verschicken[7] und können über die sogenannte *Ray Payload* miteinander kommunizieren, indem sie diese mit vom Nutzer bestimmten Datenstrukturen füllen und dann Lese- und Schreibzugriffe ausführen können. Ebenfalls neu in der RTX-API ist es, dass mithilfe der sogenannten *Shader Binding Table* mehrere Shader desselben Typs in derselben Grafikpipeline existieren können. So können abhängig davon, welche Art von Geometrie getroffen wird (festgelegt durch die Beschleunigungsdatenstruktur), verschiedene Shader desselben Typs aufgerufen werden. Genauere Informationen zu diesen Funktionen der RTX-API und deren sinnvolle Nutzung ist im Implementationsteil der Arbeit, Abschnitt 5, zu finden.

[7] `https://github.com/KhronosGroup/GLSL/blob/master/extensions/nv/GLSL_NV_ray_tracing.txt`, letzter Abruf: 5. Juni 2019

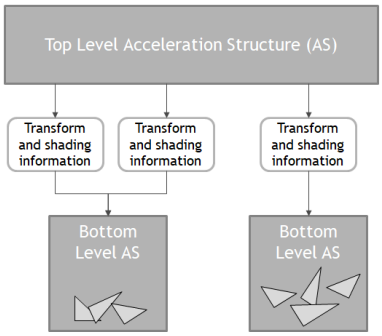


Abbildung 3: Der Aufbau der Beschleunigungsdatenstruktur der RTX-API. Bildquelle: `https://devblogs.nvidia.com/vulkan-raytracing/`, letzer Abruf am 5. Juni 2019

2.1.3 Die Beschleunigungsdatenstruktur

Die Beschleunigungsdatenstruktur, welche von der RTX-API benutzt wird, ist eine sogenannte *two level bounding volume hierarchy*, also eine zweistufige Hierarchie von Volumen, welche die Geometrie einschließen. Demnach gibt es eine lokale Beschleunigungsdatenstruktur (*bottom level acceleration structure*) pro Objekt in der Szene, welche innerhalb einer übergeordneten Beschleunigungsdatenstruktur (*top level acceleration structure*) platziert und mehrfach instanziiert werden kann, wie in Abbildung 3 gezeigt ist.

Sämtliche einzelne Beschleunigungsdatenstrukturen, sowohl die untergeordneten Strukturen auf Dreiecksbasis als auch die übergeordnete Struktur, welche die untergeordneten beinhaltet werden von der API im Grafikspeicher aufgebaut. Anstatt einer vom Treiber aufgebauten Beschleunigungsdatenstruktur für ein Objekt kann auch nur eine *axis aligned bounding box*, eine an den Achsen ausgerichtete quaderförmige Hülle, an die API übergeben werden. Die Schnittpunkte innerhalb dieser werden dann mit manuell geschrieben Intersection Shadern berechnet.

2.2 Vulkan

Vulkan, entwickelt von der Khronos Group, ist eine Grafik-Programmierschnittstelle mit einem Fokus auf explizite Kontrolle über die Hardware auf einer möglichst hardware-nahen Ebene (*low-level*)[8]. Dabei soll bewusst eine grobe Abstraktion durch den Treiber, wie es beispielsweise bei OpenGL[9] der Fall ist, vermieden werden, um dem Nutzer der Schnittstelle möglichst viel Kontrolle zu geben.

[8] `https://www.khronos.org/assets/uploads/developers/library/overview/vulkan-overview.pdf`, letzter Abruf: 5. Juni 2019

[9] `https://www.khronos.org/registry/OpenGL/index_gl.php`, letzter Abruf: 5. Juni 2019

Das Resourcenmanagement für GPU-Speicher und die Synchronisation des Speicherzugriffs muss durch den *low-level*-Ansatz von Vulkan vom Benutzer übernommen werden. Shader können zwar wie bei OpenGL in der Shadersprache GLSL geschrieben werden, müssen aber im Vorhinein in die Zwischenrepräsentation *SPIR-V*[10] kompiliert werden. Vulkan nutzt ein verzögertes Ausführungsmodell, bei dem die für die Grafikhardware bestimmten Befehle nicht sofort ausgeführt werden, sondern in *Command Buffer* gespeichert werden und anschließend gesammelt auf einer *Queue* ausgeführt werden. Das hat den Vorteil, dass mehrere *Queues* parallel angesprochen werden, was bei moderner Grafikhardware ermöglicht, Grafikbefehle und Allzweckberechnungen oder Transferoperationen parallel auszuführen. Außerdem können *Command Buffer* in mehreren CPU-Threads gleichzeitig gefüllt werden, was das Aufteilen der Berechnungslast auf CPU-Seite einer Grafik-Anwendung ermöglicht.

2.2.1 Erweiterungen von Vulkan

Für Vulkan gibt es wie für OpenGL *Extensions* (Erweiterungen), welche von der Khronos Group selbst oder von Treiber- und Hardwareherstellern entwickelt oder bereitgestellt werden[11]. Diese stellen zusätzliche, teilweise hardwareabhängige Funktionalität bereit. Anders als beispielsweise bei OpenGL ist die Anzeige eines Bilds auf den Bildschirm bei Vulkan abhängig von einer Extension (`VK_KHR_swapchain`), da Vulkan auch auf Hardware ausgelegt ist, welche keine Anzeigeausgaben unterstützt, sondern lediglich für die Beschleunigung von Allzweckberechnungen gedacht sind.

2.3 Ray Tracing

In den folgenden Abschnitten werden die für diese Arbeit benötigten Grundlagen von Ray Tracing kurz vorgestellt.

2.3.1 Beschreibung eines Strahls

Wie von Shirley et al. [Shi+19a] beschrieben, ist ein dreidimensionaler Strahl (*ray*) eine Halbgerade, eine von einem Punkt ausgehende Gerade, welche auf einer Seite ins Unendliche geht. Beschrieben werden kann solch ein Stahl als Parameterform einer Linie, repräsentierbar als gewichtetes Mittel zweier Punkte A und B.

$$P(t) = (1 - t)A + tB \tag{1}$$

[10] `https://www.khronos.org/spir/`, letzter Abruf: 5. Juni 2019

[11] `https://www.khronos.org/registry/vulkan/#repo-docs`, letzter Abruf: 5. Juni 2019

Hierbei kann der Strahl als Funktion $P(t)$ gesehen werden, welche für einen Parameter t einen beliebigen Punkt auf der Linie zurückgibt.

In der Praxis wird oft anstelle von zwei Punkten zur Repräsentation ein einzelner Punkt in Kombination mit einem Richtungsvektor verwendet. Die Gleichung kann dann wie folgt geschrieben werden:

$$P(t) = O + t\vec{d} \tag{2}$$

Der Punkt O, vormals A ist der Ursprungspunkt (*origin*) des Strahls, der Vektor $\vec{d}$ der Richtungsvektor $B - A$ und t bleibt der Parameter welcher die Schrittweite auf dem Strahl angibt. In der Praxis ist es sinnvoll, den Richtungsvektor $\vec{d}$ als Einheitsvektor festzulegen, damit beispielsweise Winkel ohne vorhergehende Normalisierung berechnet werden können und damit der Parameter t dem Abstand zum Ursprungspunkt des Strahls entspricht.

Eine weitere in der Praxis relevante und auch in der RTX-API (siehe Abschnitt 2.1.2) umgesetzte Einschränkung ist das Begrenzen eines Strahls auf ein relevantes Intervall, in dem ein gefundener Schnittpunkt sinnvoll nutzbar ist. So werden zwei Werte, genannt t_{min} und t_{max}, je nach Anwendungsfall konkret festgelegt. Diese Werte legen die für Schnittpunkte relevanten Werte von t auf $t \in [t_{min}, t_{max}]$ fest, was bedeutet, dass ein Schnittpunkt außerhalb dieser Werte ignoriert wird.

Um einen Strahl in der Praxis, wie beispielsweise in der RTX-API, zu beschrieben werden der Ursprungspunkt O, der Richtungsvektor $\vec{d}$ und die Intervallgrenzen t_{min} und t_{max} benötigt.

2.3.2 Generieren von Strahlen

Ray Tracing-Algorithmen lassen sich historisch gesehen in mehrere Kategorien einteilen. Bei klassischen Ray Tracing nach Whitted [Whi80] werden bei einem Schnittpunkt je nach Material mehrere neue Strahlen erzeugt. So wird beispielsweise bei einem Objekt aus Glas ein Reflektionsstrahl und ein Refraktionsstrahl erzeugt und beide Strahlen werden weiter verfolgt und können ihrerseits neue Strahlen erzeugen. Lichtquellen sind dabei Punkte oder Richtungen, aber keine Flächen.

Bei Ray Tracing nach Cook, Porter und Carpenter [CPC84] können für verschiedene Effekte mehrere Strahlen stochastisch generiert und verschickt werden. So können weiche Schatten bei flächigen Lichtquellen, wie in Abschnitt 2.5 beschrieben, berechnet werden. Außerdem entstehen durch das Verschicken mehrerer Strahlen für eine Reflexion entsprechend der Oberflächeneigenschaften nicht nur perfekt spiegelnde, sondern beispielsweise auch glänzende Reflexionen. Mehr zu Reflexionen ist in Abschnitt 2.6 erklärt.

Ray Tracing nach Kajiya [Kaj86], auch *path tracing*, bedeutet, dass mehrere Strahlen verschickt werden, welche bei einem Schnittpunkt jeweils nur einen neuen Strahl generieren. Dabei wird abhängig von den Materialeigenschaften des geschnittenen Objekts einer der möglichen Strahlen ausgewählt.

So entstehen „Pfade“ durch die Szene, deren Farbwerte am Ende zu dem Wert des Pixels verrechnet werden.

Bei hybridem Ray Tracing, welches in dieser Arbeit behandelt wird, bedienen sich aktuelle Algorithmen und Verfahren je nach Anspruch der verschiedenen Arten des Ray Tracing. Wie in Abschnitt 3 beschrieben, werden von verschiedenen Verfahren beispielsweise harte Schatten berechnet, was einem Whitted-Ansatz entspricht, oder weiche Schatten nach einem an Cook angelehnten Prinzip sowie Reflexionen, welche sich je nach Umsetzung an Cook oder Kajiya orientieren.

2.3.3 Nutzen von Rasterisierung als Primärstrahl

Die gemeinsame Voraussetzung für hybrides Ray Tracing ist das Nutzen eines Rasterisierungs-Renderpasses als Primärstrahl, also für den Strahl beim Ray Tracing, welcher von der Kamera aus in die Szene geschossen wird. Dieses Verfahren kann auch von klassischen Ray Tracing-Anwendungen als Beschleunigung benutzt werden, ist für hybrides Ray Tracing allerdings unerlässlich, da für die Kombination mit Rasterisierungstechniken, wie zum Beispiel *screen space ambient occlusion*, ein G-Buffer mit Informationen über die Szenengeometrie im Bildraum benötigt wird. Dieser wird dann für Ray Tracing-Techniken und Rasterisierungs-Techniken je nach Effekt gleichmaßen als Ausgangspunkt genutzt.

2.3.4 Finden von Geoemtrieschnittpunkten

Der Funktionsweise von Ray Tracing liegt zugrunde, dass Strahlen versendet und deren Schnittpunkte mit der anzuzeigenden Geometrie gefunden werden. Anhand der Position und den Oberflächeneigenschaften der geschnittenen Geometrie wird so ein Bild erzeugt. Zur Berechnung des Schnittpunkts werden normalerweise Gleichungssysteme aus der Gleichung des Strahls und der zu schneidenden Geometrie, welche im Normalfall aus Dreiecken besteht, gelöst. Bei Nutzung der RTX-API, wird die Berechnung des Schnittpunkts, wie in Abschnitt 2.1.2 beschrieben, von der Programmierschnittstelle übernommen und im Normalfall durch Hardware beschleunigt. Außerdem ist es möglich, andere Schnitttests als den mit einem Dreieck durchzuführen und eigene Shaderprogramme zu schreiben, welche diese Schnitttests berechnen.

2.3.5 Beschleunigungsdatenstrukturen

Da das Testen jedes Strahls auf einen Schnittpunkt mit jedem Geometrieelement in der Szene aufwendig ist und nicht jedes Geometrieelement für einen Schnittpunkt mit jedem Strahl in Frage kommt, werden von Ray Tracing-Anwendungen sogenannte Beschleunigungsdatenstrukturen (auch *acceleration structures*) benutzt. In der RTX-API kommt wie in Abschnitt 2.1.3 beschrieben, eine *bounding volume hierarchy* zum Einsatz.

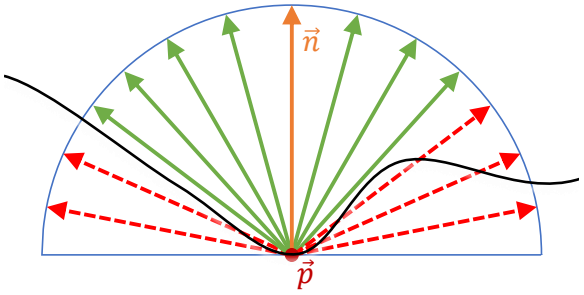


Abbildung 4: Die Umgebungsverdeckung am Punkt $\vec{p}$: Der Anteil der verdeckten Strahlen (rot) an der Gesamtanzahl der Strahlen innerhalb der an der Normale $\vec{n}$ ausgerichteten Halbkugel.

2.4 Umgebungsverdeckung

Umgebungsverdeckung nach Zhukov, Iones und Kronin [ZIK98], auch *Ambient Occlusion*, kurz *AO*, approximiert das einfallende Umgebungslicht auf einer diffusen Oberfläche durch die Evaluation der Verdeckung durch andere Geometrie aus allen Hemisphären-Richtungen, wie in Abbildung 4 dargestellt.

$$A = 1 - \frac{1}{2\pi} \int_{\Omega} V(\vec{\omega}) W(\vec{\omega}) \mathrm{d}\omega \tag{3}$$

In Formel 3 dargestellt ist die unter anderem von Bavoil, Sainz und Dimitrov [BSD08] benutzte Beschreibung der Umgebungsverdeckung A, bei der die Sichtbarkeitsfunktion V über den an der Normalen ausgerichteten Halbraum evaluiert wird. Die Sichtbarkeitsfunktion wird 1, wenn ein Strahl ausgehend vom aktuell betrachten Punkt in Richtung $\vec{\omega}$ einen Schnittpunkt mit der Szenengeometrie hat und 0, falls das nicht der Fall ist. W ist eine Abschwächungsfunktion, welche beispielsweise den Abstand zur gefundenen verdeckenden Geometrie benutzt, um verschieden weit entfernte Schnittpunkte entsprechend zu gewichten.

Zum Berechnen des Umgebungsverdeckungswertes existieren verschiedene annähernde Ansätze, welche auf dem Verfahren von Mittring [Mit07] aufbauen, welches Tiefeninformationen im Bildraum nutzt, um die Verdeckung ohne wirkliche dreidimensionale Geometrieabfragen zu approximieren, Ein Beispiel dafür ist der Ansatz von Bavoil, Sainz und Dimitrov [BSD08], welches zusätzlich den Öffnungswinkel des unverdeckten Teils des Halbraums zur Gewichtung nutzt.

Durch Ray Tracing lässt sich die Umgebungsverdeckung ermitteln, indem zufällig gewählte Strahlen mit Richtung $\vec{\omega}$ aus Formel 3 generiert und das Vorhandensein eines Schnittpunkts geprüft wird. Das Integral aus Formel 3 wird so statistisch durch Monte Carlo-Integration gelöst.

Ein hybrider Ansatz, bei dem durch Ray Tracing ermittelte Umgebungsverdeckung mit der im Bildraum approximierten Umgebungsverdeckung kombiniert wird, wurde von Bavoil et al. [Bav+18] vorgestellt.

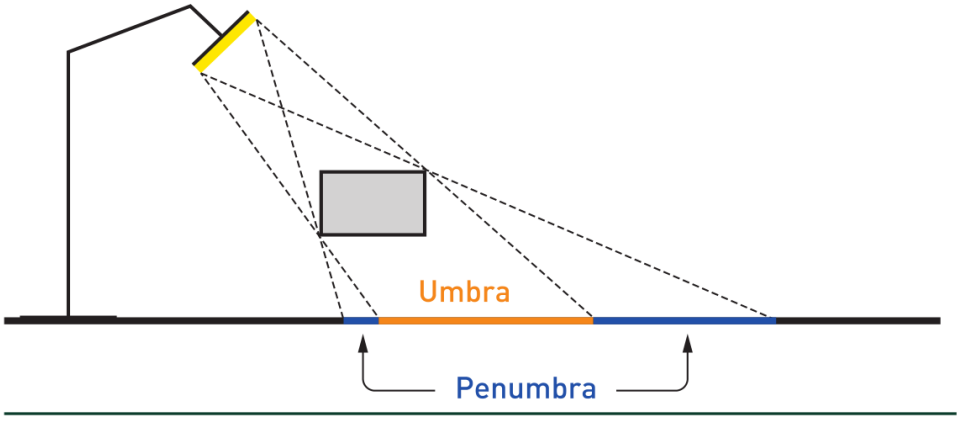


Abbildung 5: Die Schatten-Arten bei einer flächigen Lichtquelle: Halbschatten (Penumbra) und Vollschatten (Umbra). Bildquelle: Haines und Shirley [HS19]

2.5 Schatten

Die Bestimmung, ob an einer Stelle im Raum Schatten ist, wird pro Lichtquelle entschieden und besteht aus einem Sichtbarkeitstest. Wird die Lichtquelle als Punkt im Raum angenommen, gibt es entweder eine Sichtbarkeit, wenn nichts die Strecke von dem behandelten Punkt zur Lichtquelle kreuzt, oder den Fall einer totalen Verdeckung, wenn die Strecke zwischen Raumpunkt und Lichtquelle von weiterer Geometrie gekreuzt wird. Wird die Lichtquelle als Fläche oder dreidimensionale Geometrie betrachtet, kann es zum Fall einer partiellen Verdeckung beziehungsweise Sichtbarkeit kommen. Wird die Sichtbarkeit geprüft, kann ein Punkt nun im Schatten (Umbra) bei vollständiger Verdeckung, im Halbschatten (Penumbra) bei partieller Sichtbarkeit der Lichtquelle, oder außerhalb des Schattens bei vollständiger Sichtbarkeit der Lichtquelle liegen. Dies wird in Abbildung 5 dargestellt.

Rendering-Techniken, welche dieses Sichtbarkeitsproblem lösen wollen, um den Schatten für jeden sichtbaren Punkt in der Szene zu berechnen, funktionieren beispielsweise bei *shadow mapping* so, dass die Szene aus Sicht der Lichtquelle gerendert wird und dann der Abstand von der Lichtquelle zur nächstgelegenen Szenengeometrie für jeden Pixel im Tiefenbuffer steht. Ist der Abstand des auf Schatten zu testenden Punktes in der Szene höher als der korrespondierende Wert in der Tiefentextur, kann vom Schattenfall ausgegangen werden. Dieses Verfahren lässt sich wie von Boksansky, Wimmer und Bittner [BWB19] beschrieben nicht trivial auf flächige Lichtquellen erweitern, außerdem werden Halbschattenwerte oft durch Filterung approximiert und das Verfahren leidet unter der Tatsache, dass die räumlichen Auflösungen für die sichtbaren Punkte in der Szene nicht der Auflösung der Informationen im Tiefenbuffer entsprechen.

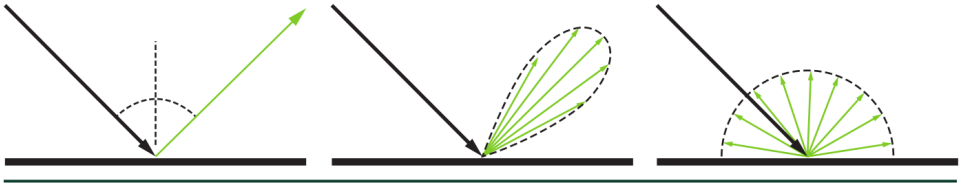


Abbildung 6: Reflexions-Typen von links nach rechts: Perfekte Spiegelung, glänzende Reflexion, diffuse Reflexion. Bildquelle: Haines und Shirley [HS19]

Das Problem der Sichtbarkeit lässt sich durch Ray Tracing intuitiver lösen, da durch Ray Tracing direkt herausgefunden werden kann, ob es Geometrieschnittpunkte auf einer Strecke gibt, in diesem Fall auf der Strecke von dem zu testenden Punkt in der Szene zur Lichtquelle. Es wird also ein Strahl verschossen, wird ein Schnittpunkt gefunden, ist die Lichtquelle verdeckt und der Punkt liegt im Schatten. Dies lässt sich außerdem auf flächige Lichtquellen erweitern, indem mehrere Strahlen zu Punkten auf der Oberfläche der Lichtquelle verschickt werden und die Sichtbarkeitswerte gemittelt werden.

2.6 Reflexionen

Reflexionen können wie von Haines und Shirley [HS19] beschrieben in drei Kategorien unterteilt werden: Perfekte Spiegelungen, glänzende Reflexionen und diffuse Reflexionen. Diese sind in Abbildung 6 gezeigt. Bei perfekten Spiegelungen wird ein einfallender Sicht- oder Lichtstrahl an der Oberflächennormalen gespiegelt. Bei einer glänzenden (*glossy*) Reflexion, welche zum Beispiel bei einer polierten Metalloberfläche auftritt, werden die gespiegelten Strahlen etwas gestreut. Ein Durchschnittswert der Ergebnisse der Strahlen innerhalb der Streuung ergeben den Reflexionswert, reflektierte Objekte sind unscharf erkennbar. Je nach Rauheit der Oberfläche werden die reflektierten Strahlen mehr oder weniger weit gestreut. Dies lässt sich zum Beispiel über ein Microfacetten-Oberflächenmodell wie in Abschnitt 2.7 beschrieben approximieren. Bei den diffusen Reflexionen wird Licht beim Auftreffen auf die Oberfläche in alle Richtungen gestreut. Eine optische Reflexion ist hierbei nicht erkennbar.

2.7 Cook-Torrance-Beleuchtungsmodell

In dieser Arbeit wird das Beleuchtungsmodell nach Cook und Torrance [CT82] verwendet. Es handelt sich dabei um ein Microfacetten-Modell, welches mit dem sogenannten *Metallic-Roughness-Workflow* arbeitet. Die Materialen sind dabei wie von Karis [Kar13] beschrieben definiert über eine unbeleuchtete Grundfarbe, die *base color*, auch *albedo* genannt, die *roughness* (Rauheit) der

Oberfläche zwischen 0 und 1, und die *metalness*, welche entscheidet ob das Material als Metall oder Nichtmetall behandelt wird. Metalle haben einen zusätzlichen Parameter F_0, Nichtmetalle haben einen konstanten F_0-Wert von $0,04$.

$$f(\vec{l},\vec{v}) = \frac{D(\vec{h})F(\vec{v},\vec{h})G(\vec{l},\vec{v},\vec{h})}{4(\vec{n}\cdot\vec{l})(\vec{n}\cdot\vec{v})} \tag{4}$$

Das Cook-Torrance-Beleuchtungsmodell für spiegelnde Oberflächen (*specular*) ist in Gleichung 4 beschrieben. Dabei ist $\vec{h}$ der *halfway*-Vektor, der Vektor zwischen dem Lichtvektor $\vec{l}$ und dem Blickvektor $\vec{v}$. Der Vektor $\vec{n}$ beschreibt die Oberflächennormale des zu beleuchtenden Punktes und wird als Funktionsparemeter nicht weiter erwähnt, da sie im Gegensatz zu den anderen Vektoren als Parameter nicht austauschbar ist.

Die Funktion $D(\vec{h})$ ist die Normalverteilungsfunktion, welche den von den Oberflächenrauheit abhängigen Anteil der Microfacetten approximiert, die senkrecht zum Vektor $\vec{h}$ stehen. Diese Arbeit nutzt wie von Karis [Kar13] beschrieben die *Trowbridge-Reitz GGX*-Funktion. Hierbei ist $\alpha = roughness^2$.

$$D(\vec{h}) = \frac{\alpha^2}{\pi((\vec{n}\cdot\vec{h})^2(\alpha^2-1)+1)^2} \tag{5}$$

Die Geometriefunktion $G(\vec{l},\vec{v},\vec{h})$ approximiert den Anteil der Microfacetten, welche sich gegenseitig verschatten, ebenfalls abhängig von der Rauheit der Oberfläche. In dieser Arbeit wird wie von Karis [Kar13] vorgeschlagen eine angepasste Version des Schlick-Modells nach Schlick [Sch94] benutzt, wie in Gleichung 6 beschrieben, mit k wie in Gleichung 7 beschrieben.

$$G_{SchlickGGX}(\vec{v}) = \frac{\vec{n}\cdot\vec{v}}{(\vec{n}\cdot\vec{v})(1-k)+k} \tag{6}$$

$$k = \frac{(\alpha+1)^2}{8} \tag{7}$$

Damit wie in Gleichung 4 beschrieben $G(\vec{l},\vec{v},\vec{h})$ berechnet werden kann, wird die Methode von Smith, beschrieben von Walter et al. [Wal+07] genutzt.

$$G(\vec{l},\vec{v},\vec{h}) = G_{SchlickGGX}(\vec{l})G_{SchlickGGX}(\vec{v}) \tag{8}$$

Für den Fresnel-Term $F(\vec{v},\vec{h})$, welcher das Verhältnis von reflektiertem zu refraktiertem Licht beschreibt, wird die Fresnel-Schlick-Approximation genutzt. Hier kommt der zuvor beschriebene Parameter F_0 hinzu, welcher die Reflektivität einer Oberfläche bei einem Blickwinkel von 0° beschreibt.

$$F_{Schlick}(\vec{v},\vec{h}) = F_0 + (1-F_0)(1-(\vec{h}\cdot\vec{v}))^5 \tag{9}$$

Die in Gleichung 10 beschriebenen Reflektanzgleichung, einem Spezialfall der Rendering-Gleichung von Kajiya [Kaj86], wie beispielsweise beschrieben von Hoffman, beschreibt die ausgehende Strahldichte *radiance* L_o an einem Punkt p mit ausgehendem Raumwinkel w_0 und einfallendem Raumwinkel w_i mit einem Integral über die Halbkugel Ω. Dabei ist f_r die bidirektionale Reflektanz-Verteilungsfunktion (englisch *bidirectional reflectance distribution function*, abgekürzt BRDF), welche die hier vorgestellte Cook-Torrance-BRDF sein kann. Das einfallende Licht ist L_i.

$$L_o(p, \vec{w_o}) = \int_\Omega f_r(p, \vec{w_i}, \vec{w_o}) L_i(p, \vec{w_i}) \vec{n} \cdot \vec{w_i} \mathrm{d}\vec{w_i} \tag{10}$$

Das gesamte Beleuchtungsmodell eingesetzt in die Reflektanzgleichung ist in Gleichung 11 beschrieben. Hier ist k_d der Anteil des diffusen Reflektierens des Materials: Das Gegenstück zum glänzenden (*specular*) Anteil k_s, welches durch den Fresnel-Term F angegeben wird. Durch das Vorhandensein des Fresnel-Terms in der Gleichung wird k_s nicht mehr explizit benötigt, nur noch kd mit $k_d = 1 - k_s$. Entsprechend ist $\frac{c}{\pi}$ das Lambert-Modell für den diffusen Anteil der BRDF, bei dem c für die Grundfarbe (*albedo*) des Materials steht.

$$L_o(p, \vec{w_o}) = \int_\Omega (k_d \frac{c}{\pi} + \frac{DFG}{4(\vec{w_o} \cdot \vec{n})(\vec{w_i} \cdot \vec{n})}) L_i(p, \vec{w_i}) \vec{n} \cdot \vec{w_i} \mathrm{d}\vec{w_i} \tag{11}$$

2.8 Monte Carlo-Integration

Monte Carlo-Integration ist wie beispielsweise von Pharr [Pha19b] beschrieben eine numerische Lösungsmethode für Integrale, welche beim Ray Tracing verwendet wird. Dabei kommt zum Einsatz, dass der Erwartungswert der Summe von k Stichproben der Lösung eines Integrals unter den folgenden Gegebenheiten entspricht:

$$E\left[\frac{1}{k} \sum_{i=1}^{k} \frac{f(X_i)}{p(X_i)}\right] = \int_{[0,1]^n} f(x) \mathrm{d}x \tag{12}$$

Die Zufallszahlen X_i sind nach der Dichtefunktion $p(X_i)$ verteilt. Dadurch, dass durch die Verteilung der Zufallszahlen geteilt wird, wird der Einfluss der Zufallszahlen auf das Ergebnis ausgeglichen. Wird die Summe nun aus den Ergebnissen von $f(X_i)$ und $p(X_i)$ wie in der Formel beschrieben berechnet, erhält man eine Abschätzung für den Wert des Integrals. Kombiniert mit *importance sampling*, bei dem die Idee ist, die Dichtefunktion so zu wählen, dass sie der zu integrierenden Funktion möglichst ähnelt.

3 Stand der Forschung

In den folgenden Abschnitten werden momentan vorhandene Ansätze für hybride Ray Tracing-Pipelines vorgestellt. Außerdem wird der Stand der Forschung bezüglich des Einsatzes von Ray Tracing für andere Echtzeitgrafik-Techniken kurz präsentiert.

3.1 PICA PICA

Barré-Brisebois et al. [Bar+19] stellen mit der *PICA PICA* genannten Demonstrationsanwendung eine der ersten Echtzeitanwendungen vor, welche die in dieser Arbeit verwendete RTX-API (siehe Abschnitt 2.1.2) in Kombination mit DirectX nutzt. Die implementierte Pipeline wird vorgestellt und ausgewählte Teilschritte, nämlich Schatten, Reflexionen, Umgebungsverdeckung, Durchsichtigkeit und globale Beleuchtung, werden näher erläutert. Außerdem liegt ein besonderer Fokus auf dem *Denoising* (Entrauschen) der Ergebnisse.

Für durch Ray Tracing ermittelte Schatten werden von Barré-Brisebois et al. [Bar+19] Pseudocode für harte und weiche Schatten vorgestellt und das gewählte Denoising-Verfahren, *spatiotemporal variance-guided filtering* (SVGF) von Schied et al. [Sch+17] erwähnt. Des Weiteren wird die Methode, mit der farbige Schatten von transparenten Objekten akkumuliert werden, erwähnt.

Bei den Reflexionen wird die Methode, mit der die Reflexionsstrahlen generiert werden erläutert: Zuerst wird mit der entsprechenden Gewichtung eine der Schichten des Materials eines Objekts, dessen Reflexion evaluiert wird, ausgewählt. Dann wird mit der BRDF ein Strahl mit einer Quasizufalls-Sequenz generiert. Dafür wird nur der *NDF*-Teil des Beleuchtungsmodells, vorgestellt in Abschnitt 2.7, benutzt. Für die Reflexionen werden bei der halben Auflösung des finalen Bildes nur jeweils ein Strahl pro Bild verschickt, was demnach einen „viertel Strahl“ pro Pixel pro Bild für Reflexionen bedeutet. Um mit so wenig Reflexionsinformationen pro Pixel trotzdem ein visuell hochwertiges Ergebnis zu erhalten, wird erst ein räumlicher Filter angewandt, mit dem auf die volle Auflösung skaliert wird. Zudem wird ein gewichtetes Mittel berechnet, wodurch Lücken gefüllt und das Rauschen reduziert wird. Danach wird ein Bilateralfilter mit einer durch die vorher geschätzte Varianz geleiteten Filtergröße benutzt, um das Bild weiter zu glätten.

Zum Ende der Pipeline werden mit temporaler Kantenglättung die Ergebnisse über die Zeit akkumuliert, was mit Problemen bei Bewegungen und dynamischen Objekten in der Szene verbunden ist. Das Umgehen der dabei entstehenden Bewegungsartefakte wird ebenfalls beschrieben.

Ein Ansatz für Umgebungsverdeckung, welcher auf einer Kombination von Ray Tracing und Umgebungsverdeckung im Bildraum nach Mittring [Mit07] basiert, wird ebenfalls von Bavoil et al. [Bav+18] vorgestellt und die Ergebnisse werden mit denen anderer Techniken verglichen.

Außerdem werden Ansätze für teilweise durchsichtige Objekte und eine Technik vorgestellt, welche Ray Tracing für diffuse globale Beleuchtung benutzt, indem die Szene mit *Surfeln* (Oberflächen-Scheiben) approximiert wird.

3.2 3DMark Port Royal

Das *Port Royal* genannte Benchmark-Programm von *3DMark* benutzt ebenfalls RTX-Technologie. Die Anwendung wird von UL Benchmarks [UL 19] beschrieben. Ray Tracing wird hierbei für Schatten, Photon Mapping und Reflexionen genutzt. Für die Schatten wird auf der *compute queue* ein einzelner Strahl pro Pixel zu jeder Lichtquelle verschickt. Allerdings wird hierbei die Lichtquelle nicht abgetastet und es wird kein Weichzeichnungsfilter benutzt, demnach werden nur harte Schattenkanten berechnet.

Das Photon Mapping soll eine globale Beleuchtung approximieren und benutzt Ray Tracing, um die Photonen zu verschicken. Eine genaue Beschreibung der Herangehensweise wird nicht von UL Benchmarks [UL 19] nicht gegeben.

Für die Reflexionen wird ausgehend von jedem Pixel, welcher einen Rauheits-Schwellwert überschreitet, ein Reflexionsstrahl generiert. Die Stellen, an denen der Strahl die Szenengeometrie trifft, werden gespeichert und für das Entnehmen der Reflexionswerte aus vorberechneten *Environment Maps* (Umgebungskarten) benutzt. Nur an stark spiegelnden Oberflächen und an Stellen, welche in den Environment Maps nicht einsehbar sind, wird ein Reflexionswert neu berechnet, das heißt, die getroffene Stelle auf der Szenengeometrie wird erneut beleuchtet. Auf den Ergebnissen wird ein räumlich-zeitlicher (*spatio-temporal*) Filter angewandt.

Neben den Ray Tracing-Komponenten der Pipeline werden noch deutlich mehr Standard-Techniken verwendet, wie das Rendern in den G-Buffer, Umgebungsverdeckung, *Shadow Maps*, Temporales Anti-Aliasing, Depth of Field, Bloom und Weitere. Dies erlaubt auch das Auslagern der Schattenberechnung auf die *compute queue*, da die Berechnungen auf der *graphics queue* genug Zeit in Ansrpuch nehmen, dass ein Auslagern von anderen Berechnungen keine zusätzlichen Mehraufwand mit sich bringt.

3.3 Weitere auf der RTX-Technologie aufbauende Techniken

Weitere Ansätze für hybrides Ray Tracing sind beispielsweise das von Aalto [Aal18] vorgestellte Verfahren, in dem Ray Tracing unter anderem dazu verwendet wird, um vorher berechnete statische Beleuchtungsdaten an der richtigen Stelle zu abzurufen.

Außerdem wurden einige Techniken veröffentlicht, welche auf der RTX-Technologie aufbauen, aber nicht direkt ein hybrider Rendering-Ansatz sind, sich aber in einen solchen einfügen lassen würden. Dazu gehört die Technik

von McGuire [McG19], bei der *light probes* mittels Ray Tracing um Sichtbarkeitsinformationen erweitert werden. Eine weitere Technik ist die von Marrs et al. [Mar+18], bei der zur Kantenglättung erst Kanten im Bild erkannt und dann mit Ray Tracing die Farbwerte an den Kanten exakter berechnet werden, um Weichzeichnungs- und Doppelbildartefakte, welche bei normaler temporaler Kantenglättung entstehen, zu verhindern.

4 Konzept

In den folgenden Abschnitten wird ein Überblick über die in dieser Arbeit erstellte Applikation vermittelt. Die genauere technische Umsetzung folgt danach in Abschnitt 5.

4.1 Aufbau der Applikation

Bei der Anwendung handelt es sich um ein einzelnes Programm, welches aus einer Start-Phase und einem sich wiederholenden Ablauf besteht.

In der Start-Phase wird das verwendete 3D-Modell geladen und für das Rendering vorbereitet, wozu auch das Erstellen der Beschleunigungsdatenstrukturen gehört. Grafikkarten-Ressourcen werden angelegt und typisch für Vulkan werden statische API-Objekte wie Pipelines, Renderpasses, aber auch *command buffer*, welche in jedem *Frame* (gerendertem Bild) garantiert gleich sein werden, bereits angelegt.

In der dauerhaft laufenden Phase der Applikation wird die Synchronisation zwischen den einzelnen Frames abgehandelt, Benutzereingaben werden verarbeitet und in Kamerabewegungen oder Benutzeroberflächen-Aktionen umgesetzt und frame-spezifische *command buffer* werden aufgenommen.

Der Ablauf des Renderings besteht aus den nachfolgend vorgestellten Schritten, die mit ihren Abhängigkeiten in Abbildung 7 dargestellt sind.

G-Buffer Renderpass Zuerst wird die Szenengeometrie mit Rasterisierung gerendert und Szeneninformationen wie Position oder Oberflächennormale werden in den G-Buffer geschrieben.

Schatten mit Ray Tracing Mithilfe der Informationen aus dem G-Buffer werden die Schatten für alle in der Szene befindlichen Lichtquellen berechnet. Dafür kommt mit RTX-Technologie umgesetztes Ray Tracing zum Einsatz. Das Ergebnis wird in Texturen gespeichert.

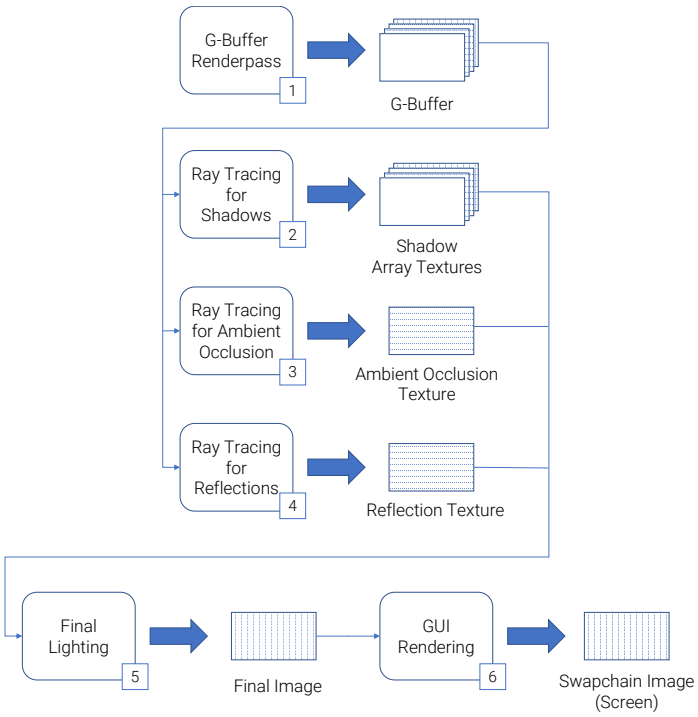


Abbildung 7: Der makroskopische Ablauf der Rendering-Phase der Applikation mit den Abhängigkeiten der einzelnen Schritte voneinander.

Umgebungsverdeckung mit Ray Tracing Ebenfalls aufbauend auf den Informationen aus dem G-Buffer wird für jeden Pixel die Umgebungsverdeckung mit Ray Tracing berechnet. Wie bei den Schatten wird das Ergebnis in Texturen gespeichert.

Reflexionen mit Ray Tracing Für die Reflexionen werden auch die Szeneninformationen aus dem G-Buffer benötigt. Mittels RTX werden Reflexionsstrahlen verfolgt. Bei einem Schnittpunkt wird die direkte Beleuchtung berechnet und ein Schattenfühler, welcher ebenfalls mit RTX verschickt wird, wird ausgesandt. Das Ergebnis wird in einer Farbtextur gespeichert.

Direkte Beleuchtung und Komposition Im letzten Renderpass wird pro Pixel die direkte Beleuchtung aufbauend auf den Informationen aus dem G-Buffer berechnet. Die Texturierung wird durchgeführt und die Informationen aus den Ray Tracing-Durchläufen, also Schatten, Umgebungsverdeckung und Reflexionen werden mit einberechnet. Danach ist das Bild fertig gerendert.

Benutzeroberfläche Nach dem eigentlichen Rendering wird die Benutzeroberfläche über das eigentliche Bild gerendert. Das Gesamtergebnis wird dann zur Präsentation auf dem Bildschirm übergeben.

5 Implementierung

In den folgenden Abschnitten wird die Implementierung der in dieser Arbeit erstellten Applikation in ihren einzelnen Komponenten vorgestellt und erläutert.

5.1 Verwendete Software-Bibliotheken

In dieser Arbeit wird als Grafik-API Vulkan benutzt. Die verwendeten Extensions (siehe Abschnitt 2.2.1) von Vulkan sind `VK_KHR_swapchain` zur Anzeige und `VK_NV_ray_tracing` für Ray Tracing mit RTX-Technologie. Für die Nutzung von Vulkan wird *Vulkan-Hpp*[12] verwendet. *Vulkan-Hpp* ist eine Bibliothek welche die in der Programmiersprache C geschriebene API Vulkan in C++ umsetzt. So lässt sich Vulkan nahtlos in einem modernen C++-Programm nutzen und C-spezifische Eigenheiten fallen weg. Vulkan-Strukturen lassen sich mit Konstruktoren in nur einer Zeile initialisieren und Aufzählungstypen besitzen Typsicherheit. Außerdem werden C++-*Exceptions*, also das Auslösen von Ausnahmen, unterstützt: Das Testen von Rückgabewerten auf Fehler kann nach Wunsch weggelassen werden.

Das Vulkan-SDK[13] (*Software Development Kit*) beinhaltet unter anderem den *Vulkan Loader*, welche die Verbindung der Vulkan-API mit dem Grafiktreiber herstellt, die *Validation Layer*, welche Fehler bei der API-Nutzung feststellen und ausgeben. Diese können auch tagesaktuell heruntergeladen und kompiliert werden. In dieser Arbeit wurde mit der Nutzung des Vulkan-SDKs die stabile Version, welche nur installiert werden muss, verwendet.

Ebenfalls im Vulkan-SDK enthalten sind die Shader-Compiler *glslangvalidator*[14] und *glslc*[15]. Für diese Arbeit wurde *glslc* wegen der Unterstützung vom Einbinden (*include*) von Dateien in andere Dateien benutzt. Bei der Verwendung von Vulkan müssen Shader im Vorhinein oder von der Applikation in die Zwischenrepräsentation *SPIR-V*[16] gebracht werden, um von der API nutzbar zu sein und vom Treiber zum letzendlichen Maschinencode umgesetzt zu werden. Nur durch die Nutzung eines externen Shader-Compilers oder einer Shader-Compiler-Bibliothek können Hochsprachen wie im Fall dieser Arbeit *GLSL*[17] verwendet werden.

Diese Arbeit benutzt außerdem die Bibliothek *Vulkan Memory Allocator*[18]

[12] `https://github.com/KhronosGroup/Vulkan-Hpp`, letzter Abruf: 5. Juni 2019

[13] `https://www.lunarg.com/vulkan-sdk/`, letzter Abruf: 5. Juni 2019

[14] `https://github.com/KhronosGroup/glslang`, letzter Abruf: 5. Juni 2019

[15] `https://github.com/google/shaderc/tree/master/glslc`, letzter Abruf: 5. Juni 2019

[16] `https://www.khronos.org/spir/`, letzter Abruf: 5. Juni 2019

[17] `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.html`, letzter Abruf: 5. Juni 2019

[18] `https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator`, letzter Abruf: 5. Juni 2019

von AMD, welche das manuelle Allokieren von Grafikspeicher, welches bei Vulkan von der Anwendung selbst organisiert werden muss, übernimmt. Bei einer naiven Implementierung kann eine Anwendung schnell an die maximale Anzahl einzeln allokierter Speicherbereiche, welche auf der verwendeten Hardware bei 4096 liegt, stoßen. Statt einzelnen Speicherbereichen für jede Ressource können größere Speicherbereiche allokiert und danach auf verschiedene Ressourcen, wie beispielsweise Buffer, aufgeteilt werden. Diesen Teil, zusammen mit einer vereinfachten Angabe vom Verwendungszweck von Speicher, welcher bei Vulkan angegeben werden muss, übernimmt *Vulkan Memory Allocator*.

Für die Fensterverwaltung wird *GLFW*[19] benutzt. GLFW ist eine Bibliothek zur Fensterverwaltung, welche auf mehreren Betriebssystemen verwendet werden kann und auch dazu benutzt werden kann, einen OpenGL-Kontext zu erstellen. Bei der Verwendung von GLFW in Kombination mit Vulkan wird kein Kontext von GLFW bereitgestellt, dies muss jede Vulkan-Applikation selbst übernehmen.

Für die Benutzerschnittstelle wird *Dear ImGui*[20] verwendet. Diese Bibliothek stellt diverse Benutzeroberflächenelemente zur Verfügung und kann in Programme, welche selbst eine Grafik-API wie OpenGL oder Vulkan verwenden, eingebaut werden. Die Implementation des Renderings der Bibliothek kann auch selbst geschrieben werden, wurde in dieser Arbeit allerdings aus den Beispiel-Implementationen, welche mit der Bibliothek bereitgestellt werden, entnommen.

Die Konsolenausgaben von Ereignisnachrichten oder Debug-Informationen vom Validation Layer werden mithilfe von *spdlog*[21] angezeigt, einer besonders schnellen Bibliothek, die unter anderem Zeitstempel automatisch mit jeder Nachricht ausgibt und auch in der Evaluation, speziell bei den in Abschnitt 6.1 vorgestellten Messungen, dazu benutzt wird, um die Zeitspannen der einzelnen Rendering-Operationen in Dateien zu schreiben.

Zum Laden von 3D-Modellen wird die Bibliothek *Assimp*[22] benutzt. Diese Bibliothek unterstützt mehrere gängige Formate für 3D-Modelle, wie die in dieser Arbeit verwendeten Formate *FBX* und *glTF*. Das Laden der Texturen erfolgt mittels der Bibliothek *stb_image*[23], welche aus einer einzelnen Datei besteht.

Außerdem wird die Mathematikbibliothek *glm*[24] verwendet, um mit Vektoren und Matrizen zu arbeiten.

[19] `https://www.glfw.org/`, letzter Abruf: 5. Juni 2019
[20] `https://github.com/ocornut/imgui`, letzter Abruf: 5. Juni 2019
[21] `https://github.com/gabime/spdlog`, letzter Abruf: 5. Juni 2019
[22] `http://www.assimp.org/`, letzter Abruf: 5. Juni 2019
[23] `https://github.com/nothings/stb`, letzter Abruf: 5. Juni 2019
[24] `https://glm.g-truc.net/`, letzter Abruf: 5. Juni 2019

5.2 Initialisierung von Vulkan

In Vulkan-Applikationen muss das Erstellen eines Fensters, einer Vulkan-Instanz, das Auswählen des Hardware-Gerätes auf dem Vulkan benutzt wird und das Anlegen der zur Präsentation der Rendering-Ergebnisse auf dem Bildschirm benötigten Ressourcen selbst durchgeführt und verwaltet werden. Da der dazu benötigte Code für viele vergleichbare Applikationen gleich ist und wiederverwendet werden kann, ist er in der Klasse `Context` abstrahiert. Dort wird zunächst der Logger und GLFW initialisiert und das Fenster angelegt. Der Ablauf danach ist in den folgenden Abschnitten beschrieben.

5.2.1 Erstellen der Vulkan-Instanz

Nachdem GLFW initialisiert und das Fenster erstellt wurde, kann die Vulkan-Instanz erstellt werden. In dieser werden die Namen der benötigten Validation Layer angegeben und die Vulkan-API-Version wird auf 1.1 festgelegt. Außerdem werden beim Erstellen der Instanz die benutzten *Instance Extensions* angegeben, jene Extensions, welche auf Instanz-Ebene unterstützt werden müssen. In der Applikation wird zwischen einem *Debug*-Kompiliermodus, welcher keine Optimierungen und volle Debug-Informationen enthält, und einem *Release*-Kompiliermodus, welcher voll optimiert ist, unterschieden. Dieser Unterschied wirkt sich normalerweise nur auf den C++-Compiler aus, wird bei der Applikation aber ebenfalls dazu benutzt, Vulkan-Debugging-Funktionalitäten wie die Validation Layer zu steuern. So wird zu Kompilierzeitpunkt der Kompiliermodus abgefragt und mit `if constexpr (...)` aus C++17 werden die für die Valdiation Layer zuständigen Codezeilen im Release-Modus gar nicht mitkompiliert. Dies sorgt für eine bessere Performance im Release-Modus.

Nach Erstellen der Instanz kann der `Debug Messenger` erstellt werden, ein Vulkan-Konstrukt, das dafür zuständig ist, die Probleme, welche vom Validation Layer gefunden werden, auszugeben.

Anschließend wird die `Window Surface` welche zur Anzeige benötigt wird mithilfe von GLFW erstellt.

5.2.2 Phsyical und Logical Device

Danach wird das `Physical Device` ausgewählt. In Vulkan repräsentiert dies die spezifische (Grafik-)Hardware mit der das Programm ausgeführt werden soll. Das Auswählen des `Physical Device` zur Laufzeit der Anwendung und die Unterstützung mehrerer solcher *devices* ist eine Neuerung, welche zum Beispiel in OpenGL nicht ohne Weiteres unterstützt wird. Bei der Auswahl des `Physical Device` wird von der Applikation darauf geachtet, eines auszuwählen, welches die benötigten `Queues` und `Device`-spezifischen Extensions, wie in diesem Fall `VK_NV_ray_tracing` unterstützt. Dafür ist

die Funktion `isDeviceSuitable(...)` zuständig. Diese manuelle Auswahl ist vor allem bei Geräten mit mehreren Grafikkarten relevant, wie beispielsweise Notebooks oder Computern, bei denen eine integrierte Grafikkarte zusätzlich zu einer diskreten Grafikkarte vorhanden ist. Außerdem ist das Programm in der Lage, vor dem eigentlichen Beginn des Hauptteils des Programms kontrolliert abzubrechen, wenn keine Hardware vorhanden ist, welche die benötigte Funktionalität unterstützt, anstatt beim Versuch der Ausführung nicht unterstützter Funktionalitäten abzustürzen.

Das `Logical Device` in Vulkan, auch nur `Device` genannt, stellt die Verbindung der mit dem `Physical Device` repräsentierten Hardware dar. Es kann mehrere `Logical Devices` geben, welche alle einen eigenen Zustand und eigene Ressourcen besitzen. Im `Logical Device` wird der Zugriff auf die *queue families* geregelt, welche mehrere *queues* enthalten.

Jede *queue family* stellt eine oder mehrere *queues* zur Verfügung, welche alle dieselben Funktionalitäten unterstützen. Diese Funktionalitäten beziehen sich auf die Bereiche *graphics* für Rasterisierungs-Pipelines, *compute* für GPGPU-Funktionalitäten wie Compute Shader, *transfer* für das asynchrone Kopieren von Speicher von CPU-Speicher (auch *RAM* oder *host memory*) in GPU-Speicher (auch *VRAM* oder *device memory*) und *presenting* für das Anzeigen auf dem Bildschirm. Im konkretem Beispiel der für diese Arbeit verwendeten Hardware, genauer beschrieben in Abschnitt 6.1, äußert sich das folgendermaßen:

- Queue Family 0
  - Anzahl Queues: 16
  - Unterstützte Funktionalität:
    - Graphics
    - Compute
    - Transfer
    - Presenting
- Queue Family 1
  - Anzahl Queues: 1
  - Unterstützte Funktionalität:
    - Transfer
- Queue Family 2
  - Anzahl Queues: 8
  - Unterstützte Funktionalität:
    - Compute

Das Besondere an den *queue families* ist, dass Befehle, welche auf verschiedenen *queue families* ausgeführt werden, parallel zueinander auf der Hardware ausgeführt werden können. So ist es in dem angegebenen Beispiel möglich, gleichzeitig zu einer Grafik-Operation auf einer *queue* in *queue familiy* 0 eine Compute-Operation auf einer *queue* in *queue familiy* 1 auszuführen. Die Synchronisierung dieser Operationen muss bei Vulkan von der Applikation übernommen werden.

Die Indizes von allen *queue families* werden durch den Aufruf der Funktion `findQueueFamilies(...)` abgerufen und werden zum Erstellen des `Logical Device` in der Funktion `createLogicalDevice()` verwendet, da dabei alle *queue families*, die man später mit dem `Logical Device` verwenden kann, angegeben werden müssen. Außerdem müssen manche Funktionalitäten beim Erstellen des `Logical Device` explizit angegeben werden, wie zum Beispiel die anisotrope Filterung von Texturen oder die Unterstützung von `multiDrawIndirect`. Außerdem müssen hier die verwendeten Validation Layer aktiviert werden. Zusätzlich werden in der Funktion die folgenden *queues* gespeichert:

```
m_presentQueue =
  m_device.getQueue(indices.presentFamily.value(), 0);
m_graphicsQueue =
  m_device.getQueue(indices.graphicsFamily.value(), 0);
m_transferQueue =
  m_device.getQueue(indices.transferFamily.value(), 0);
m_computeQueue =
  m_device.getQueue(indices.computeFamily.value(), 0);
```

Dazu ist anzumerken, dass auf der zuvor vorgestellten, verwendeten Hardware die hier gezeigte `graphicsFamily` und `presentFamily` dieselbe *queue family* repräsentiert. Durch die logische Trennung im Quellcode ist es allerdings möglich, auch Hardware zu nutzen, bei denen diese *queue families* unterschiedlich sind.

Nach dem Erstellen des `Logical Device` wird das `VmaAllocator`-Objekt erstellt, welches von der *Vulkan Memory Allocator*-Bibliothek bereitgestellt wird.

5.2.3 Erstellen der Swapchain

Zur Anzeige auf dem Bildschirm wird die sogenannte *swap chain* verwendet, welche bei Vulkan von der Applikation selbst verwaltet werden muss. Das Prinzip der *swap chain* funktioniert so, dass die *swap chain* aus mehreren Bildern besteht, in welche von der Applikation geschrieben wird. Das fertig beschriebene Bild wird dann auf dem Bildschirm angezeigt.

Zum Anlegen der *swap chain* werden zuerst die benötigten Informationen wie das Format und der Präsentiermodus abgerufen. Dabei kommen Funktionen zum Einsatz, die die verfügbaren Formate und Präsentiermodi abfragen und dann das Bestmögliche auswählen. Bei dieser Applikation wird

der Präsentiermodus `Mailbox` präferiert, welcher die zu präsentierenden Bilder hintereinander einreiht, allerdings im Vergleich zum Modus `Fifo` nicht blockiert, wenn die Schlange voll ist, sondern die Bilder stattdessen mit den nachrückenden Bildern ersetzt. Dieser ist deshalb empfohlen für *triple buffering*, welches in dieser Anwendung benutzt wird.

Außerdem wird beim Anlegen der *swap chain* darauf geachtet, dass die Bilder zwsichen der *graphics queue* und der *present queue* geteilt werden können. Falls diese aus derselben *queue family* kommen, wird der Modus `vk::SharingMode::eExclusive` verwendet, ansonsten der Modus `vk::SharingMode::eConcurrent`. Dies sorgt dafür, dass die Bilder zwischen *queue families* geteilt werden können.

Nach Erstellen der *swap chain* müssen noch die *image views* für die darin erhaltenen Bilder erstellt werden. Dies wird benötigt, um diese als Rendering-Ziel zu benutzen. Mehr zu *image views* ist in Abschnitt 5.3.3 nachzulesen. Dies schließt das Initialisieren der Klasse `Context` und damit das Initialisieren von Vulkan ab.

5.3 Anlegen von Vulkan-Ressourcen

Nach dem Initialiseren der Klasse `Context` wird der Rest der vor allem Anwendungsspezifischen Vulkan-Ressourcen angelegt. Das funktioniert in der hier vorgestellten Anwendung so, dass es eine anwendungsspezfische Klasse gibt, in diesem Fall `RTCombinedApp`, welche von einer Klasse `BaseApp` erbt. Die Klasse `BaseApp` stellt wiederverwendbare Funktionalität bereit. In den folgenden Abschnitten wird der Ablauf der Initialisierung der eigentlichen Anwendung erläutert. Da die sich allgemeine und anwendungsspezifische Funktionalitäten wegen internen Abhängigkeiten in ihrer Reihenfolge vermischen, wird hier vor allem auf die Reihenfolge eingegangen, statt die Funktionalitäten der beiden Klassen `RTCombinedApp` und `BaseApp` getrennt vorzustellen.

5.3.1 Laden des 3D-Modells

Zuerst wird das 3D-Modell von der Festplatte in den Arbeitsspeicher geladen. Dafür kommt bis zu diesem Punkt keine Vulkan-Funktionalität zum Einsatz, es wird lediglich *Assimp* benutzt, um das 3D-Modell zu laden. Dies passiert in der Klasse `PBRScene`. Geladen werden Vertices, Indices, Transformationen, Normalen und Texturkoordinaten aller Objekte in der Szene, welche im *glTF*- oder *FBX*-Format vorliegen kann. Außerdem werden Materialien und bis zu diesem Zeitpunkt die Dateipfade der Texturen geladen. Wichtig ist, dass die enthaltenen Texturen und Materialien die Informationen die für das im Abschnitt 2.7 vorgestellte Beleuchtungsmodell enthalten.

5.3.2 Erstellen der Command Pools

Command pools werden in Vulkans verzögertem Ausführungsmodell dazu benutzt, um darin *command buffer* zu allokieren und sind einzelnen *queues* zugeordnet. Der Einfachheit halber wird in der Klasse `BaseApp` jeweils ein *command pool* für jede zuvor im `Context` angelegten *queue* kreiert. Die *command pools* werden später benutzt, um beliebige *command buffer* zu allokieren, die dann *commands*, also grafikhardwarespezfische Befehle, ausführen. Die beim Erstellen des *command pool* angegebene *queue* bestimmt, auf welcher *queue* die *commands* im *command buffer*, der aus dem *command pool* allokiert wurde, ausgeführt werden können.

Nach dem Erstellen der *command pools* können nun frühestens an diesem Punkt im Programm Vulkan-Befehle, welche zum Beispiel zum Erstellen von Buffern und Texturen benötigt werden, ausgeführt werden.

5.3.3 Laden der Texturen

Die Pfade der Texturen werden von der `PBRScene`-Klasse bereitgestellt, welche auch dafür sorgt, dass kein Pfad doppelt vorkommt. Da das Laden vieler Texturen ein zeitaufwendiger Prozess ist, wird das Laden der Daten von der Festplatte mit *stb_ image* durch *OpenMP* parallelisiert ausgeführt. Das Anlegen der eigentlichen Texturen auf der Grafikkarte passiert danach sequenziell. Dieser Vorgang ist in der Klasse `BaseApp` in der Funktion `createTextureImageFromLoaded(...)` wiederverwendbar abstrahiert. Diese Funktion bekommt eine `ImageLoadInfo`-Struktur übergeben, in der ein Zeiger auf die Daten des Bildes, die Abmessungen und die Anzahl der Mipmaps stehen. Die Daten werden in einen *staging buffer* geladen, einen Buffer, der in CPU-Speicher gemappt ist. So können die Daten mit `memcpy` transferiert werden. Der Buffer wird dann nur dazu benutzt, um die Daten in das danach angelegte `vk::Image` zu kopieren. So kann die Nutzung des Bildes für bessere Performance mit *GPU only*, also Grafikkarten-Exklusiv, angegeben werden. Das `vk::Image` wird mit der Funktion `createImage(...)` erstellt. Diese Funktion kreiert das Vulkan-Objekt und die dazugehörige Speicher-Allokation, welche mit der *Vulkan Memory Allocator*-Bibliothek erstellt wird, kreiert. Zurückgegeben wird eine selbst geschriebene `ImageInfo`-Struktur, welche eine leichtgewichtige Sammlung der zu diesem Bild gehörigen Informationen ist. Dies vereinfacht das Anlegen der zum `vk::Image` gehörigen restlichen Vulkan-Objekte, wie *image views* und *sampler*.

```
struct ImageInfo
{
  vk::Image m_Image = nullptr;
  VmaAllocation m_ImageAllocation = nullptr;
  VmaAllocationInfo m_ImageAllocInfo = {};
  unsigned mipLevels;
};
```

In Vulkan müssen Mipmaps manuell erstellt werden. Dies passiert mittels *blit*-Operationen in einem temporären *command buffer* und ist in der Funktion `generateMipmaps(...)` abstrahiert.

Nachdem alle Texturen geladen, die `vk::Image`-Objekte erstellt worden sind und der GPU-Speicher allokiert und gefüllt und die Mipmaps generiert wurden, können die dazugehörigen *image views* und *sampler* erstellt werden.

Ein `vk::ImageView` gehört direkt zu einem `vk::Image` dazu und gibt an, ob damit auf eine 1D-, 2D-, 3D- oder *cube*-Textur zugegriffen wird oder auf ein Array einer solchen und gibt das Format für den Zugriff auf die Textur an. Außerdem können bestimmte Komponenten angegeben werden und ein Ausschnitt gewählt werden und einzelne Mipmap-Stufen oder eine bestimmte Schicht im Falle eines Arrays ausgewählt werden. Ein `vk::ImageView` gibt also an, auf welchen Teil des `vk::Image` mit welchem Format zugegriffen wird. Im Fall der Szenentexturen in dieser Anwendung handelt es sich bei allen Texturen um 2D-Texturen im Format `vk::Format::eR8G8B8A8Unorm`, also vier Kanäle mit Werten zwischen 0 und 1, welche mit 8 Bit gespeichert sind. Es wird immer die ganze Textur benutzt und alle Mipmap-Stufen werden verwendet.

Im `vk::Sampler` wird gespeichert, welche Filtermodi für den Texturzugriff im Shader verwendet werden, wie Ränder und automatische Vergleichsoperationen, beispielsweise bei Tiefenwerttexturen, gehandhabt werden. Zusammenfassend lässt sich sagen, dass ein `vk::Sampler` regelt, wie ein `texture(...)`-Befehl im Shader auf die Textur zugreift. Alle Szenentexturen nutzen den Filtermodus `vk::Filter::eLinear` in alle Richtungen, anisotrope Filterung und eine sich wiederholende Randbehandlung.

Nachdem alle Texturen geladen und die dazugehörigen Objekte erstellt worden sind, wird der Arbeitsspeicher, in den die Daten geladen wurden, wieder freigegeben.

5.3.4 Anlegen der Tiefentexturen

Das Anlegen der Tiefentexturen welche für den Tiefentest bei der Rasterisierung benutzt werden, passiert in der `BaseApp`-Klasse. Hier wird nur ein `vk::Image` und ein `vk::ImageView` benötigt, aber kein `vk::Sampler`, da die Texturen nur für den impliziten Tiefentest benötigt werden, aber kein explizites Lesen der Tiefe in einem Shader vorgenommen wird.

5.3.5 Anlegen der Renderpasses

In der Anwendung werden zwei verschiedene Rasterisierungs-Durchläufe (*Renderpasses*) verwendet, mit einem wird in den G-Buffer geschrieben (*G-Buffer Renderpass*), mit dem zweiten das Beleuchten und Kombinieren aller zuvor berechneten Informationen vorgenommen (*full-screen lighting Renderpass*). In Vulkan werden in einem `vk::Renderpass` nur die *Attachments*, also

Ausgabetexturen, in die geschrieben wird, und die für die Synchronisation zuständigen *dependencies*, also Abhängigkeiten gespeichert. Weitere Informationen, die semantisch einem Renderpass zugeordnet werden können, wie die verwendeten Shader, Befehle oder Rasterisierungs-Einstellungen, werden erst später im Ablauf der Initialisierungsphase der Applikation erstellt. Dies ist in Abschnitt 5.3.14 mit dem Erstellen der *pipeline* und Abschnitt 5.3.19 mit dem Erstellen der *command buffer* beschrieben.

Der G-Buffer Renderpass hat folgende Attachments: Drei Vier-Kanal-Texturen für Positionen, Normalen und Texturkoordinaten, mehr dazu in Abschnitt 5.4.3, und ein Tiefen-Attachment für den Tiefentest. An dieser Stelle werden keine konkreten Texturen angegeben, damit diese austauschbar bleiben ohne den Renderpass zu verändern, sondern nur das Format, in dem geschrieben wird und die Art des Attachments, die *Layouts*, die eine Textur vor und nach Ausführung des Renderpasses haben soll.

Layouts in Vulkan geben an, in welchem Zustand sich die Ressource befindet, also ob sie beispielsweise zum daraus Lesen im Shader verwendet wird, zum Anzeigen auf dem Bildschirm, oder zum Schreiben als Attachment. Diese müssen in Vulkan manuell verwaltet werden und durch Synchronisationsobjekte wie *barriers* geändert werden. In einem Renderpass geschieht ein semi-impliziter Übergang zwischen Layouts: Implizit da keine zusätzliche Barriere verwendet werden muss, allerdings nur semi-implizit, da das Layout vorher und nachher trotzdem im Renderpass explizit angegeben werden muss.

Der Fullscreen-Lighting-Renderpass hat nur ein Farb-Attachment, welches ein Swapchain-Bild ist und ein zusätzliches Tiefenattachmen.

5.3.6 Erstellen der Framebuffer und Framebuffer-Ressourcen

Da die Renderpasses nun erstellt sind, können die Framebuffer erstellt werden. Die Swapchain-Framebuffer wird in der `BaseApp`-Klasse erstellt, benötigt allerdings den Fullscreen-Lighting-Renderpass als Eingabe, da der Renderpass, aus dem in den Framebuffer geschrieben wird, in diesem referenziert werden muss. Es wird pro Bild in der *swap chain* ein `vk::Framebuffer` erstellt.

Um die Framebuffer für den G-Buffer-Renderpass zu erstellen, müssen zuerst die Ressourcen angelegt werden, in die gerendert wird. Es wird für jedes der drei *swap chain*-Bilder auch jeweils ein G-Buffer angelegt, so wird das *triple buffering* auch mit dem G-Buffer möglich. Dies ist ein Schritt, der sich durch die ganze Anwendung zieht, es werden also alle Ressourcen, in die geschrieben wird, in dreifacher Ausführung angelegt. Der G-Buffer selbst besteht aus drei 4-Kanal-Texturen welche pro Kanal einen 32-Bit float-Wert speichern. Zu jeder Textur wird ein `vk::ImageView` und ein `vk::Sampler` erstellt. Außerdem wird eine Tiefentextur erstellt, welche allerdings wie die für die *swap chain* keinen `vk::Sampler` benötigt, da sie nie explizit gelesen wird. Mit diesen Ressourcen wird schließlich das `vk::FrameBuffer`-Objekt erstellt, ebenfalls in dreifacher Ausführung.

5.3.7 Erstellen der Buffer

In den nächsten Schritten werden einige der für diese Applikation benötigten Buffer erstellt und gefüllt. Es handelt sich dabei um den Vertexbuffer, den Indexbuffer, den Buffer, der die nötigen Informationen für die *multi draw indirect*-Technik bereitstellt, den Buffer der die Transformationsmatrizen für die Objekte in der Szene beinhaltet, und den Buffer, der die Materialien beinhaltet.

Gefüllt werden die Buffer mit *staged transfer*, also indem erst ein temporärer Buffer gefüllt wird, dessen Inhalt dann in den eigentlichen Buffer kopiert wird. Der temporäre Buffer wird mit der Funktion `createBuffer(...)` erstellt, welche den Speicher für den Buffer mit dem *Vulkan Memory Allocator* allokiert und das `vk::Buffer`-Objekt erstellt und eine `BufferInfo`-Struktur zurückgibt.

```
struct BufferInfo
{
  vk::Buffer m_Buffer = nullptr;
  VmaAllocation m_BufferAllocation = nullptr;
  VmaAllocationInfo m_BufferAllocInfo = {};
};
```

In den temporären Buffer, welcher direkt gemappt ist, werden nun die Daten mit `memcpy` geschrieben. Dann werden die Daten in den eigentlichen Buffer, welcher dann nur auf der GPU verwendet wird, mit der Funktion `copyBuffer(...)` kopiert. In dieser Funktion wird der Vulkan-Command zum Kopieren von Daten benutzt. Dies wird in der *transfer queue* ausgeführt, damit dies parallel zu anderen Vulkan-Commands ausgeführt werden kann. Danach wird der temporäre Buffer gelöscht.

Der Vertexbuffer besteht aus allen Vertices aller Objekte in der Szene, einfach hintereinandergereiht. Der Indexbuffer ist gleich aufgebaut, allerdings mit den Indizes. Der Buffer, welcher die Informationen für *multi draw indirect* beinhaltet, besteht aus einer selbst geschriebenen `PerMeshInfoPBR`-Struktur, welche die vorgegebene `vk::DrawIndexedIndirectCommand`-Struktur erweitert.

```
struct DrawIndexedIndirectCommand
{
  uint32_t indexCount;
  uint32_t instanceCount;
  uint32_t firstIndex;
  int32_t vertexOffset;
  uint32_t firstInstance;
};
```

Die Struktur `vk::DrawIndexedIndirectCommand` beinhaltet die Informationen, welche von *multi draw indirect* gebraucht werden und existiert pro Objekt in der Szene. Die Daten werden beim Laden der Objekte ermittelt und geben an, auf welche Bereiche im Vertex- und Indexbuffer zugegriffen

werden muss. Zusätzlich kann ein Objekt in mehreren Instanzen gerendert werden, davon wird in dieser Anwendung allerdings kein Gebrauch gemacht.

```
struct PerMeshInfoPBR : vk::DrawIndexedIndirectCommand
{
  int32_t texIndexBaseColor;
  int32_t texIndexMetallicRoughness;
  int32_t assimpMaterialIndex;
};
```

`PerMeshInfoPBR` ist eine Erweiterung der von Vulkan vorgegebenen Struktur `vk::DrawIndexedIndirectCommand` und beinhaltet Materialinformationen, welche ebenfalls pro Objekt benötigt werden. Diese benutzerdefinierte Erweiterung der von der API vorgegebenen Struktur wird genau zu diesem Zweck ermöglicht, indem die Größe der benutzten Struktur im *multi draw indirect*-Befehl angegeben werden kann.

Der Model-Matrix-Buffer beinhaltet die beim Laden der Szene durch eine Szenegraphtraversierung berechneten absoluten Transformationsmatrizen aller Objekte.

Der Materialbuffer enthält alle für die Objekte in der Szene benötigten Materialien. Da mehrere Objekte dasselbe Material verwenden können, sind die Materialien nicht pro Objekt gespeichert, sondern separat, und werden mittels Indizes referenziert. Hierbei ist zu beachten, dass diese Materialinformationen nur zum Einsatz kommen, wenn ein Objekt jeweilige Textur referenziert.

```
struct MaterialInfoPBR
{
  glm::vec3 baseColor;
  float roughness;
  glm::vec3 f0;
  float metalness;
};
```

5.3.8 Erstellen des Descriptor Pools

Ein `vk::DescriptorPool` wird dazu verwendet, *descriptor sets* zu allokieren. *Descriptor sets* werden dazu benutzt, um auf Ressourcen wie Buffer und Texturen in Shadern zuzugreifen. Zum Erstellen des *descriptor pool* wird die maximale Anzahl von *descriptor sets*, welche daraus allokiert werden sollen, und deren Größen, beschrieben in einem `vk::DescriptorPoolSize`-Array. Diese Anwendung benutzt einen einzelenen *descriptor pool*, aus dem alle *descriptor sets* allokiert werden. Dieser wird in der Klasse `RTCombinedApp` mit der Funktion `createCombinedDescriptorPool()` erstellt, da die Größe des Pools applikationsspezifisch ist.

5.3.9 Anlegen der Ressourcen für die Lichtquellen in der Szene

Die Lichtquellen in der Szene werden von der Klasse `PBRLightManager` verwaltet. Diese ist für die Steuerung der Lichtquellen mit der Benutzeroberfläche verantwortlich, die dazugehörigen Vulkan-Ressourcen werden in der Klasse `RTCombinedApp` gespeichert, da dort auch Zugriff auf die Vulkan-Objekte, welche zum Verändern von Buffern notwendig sind, vorhanden ist. Es werden in dieser Anwendung drei Typen von Lichtquellen unterstützt: Gerichtete Lichtquellen, Punktlichtquellen und Spot-Lichtquellen. In späteren Abschnitten werden hauptsächlich Punktlichtquellen verwendet, da diese sich in der Praxis für die in Abschnitt 6.1 beschriebenen verwendeten Szenen als am geeignetsten gezeigt haben.

Es werden die Lichtquellen-Strukturen erstellt, dem `PBRLightManager` übergeben und die Buffer erstellt und mit den Daten gefüllt. Danach wird ein *descriptor set* erstellt und mit den Informationen zu den Lichtquellen-Buffern gefüllt. Das Gruppieren der Lichtquellen in ein eigenes *descriptor set*, welches in der Anwendung sonst pro Renderpass erstellt wird, hat den Vorteil, dass man dieses an beliebigen Stellen zusätzlich verwenden kann, ohne die Buffer der Lichtquellen nochmals einzeln in das zum Renderpass gehörige *descriptor set* hinzuzufügen.

Um ein `vk::DescriptorSet` zu erstellen, muss zunächst das dazugehörige `vk::DescriptorSetLayout` erstellt werden. In diesem steht pro Ressource das *binding*, mit dem im Shader auf die Ressource zugegriffen wird, der Typ der Ressource, hierbei `vk::DescriptorType::eStorageBuffer` für einen *shader storage buffer*, die Shader-Typen, in dem auf die Ressource zugegriffen werden kann, in diesem Fall Fragment Shader, Ray Generation Shader und Closest Hit Shader. Mit dem fertigen `vk::DescriptorSetLayout` wird das `vk::DescriptorSet` aus dem `vk::DescriptorPool` allokiert. Dies muss noch mit den eigentlichen Informationen zu den Buffern gefüllt werden. Dazu wird eine `vk::DescriptorBufferInfo` erstellt, welche die Adresse des Buffers beinhaltet, um dies in eine `vk::WriteDescriptorSet`-Struktur zu schreiben, welche das *binding*, das *descriptor set* und den Ressourcen-Typ enthält. Das Ergebnis wird dann mit der Vulkan-Funktion `vk::Device::updateDescriptorSets(...)` benutzt, um das *descriptor set* zu füllen.

5.3.10 Anlegen der Texturen für Ray Tracing-Ergebnisse

Nachdem die Lichtquellen erstellt sind, können die Texturen, in die die Ergebnisse der Ray Tracing-Verfahren geschrieben werden, erstellt werden. Für die Schatten wird pro Lichtquellen-Typ ein 1-Kanal-Textur-Array mit 32-Bit-float-Werten in jedem Kanal angelegt, bei dem die Anzahl der Texturarray-Schichten der Anzahl der Lichtquellen entspricht. Für Umgebungsverdeckung wird eine einzelne 1-Kanal-Textur mit 32-Bit-float-Werten pro Kanal angelegt.

Für Reflexionen werden zwei 4-Kanal-Texturen mit 32-Bit-float-Werten pro Kanal angelegt, eine davon in voller Auflösung und die andere mit der halben Kantenlänge, also einem Viertel der Pixel. Anzumerken ist hier, dass die beschriebenen Ressourcen wegen des *triple buffering* in dreifacher Ausführung angelegt werden. Alle Ressourcen werden mit den Verwendungs-Einstellungen für das Beschreiben als *storage image* als auch für das Lesen als Textur mit *sampling* angelegt.

Danach werden die Ressourcen mit einer `vk::ImageMemoryBarrier` in das Layout `vk::ImageLayout::eShaderReadOnlyOptimal` für die Pipeline-Stage `vk::PipelineStageFlagBits::eFragmentShader` gebracht, da dies der Zustand der Ressource ist, die später erwartet wird.

5.3.11 Erstellen der Descriptor Sets für Ray Tracing-Ressourcen

Für den Zugriff auf die zuletzt erstellten Ray Tracing-Texturen werden drei *descriptor sets* jeweils in dreifacher Ausführung erstellt. So kann das *descriptor set* bei jedem Frame für das *triple buffering* gewechselt werden. Dies hat den Vorteil, dass beim *triple buffering* nicht die Texturen, welche in den *descriptor sets* referenziert werden, ausgetauscht werden müssen wobei das *descriptor set* für jeden Frame verändert werden müsste. So wird zusätzlicher Aufwand auf der CPU-Seite während des eigentlichen Renderings vermieden, da keine *descriptor sets* verändert werden müssen, sondern lediglich unterschiedliche *descriptor sets* in den unterschiedlichen *frames* benutzt werden müssen.

Das erste der drei für die Ray Tracing-Ressourcen angelegten *descriptor sets* beinhaltet alle Ray Tracing-Ergebnistexturen beziehungsweise die Arrays als `vk::DescriptorType::eCombinedImageSampler`, damit diese als Textur mit *sampling* gelesen werden können und kommt beim Zusammensetzen der Texturen zum fertigen Bild zum Einsatz. Die anderen beiden *descriptor sets* beinhalten die Texturen für Schatten beziehungsweise die Textur für die Umgebungsverdeckung als `vk::DescriptorType::eStorageImage` um beschrieben zu werden. Das *descriptor set* mit den Ressourcen für die Reflexionen wird wegen des größeren Umfangs an benötigten Daten zu einem einem späteren Zeitpunkt, beschrieben in Abschnitt 5.3.17, erstellt.

5.3.12 Erstellen der Descriptor Sets für das Rendern in den G-Buffer

In der Funktion `createGBufferDescriptors()` wird das *descriptor set* für den G-Buffer-Renderpass gefüllt. Dieses liegt nicht in dreifacher Ausfürhung vor, da die Ausgaben des Renderpasses vom Framebuffer-Objekt verwaltet werden. So wird statt dem Wechseln des *descriptor sets* das *framebuffer object* gewechselt. Die Shader im G-Buffer-Renderpass benötigen Zugriff auf den Model-Matrix-Buffer, den Buffer mit den Informationen für *multi draw indirect*, da dort wie in Abschnitt 5.3.7 beschrieben auch enthalten ist,

welche Textur zu welchem Objekt benötigt wird. Der G-Buffer-Shader führt, wie in Abschnitt 5.4 beschrieben, zwar keine Texturierung durch, speichert aber die Mipmap-Stufe, welche zum texturieren später benötigt wird. Deshalb beinhaltet dieses *descriptor set* auch die Referenz auf alle Texturen. Dafür kommt ein Array von Texturen zum Einsatz. Dies kann im Gegensatz zu einer Array-Textur aus beliebig vielen, verschieden großen Texturen bestehen.

5.3.13 Erstellen der Descriptor Sets für die Beleuchtung

Der *full screen lighting*-Renderpass, welcher die direkte Beleuchtung anhand der Informationen aus dem G-Buffer berechnet und sowohl damit als auch mit den Ray Tracing-Ergebnissen das fertige Bild produziert, braucht den Buffer mit den *multi draw indirect*-Informationen wegen der Materialien, den Zugriff auf alle G-Buffer-Texturen und den Material-Buffer sowie alle Szenen-Texturen in dem ihm zugeordneten *descriptor set*. Der Zugriff auf die Ray Tracing-Ergebnistexturen und Lichtquellen ist über die jeweiligen separaten *descriptor sets* geregelt. Da die G-Buffer-Texturen in dreifacher Ausführung für das *triple buffering* vorhanden sind, muss das *descriptor set* ebenfalls in dreifacher Ausführung erstellt werden.

5.3.14 Erstellen der Rasterisierungs-Pipelines

Eine `vk::Pipeline` speichert den gesamten Rendering-Zustand für einen Renderpass. Dazu gehören die verwendeten Shader, die Informationen zum Vertex-Buffer, wie die Topologie der Primitive, der Viewport, Einstellungen zu Culling, Multisampling, Tiefentest, und Blending. Durch das Festlegen dieser Werte im Vorhinein werden Performance-Einbußen durch das spontane Ändern dieses Rendering-Zustands verhindert und zusätzlicher Aufwand auf der CPU-Seite des Renderings gespart.

Die Einstellungen für die Pipeline für das Rendering in den G-Buffer sind wie folgt: Vertices sind als `vk::PrimitiveTopology::eTriangleList` gegeben vor, Backface-Culling ist aktiviert und Front Faces sind gegen den Uhrzeigersinn definiert. Der Tiefentest ist aktiviert, Blending wird nicht vorgenommen. Erwähnenswert ist, dass *specialization constants* zum Einsatz kommen. Diese erlauben es, eine Variable in einem Shaderprogramm zur Kompilierzeit des Shaders von der SPIR-V-Zwischenrepräsentation zum Maschinencode mit einem konstanten Wert zu belegen. Dies wird benutzt, um die Länge des Arrays mit den Szenentexturen festzulegen. Diese muss konstant sein und so lässt sich ein Festlegen auf eine konkrete Anzahl, was dann beim Wechseln des geladenen 3D-Modells manuell geändert werden müsste, verhindern.

Ebenfalls angegeben werden beim Erstellen die Informationen über die verwendeten *push constants*. Diese sind sind kleine Mengen von Daten, welche ohne Buffer an den Shader übergeben werden können. In dieser Anwendung

werden sie dazu benutzt, Kameramatrizen zu übergeben. Außerdem müssen die Layouts der *descriptor sets*, welche mit der Pipeline benutzt werden können, referenziert werden.

Die Einstellungen für die Pipeline des Renderpasses, welcher die finale Beleuchtungsberechnung durchführt, unterscheidet sich von der anderen Pipeline nur durch die anderen Shader und durch das Referenzieren von mehreren *descriptor sets*.

5.3.15 Anlegen der Zufallszahl-Texturen

Da für das Generieren der Strahlen in den Ray Tracing-Shadern Zufallszahlen verwendet werden, müssen Texturen mit initialen Zufallszahlen gefüllt werden. Das sind in diesem Fall drei Bilder mit 4 Kanälen und 32-Bit-Integern ohne Vorzeichen pro Kanal.

Die eigentliche Berechnung der verwendeten Zufallszahlen geschieht wie in Abschnitt 5.5 beschrieben auf der GPU mit einer Kombination eines Tausworthe-Step mit einem linearen Kongruenzgenerator, wie von Howes und Thomas [HT07] beschrieben.

Generiert werden die Initialisierungsdaten mit dem in der C++-Standardbibliothek gegebenen `std::mt19937`-Zufallszahlengenerator, parallelisiert mit dem Ausführungsmodus `std::execution::par` in `std::for_each`.

Zusätzlich wird hier noch ein Buffer erstellt, welcher pro Frame veränderliche Daten für die Ray Tracing-Shader bereitstellt: Die Kameraposition, die Nummer des aktuellen Frames und Einstellungen wie die Anzahl der für das Ray Tracing benutzten Samples. Mehr dazu ist in Abschnitt 5.3.17 beschrieben.

5.3.16 Erstellen der Beschleunigungsdatenstruktur

Das Erstellen der Beschleunigungsdatenstruktur ist der erste Punkt in der Anwendung, an der die `VK_NV_ray_tracing`-Extension zum Einsatz kommt. Zuerst wird ein *offset buffer* erstellt, welcher nicht durch die Extension vorausgesetzt wird, später aber für den Zugriff in den Vertexbuffer bei einem gefundenen Schnittpunkt notwendig wird.

Das Bereitstellen der Daten für das Erstellen der Beschleunigungsdatenstruktur geschieht in mehreren Schritten, die in den folgenden Abschnitten erklärt werden. Der grundlegende Aufbau der zweistufigen Beschleunigungsdatenstruktur ist in Abbildung 3 abgebildet.

Anlegen der Geometrie-Objekte Für jedes Objekt in der Szene muss eine `vk::GeometryTrianglesNV`-Struktur gefüllt werden. In dieser steht, in welchem Vertexbuffer und an welcher Stelle darin die Vertex-Daten stehen, das Vertex-Format, in welchem Indexbuffer die Indizes stehen und der Index-Typ. Außerdem kann eine Transformation angegeben werden, was

in dieser Anwendung nicht verwendet wird. Die Transformation wird später vorgenommen. Die `vk::GeometryTrianglesNV`-Struktur wird in eine `vk::GeometryDataNV`-Struktur geschrieben. Diese ist dafür da, dass entweder Dreiecke, so wie in diesem Fall, oder *axis aligned bounding boxes* benutzt werden können, da beides darin gespeichert werden kann. Die `vk::GeometryDataNV`-Struktur wird in eine `vk::GeometryNV`-Struktur geschrieben, welche speichert, dass es sich um Dreiecke handelt, und zusätzliche Informationen dazu speichern kann.

Anlegen der Bottom-Level Acceleration Structure Für jedes Objekt in der Szene wird nun ein `vk::AccelerationStructureNV`-Objekt angelegt und der dazugehörige Speicher allokiert. Diese stellen eine lokale Beschleunigungsdatenstruktur für das jeweilige Objekt dar, welche später transformiert, mehrfach instantiiert, neu aufgebaut und erneuert werden kann. Um das Erstellen zu abstrahieren, kommt eine Lambda-Funktion zum Einsatz. In dieser wird zuerst eine `vk::AccelerationStructureInfoNV`-Struktur angelegt. Darin steht, dass es sich in diesem Fall um eine *bottom level*-Beschleunigungsdatenstruktur, also eine lokale Beschleunigungsdatenstruktur handelt. Die Anzahl der zuvor beschriebenen Geometrie-Objekte und eine Einstellung zum Aufbau der Struktur wird übergeben. Im Fall einer *top level*-Beschleunigungsdatenstruktur kann auch noch die Anzahl der Instanzen angegeben werden. Einstellungen für den Aufbau der Struktur sind die folgenden:

```
typedef enum VkBuildAccelerationStructureFlagBitsNV {
  VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_UPDATE_BIT_NV = ↩
      0x00000001,
  VK_BUILD_ACCELERATION_STRUCTURE_ALLOW_COMPACTION_BIT_NV = ↩
      0x00000002,
  VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_TRACE_BIT_NV = ↩
      0x00000004,
  VK_BUILD_ACCELERATION_STRUCTURE_PREFER_FAST_BUILD_BIT_NV = ↩
      0x00000008,
  VK_BUILD_ACCELERATION_STRUCTURE_LOW_MEMORY_BIT_NV = 0x00000010,
  VK_BUILD_ACCELERATION_STRUCTURE_FLAG_BITS_MAX_ENUM_NV = ↩
      0x7FFFFFFF
} VkBuildAccelerationStructureFlagBitsNV;
```

Für die *bottom level*-Beschleunigungsdatenstrukturen wird nur das *prefer fast trace*-Bit gesetzt, also der Aufbau auf eine Art und Weise, welche das schnelle Traversieren der Beschleunigungsdatenstruktur für Schnitttests gegenüber dem schnellen Aufbau priorisiert. Die lokalen Objekt-Datenstrukturen müssen nicht veränderbar sein, da das nur für in sich verformbare Objekte relevant ist, welche in dieser Anwendung nicht zum Einsatz kommen.

Die `vk::AccelerationStructureInfoNV`-Struktur wird in eine vom Typ `vk::AccelerationStructureCreateInfoNV` geschrieben, welche dann benutzt wird, um das `vk::AccelerationStructureNV` Objekt zu

erstellen. Danach muss der benötigte GPU-Speicher allokiert werden. Dieser Vorgang verlangt, dass die Ansprüche, die an den allokierten Speicher gestellt werden, vom Treiber abgefragt werden. Ebenfalls muss überprüft werden, dass die gewünschte Art der Allokierung, nämlich möglichst effizienter GPU-Speicher, bei Vulkan *device local* genannt, in den vom Treiber gewünschten Speicheranforderungen enthalten ist. Ist dies der Fall, kann der Speicher mit dem *Vulkan Memory Allocator* allokiert werden. Danach wird der allokierte Speicher mit einer `vk::BindAccelerationStructureMemoryInfoNV`-Struktur an das `vk::AccelerationStructureNV`-Objekt gebunden. Um die zusammengehörigen Informationen zu gruppieren, benutzt die Anwendung ein wie bei `vk::Image` und `vk::Buffer` eine eigene Struktur:

```
struct ASInfo
{
  vk::AccelerationStructureNV m_AS = nullptr;
  VmaAllocation m_BufferAllocation = nullptr;
  VmaAllocationInfo m_BufferAllocInfo = {};
};
```

Erstellen der Geometrie-Instanzen Um die lokalen, objektbezogenen *bottom level*-Beschleunigungsdatenstrukturen in der übergeordneten *top level*-Beschleunigungsdatenstruktur zu referenzieren, wird ein Buffer, gefüllt mit `GeometryInstance`-Strukturen benötigt.

```
struct GeometryInstance
{
  // row major 4x3 model matrix
  float transform[12];

  // instanceId is exposed as gl_InstanceCustomIndexNV
  uint32_t instanceId : 24;

  // mask to exclude hitting this geometry.
  // if rayMask & instance.mask == 0, the geometry will NOT be hit
  uint32_t mask : 8;

  // instance offset is the hit shader index
  uint32_t instanceOffset : 24;

  // any of VkGeometryInstanceFlagBitsNV
  uint32_t flags : 8;

  // bottom level AS handle this instance corresponds to
  uint64_t accelerationStructureHandle;
};
```

In der Anwendung wird von jeder *bottom level*-Beschleunigungsdatenstruktur nur eine Instanz erstellt. Die `transform`-Matrix ist die Model-Matrix des Objekts. Bei den `flags` wird von der Anwendung das Culling expli-

zit ausgeschaltet, was für eine bessere Performance empfohlen wird[25]. Der `instanceOffset` erlaubt es, für verschiedene Objekte und deren Instanzen unterschiedliche Shader bei einem Schnittpunkt auszuführen, was von der Anwendung wegen des allgemein gehaltenen Materialsystems nicht benötigt wird.

Anlegen der Top-Level Acceleration Structure Das Vulkan-Objekt für die *top level*-Beschleunigungsdatenstruktur wird wie die untergeordneten Beschleunigungsdatenstrukturen ebenfalls mit der Lambda-Funktion erstellt. Hierbei wird zusätzlich zum *prefer fast trace*-Bit das *allow update*-Bit gesetzt, um das Bewegen der Objekte in der Szene zu ermöglichen.

Aufbauen der Beschleunigungsdatenstruktur Bis zu diesem Punkt sind die Objekte für die Beschleunigungsdatenstrukturen angelegt und der benötigte Speicher ist reserviert, das eigentliche Aufbauen der Beschleunigungsdatenstruktur muss allerdings noch manuell angestoßen werden. Dafür wird zuerst ein *scratch buffer* benötigt, welcher Speicherplatz bereitstellt, der für den Aufbau der Beschleunigungsdatenstruktur benötigt wird. Dieser wird nur während des Aufbaus und später während einer Erneuerung der Beschleunigungsdatenstruktur gebraucht. Das benötigte Minimum an Speicherplatz für jede Beschleunigungsdatenstruktur wird mit der Vulkan-Funktion `getAccelerationStructureMemoryRequirementsNV()` vom Treiber abgefragt. Da ein *scratch buffer* reicht, wird der benötigte Speicherplatz aller Beschleunigungsdatenstrukturen abgefragt und der Maximalwert verwendet, um den *scratch buffer* zu erstellen.

Da das eigentliche Aufbauen der Beschleunigungsdatenstruktur auf der GPU erledigt wird, kommt ein temporärer *command buffer* zum Einsatz, in dem die `buildAccelerationStructureNV`-Commands geschrieben werden. In diesem wird bei den *bottom level*-Strukturen nur der *scratch buffer* benötigt, bei der *top level*-Struktur wird auch der *instance buffer* gebraucht. Die Beschleunigungsdatenstrukturen werden nach dem Aufbau mit einer `vk::MemoryBarrier` in den Zustand gebracht, in dem sie in den Ray Tracing Shadern benutzt werden können, also mit dem Zugriffstyp `vk::AccessFlagBits::eAccelerationStructureReadNV` und `vk::PipelineStageFlagBits::eRayTracingShaderNV` als Shader-Typ.

Die Dauer des Aufbauens der Beschleunigungsdatenstrukturen wird mit dem in Abschnitt 5.3.22 beschriebenen Timer gemessen und in der Konsole ausgegeben.

[25] `https://devblogs.nvidia.com/rtx-best-practices/`, letzter Abruf: 5. Juni 2019

5.3.17 Erstellen der Ray Tracing-Pipelines

Die Ray Tracing-Pipelines werden in Vulkan trotz der nicht genutzten Rasterisierungs-Funktionalität ebenfalls als `vk::Pipeline`-Objekt angelegt. Für das Erstellen wird statt einer `vk::PipelineCreateInfo`- eine `vk::RayTracingPipelineCreateInfoNV`-Struktur verwendet. In dieser werden die *shader stages* und *shader groups* angegeben, zusammen mit der maximalen Rekursionstiefe für das Ray Tracing. Die *shader stages* repräsentieren alle konkreten Shader in der Pipeline und beinhalten deren Programmcode in der SPIR-V-Zwischenrepräsentation. Die *shader groups* hingegen gruppieren die Shader für das Ray Tracing in sogenannte *hit groups* und indexieren jeden Shader innerhalb der Gruppe. Zusätzlich zur Pipeline wird eine *shader binding table* benötigt, welche man sich als Buffer, der die Shader enthält, vorstellen kann.

Soft Shadows Pipeline Für die Ray Tracing-Schatten wird erst das *descriptor set* angelegt, welches die Beschleunigungsdatenstruktur, die Positions-Text des G-Buffers, die Zufallszahlentextur und den Buffer mit den pro-Frame veränderbaren Informationen für das Ray Tracing referenziert.

Dann werden die Shader geladen. Im Fall der Schatten-Pipeline werden ein Ray Generation Shader und ein Miss Shader benötigt. Ein Closest Hit Shader ist nicht notwendig, genau wie ein Any Hit Shader.

```
const auto rgenShaderCode =
  Utility::readFile("combined/softshadowPBR.rgen" +
  m_shaderExtension);
const auto rgenShaderModule =
  m_context.createShaderModule(rgenShaderCode);

const auto missShaderCode =
  Utility::readFile("combined/softshadow.rmiss" +
  m_shaderExtension);
const auto missShaderModule =
  m_context.createShaderModule(missShaderCode);
```

Die *shader stages*, welche angeben, welcher geladene Shader-Code welche Art von Ray Tracing-Shader beinhaltet, sehen dann wie folgt aus:

```
std::array rtShaderStageInfos = {
  vk::PipelineShaderStageCreateInfo({},
    vk::ShaderStageFlagBits::eRaygenNV,
    rgenShaderModule, "main"),
  vk::PipelineShaderStageCreateInfo({},
    vk::ShaderStageFlagBits::eMissNV,
    missShaderModule, "main")
};
```

Die *shader groups*, welche benutzt werden, um die Shader nachher zu indexieren und damit zu kontrollieren, bei welchem Strahl welcher Shader aufgerufen wird, haben das folgende Format:

```
std::array shaderGroups = {
  // group 0: raygen
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eGeneral,
    0, VK_SHADER_UNUSED_NV,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV},
  // group 1: miss
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eGeneral,
    1, VK_SHADER_UNUSED_NV,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV}
};
```

Sowohl der Ray Generation Shader als auch der Miss Shader zählen hier als `vk::RayTracingShaderGroupTypeNV::eGeneral` und werden mit 0 beziehungseise 1 indexiert.

Die `vk::Pipeline` wird dann mit diesen *shader stages* und *shader groups* erstellt und bekommt zusätzlich die Information, dass die maximale Rekursionstiefe in diesem Fall 1 beträgt.

Zuletzt wird die zur Pipeline gehörige *shader binding table* erstellt. Dies ist ein Buffer, in dem die sogenannten *shader group handles* stehen und wird benutzt, um auf der GPU auf beliebige Ray Tracing-Shader zuzugreifen. Dies stellt einen klaren Unterschied zur klassischen Rendering-Pipeline dar, in der *shader stages* zwar teilweise weggelassen werden können, jedoch keine Shader-Typen doppelt vorkommen können oder dynamisch der aufgerufene Shader ausgesucht werden kann. Bei den Ray Tracing-Shadern ist dies hingegen der Fall und wird mit der *shader binding table* umgesetzt, welche später im Ray Tracing-Befehl referenziert wird. Im folgenden Quellcode-Ausschnitt wird erst die benötigte Buffer-Größe berechnet, dann werden die *shader group handles* abgerufen und in den gemappten Speicher des Buffers geschrieben. Das Erstellen des Buffers und das Mapping des Speichers ist für eine bessere Übersicht ausgelassen.

```
const uint32_t shaderBindingTableSize =
  m_context.getRaytracingProperties().shaderGroupHandleSize *
  rayPipelineInfo.groupCount;
...

const auto res =
  m_context.getDevice().getRayTracingShaderGroupHandlesNV(
    m_rtSoftShadowsPipeline, 0, rayPipelineInfo.groupCount,
    shaderBindingTableSize, mappedData);
```

Ray Traced Ambient Occlusion Pipeline Die Pipeline für das Berechnen der Umgebungsverdeckung mit Ray Tracing (*ray traced ambient occlusion pipeline* unterscheidet sich von der für die Schatten an mehreren Stellen. Im zugehörigen *descriptor set* wird zusätzlich die Oberflächennormalen-Textur des G-Buffers benötigt und es wird zusätzlich ein Closest Hit Shader verwendet.

```
const auto rgenShaderCode =
  Utility::readFile("combined/rtao.rgen" + m_shaderExtension);
const auto rgenShaderModule =
  m_context.createShaderModule(rgenShaderCode);

const auto chitShaderCode =
  Utility::readFile("combined/rtao.rchit" + m_shaderExtension);
const auto chitShaderModule =
  m_context.createShaderModule(chitShaderCode);

const auto missShaderCode =
  Utility::readFile("combined/rtao.rmiss" + m_shaderExtension);
const auto missShaderModule =
  m_context.createShaderModule(missShaderCode);
```

Die *shader stages* sind demnach wie folgt:

```
std::array shaderGroups = {
  // group 0: raygen
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eGeneral,
    0, VK_SHADER_UNUSED_NV,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV},
  // group 1: closest hit
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eTrianglesHitGroup,
    VK_SHADER_UNUSED_NV, 1,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV},
  // group 2: miss
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eGeneral,
    2, VK_SHADER_UNUSED_NV,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV}
};
```

Dabei ist der Ray Generation Shader an Index 0, der Closest Hit Shader an Index 1 und der Miss Shader an Index 2. Der Closest Hit Shader ist in der Gruppe `vk::RayTracingShaderGroupTypeNV::eTrianglesHitGroup`, Ray Generation Shader und Miss Shader sind in der generellen Gruppe. Das Füllen der *shader binding table* passiert analog zu der für die Schatten, nur mit den hier vorgestellten Shadern aus dieser Pipeline.

Ray Traced Reflections Pipeline Die Pipeline für die Reflexionen (*ray traced reflections pipeline*) ist die Umfangreichste der drei Ray Tracing-Pipelines in der Anwendung. Im *descriptor set* werden der gesamte G-Buffer, der Vertex-, Index- und Offsetbuffer, der Material-Buffer, der Buffer mit den pro-Objekt-Informationen für *multi draw indirect* und alle Szenen-Texturen zusätzlich referenziert.

Bei den Shadern werden ein Ray Generation Shader, ein Closest Hit Shader und ein Miss Shader für die Reflexionen selbst gebraucht, zusammen mit einem zusätzlichen Miss Shader, welcher analog zu den Schatten dazu

verwendet wird, die Verschattung in den Reflexionen zu ermitteln. Dies kann, wie bei den Schatten schon beschrieben, auch mit einem zusätzlichen Closest Hit Shader oder Any Hit Shader geschehen, ist aber nicht nötig und wird deshalb aus Optimierungsgründen weggelassen.

```
const auto rgenShaderCode =
  Utility::readFile("combined/rtreflectionsPBR.rgen"
  + m_shaderExtension);
const auto rgenShaderModule =
  m_context.createShaderModule(rgenShaderCode);

const auto chitShaderCode =
  Utility::readFile("combined/rtreflectionsPBR.rchit"
  + m_shaderExtension);
const auto chitShaderModule =
  m_context.createShaderModule(chitShaderCode);

const auto missShaderCode =
  Utility::readFile("combined/rtreflections.rmiss"
  + m_shaderExtension);
const auto missShaderModule =
  m_context.createShaderModule(missShaderCode);

const auto missSecondaryShaderCode =
  Utility::readFile("combined/rtreflectionsSecondaryShadow.rmiss"
  + m_shaderExtension);
const auto missSecondaryShaderModule =
  m_context.createShaderModule(missSecondaryShaderCode);
```

Entsprechend muss die maximale Rekursionstiefe hier auf zwei festgelegt werden. Die *shader groups* referenzieren die Shader wie folgt:

```
std::array shaderGroups = {
  // group 0: raygen
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eGeneral,
    0, VK_SHADER_UNUSED_NV,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV},
  // group 1: closest hit (for reflections)
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eTrianglesHitGroup,
    VK_SHADER_UNUSED_NV, 1,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV},
  // group 3: miss (for reflection rays)
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eGeneral,
    2, VK_SHADER_UNUSED_NV,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV},
  // group 4: miss (for secondary rays: shadows in reflection)
  vk::RayTracingShaderGroupCreateInfoNV{
    vk::RayTracingShaderGroupTypeNV::eGeneral,
    3, VK_SHADER_UNUSED_NV,
    VK_SHADER_UNUSED_NV, VK_SHADER_UNUSED_NV}
};
```

An Index 0 ist der Ray Generation Shader, an Index 1 der Closest Hit Shader, an Index 2 der Miss Shader für die Reflexionen und an Index 3 der Miss Shader für die Schatten in den Reflexionen. Auch hier werden diese Daten analog zu den beiden anderen Pipelines in eine eigene *shader binding table* geschrieben.

5.3.18 Initialisieren der Kamera

Die `Camera`-Klasse, welche die Benutzereingaben verarbeitet und die View-Matrix erstellt und die Projektionsmatrix werden initialisiert. Für die Matrizen wird `glm` benutzt, für die Nutzereingaben wird zusätzlich `glfw` verwendet.

5.3.19 Erstellen der statischen Command Buffer

In Vulkan werden *commands*, also Befehle, die auf der Grafikkarte ausgeführt werden, in *command buffern* gesammelt und dann gebündelt eingereicht (*submit*). Dies hat den Vorteil, dass man Befehle, die immer wieder ausgeführt werden, nicht jedes mal neu auf der CPU-Seite durchlaufen werden müssen, sondern der fertig konstruierte *command buffer* übergeben kann. Da wegen der Übergabe von Benutzereingaben und anderen Daten, welche sich pro Frame verändern, nicht alle *command buffer* im Vorhinein gefüllt werden können, benutzt diese Anwendung ein System aus *primary*, also primären, und *secondary*, also sekundären *command buffer*. Ein primärer *command buffer* kann sekundäre *command buffer* mit dem `executeCommands(...)`-*command* ausführen.

In der Funktion `createAllCommandBuffers()` werden alle für die Applikation verwendeten *command buffer* in dreifacher Ausführung allokiert. Der genutzte primäre *command buffer* wird in `m_commandbuffers` gespeichert, allerdings noch nicht mit *commands* gefüllt, da dieser jeden Frame neu aufgenommen werden muss. Allokiert und mit *commands* gefüllt werden die jeweiligen *secondary command buffer* für das Rendering in den G-Buffer, die mit Ray Tracing berechneten Schatten, Umgebungsverdeckung und Reflexionen. Für die Reflexionen werden zwei unterschiedliche *command buffer* (beide auch in dreifacher Ausführung) aufgenommen, jeweils einer für die Reflexionen in voller und halber Auflösung. So kann nahtlos mit der Bedienoberfläche dazwischen gewechselt werden. Zusätzlich wird noch ein weiterer *secondary command buffer* allokiert, welcher benötigt wird, da *push constants* nur innerhalb eines Renderpasses übergeben werden können und das somit nicht in dem *primary command buffer* passieren kann.

Die Inhalte der *command buffer* werden in Abschnitt 5.4.3 beschrieben, da dort der Ablauf des Renderings zusammenhängend beschrieben werden kann.

5.3.20 Erstellen der Synchronisierungsobjekte

Bevor das eigentliche Rendering beginnen kann, müssen Vulkan-Synchronisationsobjekte erstellt werden. In dieser Anwendung kommen einerseits Semaphore zum Einsatz, welche in Vulkan für die Synchronisierung zwischen GPU-Vorgängen, die zu verschiedenen Zeitpunkten eingereicht worden sind, übernehmen. Für die Synchronisierung der Präsentation auf dem Bildschirm werden *Fences* benutzt, welche CPU-GPU-Synchronisierung verantworten. Der genaue Ablauf der Synchronisierung wird in Abschnitt 5.4 beschrieben.

5.3.21 Initialisieren der Benutzeroberflächen-Bibliothek

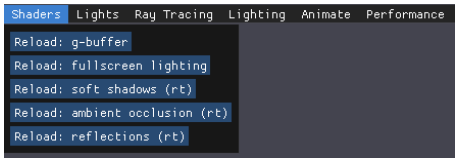
Für die Initialisierung der Benutzeroberflächen-Bibliothek *Dear Imgui* muss in einem temporären *command buffer* die vorgegebene Funktion zum Erstellen der Textur mit den Schriftarten ausgeführt werden und ein *command buffer* für den späteren Gebrauch muss allokiert werden. Alles andere zur Anzeige der Benutzeroberfläche passiert während des Renderings und wird in Abschnitt 5.4 beschrieben.

5.3.22 Erstellen der Timer für Timer Queries

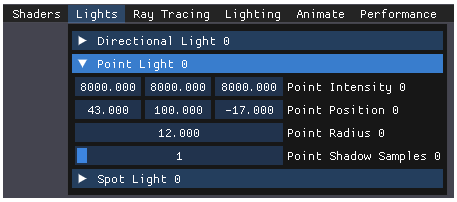
Vulkan ermöglicht es, GPU-basierte Zeitstempel innerhalb von *command buffer* zu schreiben und diese aus einem *query pool* auszulesen. Die Anwendung beinhaltet die Klasse `TimerManager`, welche `Timer`-Objekte verwaltet. Diese beinhalten die Funktionalität, Zeitstempel zu schreiben, insbesondere einen Start- und einen Stopp-Zeitstempel, deren Zeitdifferenz dann gespeichert wird und mit dem Namen des Timers in der Benutzeroberfläche angezeigt und als Graph dargestellt wird. Die Verwaltung dieser Timer im `TimerManager` vereinfacht die Handhabung mehrerer Timer, da die Zeitstempel aller Timer damit gesammelt von der GPU abgefragt werden können, nur ein Funktionsaufruf zur Anzeige aller benutzter Timer benötigt wird und Timer jederzeit entfernt, hinzugefügt, deaktiviert und aktiviert werden können.

5.4 Die Rendering-Phase

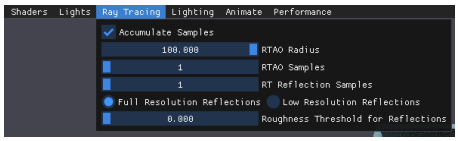
Nach dem Abschluss der Initialisierungsphase wird die Funktion `mainLoop` der Klasse `RTCombinedApp` aufgerufen, welche bis das Fenster vom Benutzer geschlossen wird, den folgenden makroskopischen Ablauf in dauerhaften Schleife durchläuft: Zuerst werden mit `glfwPollEvents()` Benutzereingaben abgefragt. Dann wird die Funktion `configureImgui()` aufgerufen, welche den aktuellen Zustand der Menüs, Buttons und Anzeigen der Benutzeroberfläche berechnet. Danach wird die Funktion `drawFrame()` der Oberklasse `BaseApp` aufgerufen. In dieser Funktion wird die Präsentation des Frames auf dem Bildschirm inklusive der Synchronisation des Präsentiervorgangs verwaltet und die virtuelle Funktion `recordPerFrameCommandBuffers()`



(a) Das *Shaders*-Menü



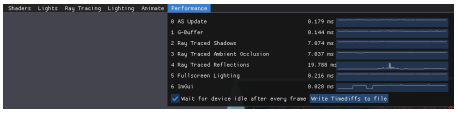
(b) Das *Lights*-Menü



(c) Das *Ray Tracing*-Menü

(d) Das *Lighting*-Menü

(e) Das *Animate*-Menü



(f) Das *Performance*-Menü

Abbildung 8: Bilder der einzelnen Menüs in der Menüleiste.

aufgerufen, deren Implementierung in der jeweiligen Unterklasse, in diesem Fall `RTCombinedApp`, dafür sorgt, die *command buffer* zu füllen, damit diese in `drawFrame()` eingereicht werden können. Zuletzt werden noch die Zeitstempel aller in Abschnitt 5.3.22 beschriebenen Timer abgerufen und falls durch die Benutzeroberfläche eingegeben, sichergestellt, dass die Anwendung wartet bis der vorherige Frame fertig ist, bevor der nächste behandelt wird. Dies ist eine Ausnahme und widerspricht der eigentlichen Synchronisation mit dem *triple buffering* und wird nur für Debugging-Zwecke verwendet.

5.4.1 Aufbauen der Inhalte der Benutzeroberfläche

In der Funktion `configureImgui()` wird der aktuelle Zustand der Benutzeroberfläche je nach Benutzereingabe definiert. Die Benutzeroberfläche besteht aus einer Menüleiste mit mehreren ausklappbaren Menüs, alle gezeigt in Abbildung 8.

Das *Shaders*-Menü erlaubt es, Shader während das Programm läuft neu zu laden. Dazu wird das jeweilige `vk::Pipeline`-Objekt und das dazugehörige *pipeline layout* zerstört und mit der jeweiligen Funktion neu erstellt, wobei die Shader neu geladen werden. Dies ist als Debugging-Funktionalität gedacht.

Das Menü für die Lichtquellen wird von der `LightManager`-Klasse verwaltet und wird benutzt, um die Parameter der Lichtquellen, wie deren Position oder Farbe, aber auch die für die Schattenberechnung verwendete Anzahl der Samples pro Frame zu ändern.

Über das *Ray Tracing*-Menü können Einstellungen für das Ray Tracing vorgenommen werden, welche mit dem Buffer, in dem pro-Frame-veränderliche

Daten für das Ray Tracing stehen, an die Shader übergeben werden. Dazu gehören die Einstellung, ob Samples akkumuliert werden, der Radius und die Anzahl der Samples für die Berechnung der Umgebungsverdeckung, die Einstellung ob Reflexionen in voller oder halber Auflösung berechnet werden und ein Rauheitsschwellwert für die Reflexionen. Mehr zur Bedeutung dieser Parameter ist in Abschnitt 5.4.3 beschrieben.

Ein weiteres Menü ermöglicht die Einstellung der *exposure*, also Belichtung und das *Animate*-Menü erlaubt die Auswahl von Szenenobjekten, welche bewegt werden sollen. Mehr zur Animation ist in Abschnitt 5.4.3 nachzulesen.

Das *Performance*-Menü zeigt die aktuellen Differenzen der Zeitstempel der `Timer` mit deren Namen, zeigt also an, wie lang jeder einzelne Rendering-Vorgang dauert. Die Gesamtdauer pro Frame wird in der Menüleiste angezeigt.

5.4.2 Präsentieren der Frames mit Synchronisation

Als nächstes wird die Funktion `drawFrame()` der Oberklasse `BaseApp` aufgerufen. In dieser wird zuerst gewartet, bis der `vk::Fence`, also das GPU-CPU-Synchronisationsobjekt, welches für denselben Frame innerhalb des letzten *triple buffering*-Durchgangs zuständig war, im signalisierten Zustand ist. Das *triple buffering* erlaubt zwar, mehrere Frames hintereinander für die Verarbeitung auf der GPU einzureichen, allerdings nicht unbegrenzt. Deshalb hat jeder Frame einen Index, welcher wiederholt wird. Beim *triple buffering* werden folglich die Indizes in der Form $0-1-2-0-1-2-\ldots$ wiederholt. Hat der aktuelle Frame also beispielsweise den Index 1, muss gewartet werden bis der letzte Frame mit dem Index 1 abgeschlossen ist. Dafür werden die `vk::Fence`-Objekte benutzt.

Um auf das jeweils nächste Bild der *swap chain* für die Präsentation zuzugreifen, muss dessen Index abgefragt werden. Die dazu anzugebende maximale Dauer (*timeout*) wird auf den maximal möglichen Wert gesetzt.

```
auto nextImageResult = m_context.getDevice().acquireNextImageKHR(
    m_context.getSwapChain(),
    std::numeric_limits<uint64_t>::max(),
    m_imageAvailableSemaphores.at(m_currentFrame),
    nullptr);
```

Dieser Befehl signalisiert bei erfolgreichem Wechseln des zu präsentierenden Bildes den als `m_imageAvailableSemaphores.at(m_currentFrame)` gespeicherten Semaphor. Danach wird abgefragt, ob die *swap chain* veraltet (*out of date*) ist, was zum Beispiel bei einer Änderung der Fenstergröße der Fall ist.

Ist das geschehen, wird die virtuelle Funktion aufgerufen, in der die pro Frame veränderlichen *command buffer* aufgenommen werden. Diese ist im Fall dieser Applikation in der `RTCombinedApp`-Klasse implementiert und füllt die *command buffer* für den jeweiligen Frame. Der Inhalt dieser Funktion ist in Abschnitt 5.4.3 beschrieben. Sind die *command buffer* aufgenommen,

Abbildung 9: Die übergeordnete Synchronisierung innerhalb eines Frames zwischen den einzelnen *submit*-Operationen und Präsentations-Operationen.

wird der *submit*, also das Einreichen der *commands* in die *queue*, ausgeführt. Das ist der Punkt, an dem die Grafikkarte anfängt, die Befehle auszuführen und damit das Bild zu rendern. Im *submit*-Befehl wird angegeben, dass auf den vorhin vorgestellten Semaphor, welcher angibt, dass das nächste Bild zum Präsentieren bereit ist, gewartet wird. Als Semaphor, welcher signalisiert wird, wenn die eingereichten *commands* vollständig ausgeführt worden sind, wird `m_graphicsRenderFinishedSemaphores.at(m_currentFrame)` genutzt. Dieser wird später benötigt, um vor dem Rendering der Benutzeroberfläche zu warten, bis das Rendering des eigentlichen Bildes abgeschlossen ist.

Direkt danach wird die überschriebene Funktion, welche den *command buffer* für die Benutzeroberfläche aufbaut und den *submit* vornimmt, aufgerufen. In der Klasse `BaseApp` signalisiert diese den Semaphor, der angibt, dass das Rendering der Benutzeroberfläche abgeschlossen ist, nach dem Warten auf den zuvor erwähnten Semaphor, welcher angibt, dass das eigentliche Bild fertig ist und setzt den am Anfang dieses Abschnitts beschriebenen `vk::Fence` zurück, um anzugeben, dass das Rendering für diesen Frame jetzt abgeschlossen ist. Dies ist dafür gedacht, dass dieselbe `drawFrame()`-Funktion benutzt werden kann, egal ob eine Unterklasse eine Benutzeroberfläche anzeigen will oder nicht. In dieser Anwendung, also in der Klasse `RTCombinedApp` ist die Funktion `buildImguiCmdBufferAndSubmit(imageIndex)` überschrieben und schreibt die *commands*, die von *Dear Imgui* aus den in Abschnitt 5.4.1 beschrieben Informationen generiert worden sind, in den für die Benutzeroberfläche gedachten *command buffer*, der auch den dafür gedachten Renderpass enthält.

```
m_imguiCommandBuffers.at(imageIndex).beginRenderPass(
  imguiRenderpassInfo, vk::SubpassContents::eInline);

ImGui_ImplVulkan_RenderDrawData(ImGui::GetDrawData(),
  m_imguiCommandBuffers.at(imageIndex));

m_imguiCommandBuffers.at(imageIndex).endRenderPass();
```

Zeitstempel werden ebenfalls geschrieben. Der *command buffer* wird dann auf die *graphics queue* eingereicht, dabei auf den Semaphor gewartet, der angibt, dass das eigentliche Bild fertig gerendert wurde und zum Signalisieren

der Semaphor referenziert, der angibt, dass die Benutzeroberfläche fertig gerendert wurde. Dann wird der `vk::Fence` freigegeben, um anzugeben, dass der Frame mit diesem Index wieder frei ist.

Nachdem diese beiden *submit*-Operationen durchgeführt worden sind, kann das Präsentieren des Frames auf dem Bildschirm vorgenommen werden. Dafür wird als Semaphor, auf den gewartet werden soll, derjenige angegeben, der besagt, dass die Benutzeroberfläche fertig gerendert ist, angegeben. Eine Übersicht über den bis hierhin beschriebenen Synchronisationsprozess zwischen den einzelnen *submit*-Operationen und der Präsentations-Operation ist in Abbildung 9 gezeigt.

Es ist anzumerken, dass zwischen Befehlen durch das Angeben der Semaphoren, auf die gewartet werden soll beziehungsweise die signalisiert werden sollen wenn der Befehl ausgeführt wurde, eine Abfolge aus reinen GPU-GPU-Abhängigkeiten festgelegt wird. Deshalb können *submits* in Vulkan auf der GPU eingereiht werden und die Ausführungsreihenfolge ergibt sich aus den Abhängigkeiten. So ist keine CPU-GPU-Synchronisierung innerhalb eines Frames nötig und wegen des *triple buffering* können sogar mehrere Frames ohne CPU-GPU-Synchronisierung hintereinander eingereiht werden. Dies ermöglicht einen höheren Grad der Entkopplung der CPU- und GPU-Seite des Programms und garantiert so eine bessere Performance, da bei einem langsameren Frame auf GPU-Seite die CPU-Seite trotzdem weiter arbeiten und *command buffer* füllen kann und bei einem langsamerem Frame auf CPU-Seite die GPU-Seite nicht warten muss, da noch mehr Frames darauf warten, abgearbeitet zu werden.

5.4.3 Die Rendering-Commands

In den folgenden Abschnitten wird der gesamte Verlauf der Rendering-Commands in der Reihenfolge beschrieben, in der diese ausgeführt werden. Dadurch, dass die *command buffer*, deren Inhalt sich nicht ändert, bereits beim Initialisieren des Programms aufgenommen werden, unterscheidet sich die Reihenfolge des Ausführens von der des Aufnehmens.

Der tatsächliche Ablauf der Ausführung beginnt damit, dass der übergeordnete primäre *command buffer*, `m_commandBuffers.at(currentImage)`, welcher alle anderen sekundären *command buffer* später beinhaltet, zurückgesetzt wird und neu beginnt. Dies muss in jedem Frame passieren, denn nur dann können wieder *commands* in den *command buffer* geschrieben werden, nachdem die Aufnahme beendet wurde, was für das Einreichen notwendig ist.

Aktualisieren der Beschleunigungsdatenstruktur Als ersten inhaltlichen Schritt im Ablauf der GPU-Operationen wird die Beschleunigungsdatenstruktur aktualisiert. Dies geschieht nur, wenn über die Benutzeroberfläche die Animation in der Szene aktiviert wird. Dann kann der Nutzer den Index eines Objektes auswählen, welches zu Demonstrationszwecken in der Szene

bewegt werden soll. Für die Animation wird dann zuerst die Modelmatrix des Objekts verändert und im Modelmatrix-Buffer für das normale Rendering und im Instance-Buffer für die Beschleunigungsdatenstruktur aktualisiert. Im Anschluss kann mit dem Befehl `vkCmdBuildAccelerationStructureNV` die übergeordnete Beschleunigungsdatenstruktur entweder aktualisiert oder komplett neu aufgebaut werden. Was gemacht werden soll, wird auch über die Benutzeroberfläche eingegeben. Die Signatur des Befehls ist wie folgt[26]:

```
void vkCmdBuildAccelerationStructureNV(
  VkCommandBuffer                               commandBuffer,
  const VkAccelerationStructureInfoNV*          pInfo,
  VkBuffer                                      instanceData,
  VkDeviceSize                                  instanceOffset,
  VkBool32                                      update,
  VkAccelerationStructureNV                     dst,
  VkAccelerationStructureNV                     src,
  VkBuffer                                      scratch,
  VkDeviceSize                                  scratchOffset);
```

Wenn die Beschleunigungsdatenstruktur aktualisiert werden soll, muss als `update`-Parameter `true` übergeben werden, sonst `false`. Als Parameter `dst` muss immer die Beschleunigungsdatenstruktur angegeben werden, bei einem Update auch als `src`-Parameter, bei einem Neu-Aufbau `nullptr`. Der *scratch buffer*, der beim erstmaligen Bauen der Beschleunigungsdatenstruktur verwendet wurde, wird hier wiederverwendet. Vor und nach dem Aktualisieren oder Aufbauen der Beschleunigungsdatenstruktur muss der Zugriff auf diese synchronisiert werden und die Beschleunigungsdatenstruktur muss das richtige Layout für das den *build*-Befehl beziehungsweise den Zugriff im Shader besitzen. Diese Synchronisierung wird mit *memory barriers* vor und nach dem *command* erreicht. Außerdem wird vorher und nachher ein von einem `Timer`-Objekt verwalteter Zeitstempel geschrieben.

An dieser Stelle wurde außerdem getestet, das Aktualisieren der Beschleunigungsdatenstruktur asynchron zum Rendering auf der *compute queue* auszuführen. Dafür wurde ein eigener *command buffer*, welcher aus dem der *compute queue* zugeordnetem *command pool* allokiert wurde, genutzt und die *commands* in die *compute queue* eingereicht. Die für die Beschleunigungsdatenstruktur und deren Aufbau benötigten Ressourcen (*instance buffer*, *scratch buffer*) wurden so angelegt, dass sie zwischen *queues* geteilt werden können. Die Synchronisierung wurde mit *memory barriers*, Semaphoren und zu Debugging-Zwecken auch mit dem Warten auf das Abschließen aller eingereihten *commands* getestet. Bei allen getesteten Versionen stürzt das Programm mit einem *device lost*-Error ohne eine Fehlermeldung der *Validation Layer* ab. Wird das Aktualisieren der Beschleunigungsdatenstruktur mit dem Ausführen eines simplen Compute Shaders ersetzt, ist das asyn-

[26] `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/man/html/vkCmdBuildAccelerationStructureNV.html`, letzter Abruf: 5. Juni 2019

chrone Ausführen möglich. Dies weist darauf hin, garantiert allerdings nicht, dass das Problem nicht auf der Anwendungsseite, sondern beispielsweise auf Treiber-Ebene liegt. Der Programmcode für das Aktualisieren der Beschleunigungsdatenstruktur auf der *compute queue* verbleibt im Programm, um Tests mit neueren Treibern und neueren Versionen der *Validation Layer*, welche dann, falls es wider Erwarten Fehler auf Seite der Anwendung geben sollte, auf diese hinweisen, wenn der Fehlerfall in die *Validation Layer* aufgenommen wurde.

Commands für Rendering in den G-Buffer Der erste Rendering-Schritt ist das Füllen des G-Buffers mit den Szenen-Informationen. Zuerst muss dafür der G-Buffer-Renderpass gestartet werden. Dies geschieht aus dem primären *command buffer*. Das Starten des Renderpasses setzt gleichzeitig die Attachments auf die übergebenen *clear values* zurück. Dies ist in Vulkan die einfachste Art, die Inhalte einer Textur zu überschreiben, da keine zusätzlichen Änderungen des Layouts gemacht werden müssen.

Nach dem Starten des Renderpasses wird der sekundäre *command buffer*, welcher die mittels *push constants* an die Shader zu übergebenden Daten enthält, ausgeführt. Dieser wird ebenfalls wie aktuell der übergeordnete *command buffer* pro Frame aufgenommen und ist der einzige sekundäre *command buffer*, bei dem dies geschieht. Der folgende Quellcode-Ausschnitt zeigt die Nutzung von *push constants* für die View-Matrix. Analog geschieht dies für die Projektionsmatrix, die Kameraposition, die Belichtung und die Information, ob die Reflexionen in halber Auflösung berechnet werden.

```
m_perFrameSecondaryCommandBuffers.at(currentImage).pushConstants(
  m_gbufferPipelineLayout, vk::ShaderStageFlagBits::eVertex |
  vk::ShaderStageFlagBits::eFragment, 0, sizeof(glm::mat4),
  glm::value_ptr(m_camera.getView()));
```

Nach der Referenzierung dieses sekundären *command buffers* wird der sekundäre *command buffer*, welcher die *commands* für das Rendering in den G-Buffer enthält referenziert. Der übergeordnete, primäre *command buffer* sieht bis zu diesem Punkt wie folgt aus:

```
m_commandBuffers.at(currentImage).beginRenderPass(
  renderpassInfo, vk::SubpassContents::eSecondaryCommandBuffers);

// execute command buffer which updates per-frame information
m_commandBuffers.at(currentImage).executeCommands(
  m_perFrameSecondaryCommandBuffers.at(currentImage));

// execute command buffers which contains rendering commands
m_commandBuffers.at(currentImage).executeCommands(
  m_gbufferSecondaryCommandBuffers.at(currentImage));

m_commandBuffers.at(currentImage).endRenderPass();
```

Die eigentlichen *commands* für das Rendering in den G-Buffer sind also in `m_gbufferSecondaryCommandBuffers` enthalten. Dieser *command buffer* in dreifacher Ausführung ist statisch und wird bereits bei Initialisierung der Anwendung in der Funktion `createAllCommandBuffers()` gefüllt.

Um einen sekundären *command buffer* innerhalb eines laufenden Renderpass einzubinden, muss dieser beim `begin`-Befehl eine Struktur vom Typ `vk::CommandBufferInheritanceInfo` enthalten, in der der Renderpass, in dem der *command buffer* ausgeführt wird und das Framebuffer-Objekt, in das gerendert wird, referenziert wird. Beim Aufnehmen der drei *command buffer* wird in den *command buffer* mit Index i auch immer das Framebuffer-Objekt mit dem Index i referenziert. Zudem muss der *command buffer* mit der `vk::CommandBufferUsageFlagBits::eRenderPassContinue`-Flagge gestartet werden. Nach dem Starten beinhaltet der *command buffer* die folgenden *commands*: Zuerst wird ein Zeitstempel geschrieben, dann werden die Pipeline, der Vertex-Buffer, der Index-Buffer und das G-Buffer-Descriptor Set gebunden. Dieses Descriptor Set liegt wie in Abschnitt 5.3.12 beschrieben nicht in dreifacher Ausführung vor, da die Ausgabe dieses Renderpasses über die Framebuffer geregelt wird. Nachdem alle Ressourcen gebunden sind, wird der *draw call* ausgeführt. Der gesamte *command buffer* sieht also wie folgt aus:

```
vk::CommandBufferInheritanceInfo inheritanceInfo(
  m_gbufferRenderpass, 0, m_gbufferFramebuffers.at(i), 0, {}, {});
vk::CommandBufferBeginInfo beginInfo(
  vk::CommandBufferUsageFlagBits::eSimultaneousUse |
  vk::CommandBufferUsageFlagBits::eRenderPassContinue,
  &inheritanceInfo);

m_gbufferSecondaryCommandBuffers.at(i).begin(beginInfo);

m_timerManager.writeTimestampStart("1 G-Buffer",
  m_gbufferSecondaryCommandBuffers.at(i),
  vk::PipelineStageFlagBits::eAllGraphics, i);

m_gbufferSecondaryCommandBuffers.at(i).bindPipeline(
  vk::PipelineBindPoint::eGraphics, m_gbufferGraphicsPipeline);
m_gbufferSecondaryCommandBuffers.at(i).bindVertexBuffers(
  0, m_vertexBufferInfo.m_Buffer, 0ull);
m_gbufferSecondaryCommandBuffers.at(i).bindIndexBuffer(
  m_indexBufferInfo.m_Buffer, 0ull, vk::IndexType::eUint32);
m_gbufferSecondaryCommandBuffers.at(i).bindDescriptorSets(
  vk::PipelineBindPoint::eGraphics, m_gbufferPipelineLayout, 0, 1,
  &m_gbufferDescriptorSets.at(0), 0, nullptr);

m_gbufferSecondaryCommandBuffers.at(i).drawIndexedIndirect(
  m_indirectDrawBufferInfo.m_Buffer, 0,
  static_cast<uint32_t>(m_scene.getDrawCommandData().size()),
  sizeof(std::decay_t<decltype(
    *m_scene.getDrawCommandData().data())>));
```

```
m_timerManager.writeTimestampStop("1 G-Buffer",
  m_gbufferSecondaryCommandBuffers.at(i),
  vk::PipelineStageFlagBits::eAllGraphics, i);
m_gbufferSecondaryCommandBuffers.at(i).end();
```

Die Shader, die hier benutzt werden, werden im folgenden Abschnitt erklärt.

Shader für Rendering in den G-Buffer Der Vertex-Shader in der Datei `gbuffer.vert` transformiert die Vertices der Objekte in der Szene und reicht deren Attribute weiter.

```
#version 460
#extension GL_ARB_separate_shader_objects : enable

layout(std430, set = 0, binding = 0) readonly buffer ↩
    modelMatrixSSBO
{
  mat4 model[];
} mms;

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec2 inTexCoord;
layout(location = 2) in vec3 inNormal;

layout(location = 0) out vec2 fragTexCoord;
layout(location = 1) flat out int drawID;
layout(location = 2) out vec3 passNormal;
layout(location = 3) out vec3 passWorldPos;

layout (push_constant) uniform perFramePush
{
  mat4 view;
  mat4 proj;
} matrices;

void main()
{
  gl_Position = matrices.proj * matrices.view *
    mms.model[gl_DrawID] * vec4(inPosition, 1.0);
  fragTexCoord = inTexCoord;
  drawID = gl_DrawID;
  passNormal = inNormal;
  passWorldPos =
    vec3(mms.model[gl_DrawID] * vec4(inPosition, 1.0));
}
```

Für den Zugriff auf die Model-Matrix und für das Weiterreichen einer Objekt-ID wird die GLSL-builtin-Variable `gl_DrawID` benutzt, welche den Index des Objekts in einem *draw call*, der mehrere Objekte rendert, angibt. Zu den *push constants* ist anzumerken, dass hier nur ein Teil der auf CPU-Seite angegebenen *push constants* verwendet wird. Dies ist möglich, so lange es sich um den Anfang der *push constants* handelt, aus der Mitte können keine Daten ausgelassen werden.

Im Fragment Shader `gbuffer.frag` werden die aus dem Vertex-Shader übergebenen interpolierten Daten in die G-Buffer-Texturen geschrieben.

```
#version 460
#extension GL_ARB_separate_shader_objects : enable

layout(location = 0) in vec2 fragTexCoord;
layout(location = 1) flat in int drawID;
layout(location = 2) in vec3 passNormal;
layout(location = 3) in vec3 passWorldPos;

layout(location = 0) out vec4 gbufferPosition;
layout(location = 1) out vec4 gbufferNormal;
layout(location = 2) out vec4 gbufferUV;

layout(constant_id = 0) const int NUM_TEXTURES = 64;
layout(set = 0, binding = 1) uniform sampler2D ↩
    allTextures[NUM_TEXTURES];

struct PerMeshInfoPBR
{
  // standard
  uint    indexCount;
  uint    instanceCount;
  uint    firstIndex;
  int     vertexOffset;
  uint    firstInstance;
  // additional
  int texIndexBaseColor;
  int texIndexMetallicRoughness;
  int assimpMaterialIndex;
};

layout(std430, set = 0, binding = 2) readonly buffer ↩
    indirectDrawBuffer
{
  PerMeshInfoPBR perMesh[];
} perMeshInfos;

void main()
{
  gbufferPosition = vec4(passWorldPos, drawID);
  gbufferNormal = vec4(passNormal, 0.0f);
  vec2 lod =
    textureQueryLod(
      allTextures[perMeshInfos.perMesh[drawID].texIndexBaseColor],
      fragTexCoord);
  gbufferUV = vec4(fragTexCoord, lod.x, lod.y);
}
```

Dabei wird die Objekt-ID in den vierten Kanal der Positions-Textur geschrieben. Das *level of detail* für die Texturierung, welches sonst automatisch mit Geometrie-Ableitungen ermittelt wird, wird mit `textureQueryLod` ausgelesen und in den dritten und vierten Kanal der Texturkoordinaten-Textur

geschrieben. So kann die letztendliche Texturierung im Beleuchtungs-Shader erfolgen, der ohne das Rendern tatsächlicher Geometrie normalerweise keinen Zugriff auf diese Daten hätte. Am Beispiel dieses Shaders lassen sich auch die *specialization constants* zeigen. `NUM_TEXTURES` wird als Standardwert auf 64 gesetzt, beim Kompilieren des Shaders von SPIR-V zu Maschinencode (nicht beim Kompilieren des Shaders von GLSL zu SPIR-V) wird dies wie in Abschnitt 5.3.14 beschrieben durch die Anzahl der Texturen in der Szene ersetzt. So kann `NUM_TEXTURES` als Längenangabe für das Array `allTextures` verwendet werden, da es als konstanter Wert zählt. Damit ist das Rendering in den G-Buffer abgeschlossen.

Aktualisieren der Daten im pro-Frame-Buffer für das Ray Tracing Als nächster Schritt wird der Buffer, in dem die pro Frame veränderlichen Daten stehen, aktualisiert. Dies geschieht aus dem primären *command buffer* heraus. Zuerst wird der Buffer mit einer `vk::BufferMemoryBarrier` in den Zustand gebracht, dass man Daten hinein transferieren kann. Für das Aktualisieren der Daten wird der *command* `vkCmdUpdateBuffer` verwendet. Dieser schreibt bis zu 65536 Bytes an Daten in den *command buffer* selbst hinein, welche dann in den Buffer geladen werden, wenn der *command buffer* an die GPU übergeben wird[27]. Dies hat den Vorteil, dass die Notwendigkeit eines zusätzlichen Buffers, in den man die Daten erst schreibt und diese dann mit einem *copy buffer-command* in den eigentlichen Buffer hineinkopiert, entfällt.

Die Daten, die aktualisiert werden, sind die Kameraposition, die Anzahl der Frames für das Akkumulieren der Bilder, die Anzahl der Samples jeweils für Umgebungsverdeckung und Reflexionen, der Radius für die Umgebungsverdeckung, ein Rauheitsschwellwert für die Reflexionen und ob die Reflexionen in niedriger Auflösung berechnet werden sollen. Wurden alle `vkCmdUpdateBuffer`-*commands* aufgenommen, wird der Buffer wieder mit einer `vk::BufferMemoryBarrier` in den Zustand zurückversetzt, in dem er in einem Ray Tracing-Shader gelesen werden kann.

Commands für Schatten mit Ray Tracing Nachdem die *commands* für das Aktualisieren der Daten für das Ray Tracing im primären *command buffer* aufgenommen sind, wird der sekundäre *command buffer* für das Ray Tracing der Schatten eingebunden.

```
m_commandBuffers.at(currentImage).executeCommands(
  m_rtSoftShadowsSecondaryCommandBuffers.at(currentImage));
```

Auch dieser *command buffer* in dreifacher Ausführung wird in der Initialisierungsphase bereits vollständig aufgenommen. Da die Ray Tracing-*commands* keinen `vk::Renderpass` benötigen, muss keiner vom primären *command*

[27] `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/man/html/vkCmdUpdateBuffer.html`, letzter Abruf: 5. Juni 2019

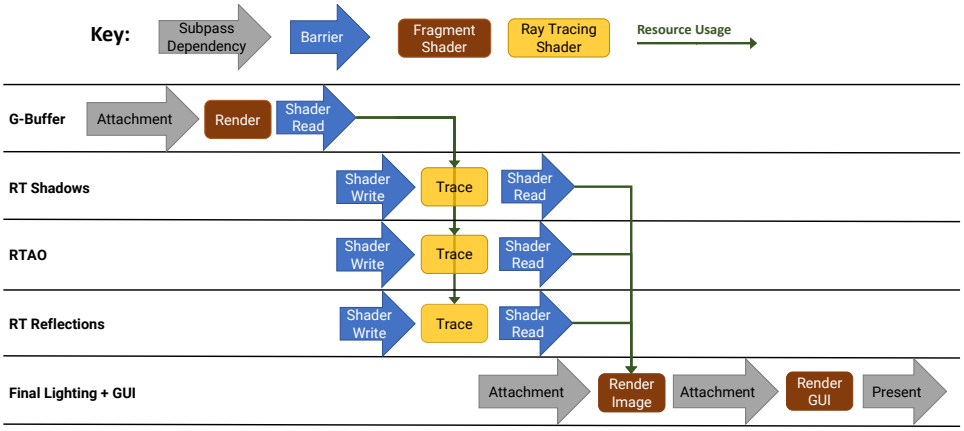


Abbildung 10: Die Synchronisierung der Ergebnis-Ressourcen der einzelnen Rendering-Schritte. Der zeitliche Ablauf ist von links nach rechts und von oben nach unten gegeben.

buffer gestartet und keiner vom sekundären *command buffer* referenziert werden. Es wird die passende Pipeline `m_rtSoftShadowsPipeline` gebunden, genau wie die folgenden *descriptor sets*: Das für die Eingaben und das für die Ausgabebilder mit dem jeweiligen zum *command buffer* passenden Index und das *descriptor set*, welches die Buffer mit den Daten der Lichtquellen enthält.

```
m_rtSoftShadowsSecondaryCommandBuffers.at(i).bindPipeline(
  vk::PipelineBindPoint::eRayTracingNV, m_rtSoftShadowsPipeline);

std::array dss = {
  m_rtSoftShadowsDescriptorSets.at(i),
  m_lightDescriptorSet,
  m_shadowImageStoreDescriptorSets.at(i)
};

m_rtSoftShadowsSecondaryCommandBuffers.at(i).bindDescriptorSets(
  vk::PipelineBindPoint::eRayTracingNV,
  m_rtSoftShadowsPipelineLayout, 0,
  static_cast<uint32_t>(dss.size()), dss.data(), 0, nullptr);
```

Bevor das Ray Tracing gestartet werden kann, muss der Zugriff auf die G-Buffer-Texturen synchronisiert und das Texturlayout für das Lesen im Ray Tracing Shader gewechselt werden. Eine `vk::ImageMemoryBarrier` transformiert die Texturen vom Zugriff als Attachment und dem Layout `vk::ImageLayout::eColorAttachmentOptimal` an der Pipeline-Stage `vk::PipelineStageFlagBits::eColorAttachmentOutput` zum Lesen im Shader mit `vk::ImageLayout::eShaderReadOnlyOptimal` als Layout in der Pipeline-Stage für Ray Tracing Shader. Außerdem wird eine weitere `vk::ImageMemoryBarrier` genutzt, um das Output-Image für das Schreiben im Ray Generation Shader vorzubereiten. Dies wird hier getan, da es später, vor dem Lesen im Fragment Shader bei der finalen

Beleuchtung, dafür transformiert wurde und jetzt wieder zurücktransformiert wird. Um immer den gleichen *command buffer* nutzen zu können, wird die Textur nach dem Anlegen in der Initialisierungsphase der Anwendung bereits in das Layout gebracht, das es auch am Ende eines Frames hätte. Nach diesen beiden Barrieren wird Timer analog zu dem vom Rendering in den G-Buffer benutzt, um einen Zeitstempel zu schreiben. Die Synchronisierung aller Bild-Ressourcen, also des G-Buffers, des finalen Bildes und der Ray Tracing-Ergebnisbilder, ist in Abbildung 10 gezeigt.

Nun folgt der *trace rays-command*, welcher das Ray Tracing initialisiert, indem der Ray Generation Shader aufgerufen wird.

```
vkCmdTraceRaysNV(m_rtSoftShadowsSecondaryCommandBuffers.at(i),
  // ray gen
  m_rtSoftShadowSBTInfo.m_Buffer, 0,
  // miss
  m_rtSoftShadowSBTInfo.m_Buffer,
  1 * m_context.getRaytracingProperties().shaderGroupHandleSize,
  m_context.getRaytracingProperties().shaderGroupHandleSize,
  // (any) hit
  nullptr, 0, 0,
  // callable
  nullptr, 0, 0,
  // ray gen dimensions
  m_context.getSwapChainExtent().width,
  m_context.getSwapChainExtent().height, 1
);
```

In diesem wird erst der Buffer angegeben, der die *shader binding table* enthält. In dieser steht an der Stelle 0 der Ray Generation Shader, was hier als `offset` mit übergeben wird. Danach wird der Buffer für den Miss Shader angegeben, dieser ist der gleiche, und der Miss Shader steht in diesem Fall direkt hinter dem Ray Generation Shader, also ist der *offset* 1 multipliziert mit der Größe eines Shader-Eintrages im Buffer. Die Schrittweite zwischen Shadern ist ebenfalls so groß wie die Größe eines Shader-Eintrages im Buffer. Diese lassen sich manuell angeben, da einerseits beliebige Buffer benutzt werden können, um die *shader binding table* zu enthalten oder auch ein Buffer für mehrere, nicht-zusammengehörige Ray Tracing Shader benutzt werden kann. Andererseits ist es auch möglich, Daten, welche im Shader zur Verfügung stehen sollen, in den Buffer mit der *shader binding table* zu schreiben, so wäre eine andere Schrittweite zwischen Shadern nötig. Dies wird in dieser Anwendung allerdings nicht genutzt. Nach dem Buffer, dem *offset* und der Schrittweite für den Miss Shader kämen dieselben Einträge für die Hit Shader-Gruppe, welche bei den Schatten nicht benötigt wird und deshalb leer ist, und Callable Shader, welche beliebigen, aus Shadern aufrufbaren Programmcode enthalten können, welche ebenfalls nicht verwendet werden. Als letzte Parameter kommen die Dimensionen in X-, Y- und Z-Richtung für die Ausführung des Ray Generation Shaders, welchen man sich hierbei wie einen Compute Shader vorstellen kann. Die Dimensionen sind hier die

Breite und Höhe des Bildes, es wird folglich ein Ray Generation Shader-Thread pro Pixel im Bild ausgeführt. Nach dem *trace rays-command* wird nochmal ein Zeitstempel geschrieben, um in Kombination mit dem davor geschriebenen Zeitstempel die Dauer der Ausführung der Ray Tracing-Shader zu berechnen. Außerdem werden die Texturen, in welche die Daten für die Schatten im Ray Generation Shader geschrieben werden, wieder mit Barrieren zurücktransformiert, so dass später im Fragment Shader aus ihnen gelesen werden kann.

Shader für mit Ray Tracing berechnete Schatten Der Ray Generation Shader `softshadowPBR.rgen` wird an dieser Stelle inhaltlich vorgestellt und genutzt, um daran die Funktionsweise der Ray Tracing Shader einzuführen. Aus Übersichtsgründen wird nicht der gesamte Shader als Quellcode-Ausschnitt eingeführt, sondern wichtige Teile einzeln erklärt.

Am Anfang des Shaders wird die *ray payload* deklariert. In dieser können beliebige Daten stehen, welche von den Ray Tracing Shadern gelesen und geschrieben werden können. Im Fall dieses Shaders handelt es sich um einen vorzeichenlosen Integer-Wert, welcher angibt, ob die eine Lichtquelle erreicht wird oder nicht.

```
#version 460
#extension GL_NV_ray_tracing : require
layout(location = 0) rayPayloadNV uint hitValue;
```

Außerdem wird die Beschleunigungsdatenstruktur auf dieselbe Weise wie ein Buffer oder eine Textur angegeben.

```
layout(set = 0, binding = 0) uniform accelerationStructureNV ↩
    topLevelAS;
```

Der Shader hat außerdem Zugriff auf die Positionstextur des G-Buffers, die Zufallszahlentextur, den Buffer mit den pro-Frame-Informationen, in den folgenden Quellcode-Ausschnitten mit `perFrameInfo` bezeichnet. Als Ausgabe werden die Arrays von Bildern benutzt, welche pro Lichtquelle des jeweiligen Typs ein Layer haben:

```
layout(set = 2, binding = 0, r32f) uniform image2DArray ↩
    softShadowDirectionalImage;
layout(set = 2, binding = 1, r32f) uniform image2DArray ↩
    softShadowPointImage;
layout(set = 2, binding = 2, r32f) uniform image2DArray ↩
    softShadowSpotImage;
```

In der `main()`-Funktion des Shaders werden zuerst die Koordinaten für den Texturzugriff aus dem G-Buffer, also im Bereich von 0 bis 1 berechnet. Dafür kommen die für Ray Tracing-Shader spezifischen built-in Variablen `gl_LaunchIDNV`, welche den Index des Threads zwischen 0 und der Anzahl der Threads in die jeweilige Richtung angibt, und `gl_LaunchSizeNV`, welche die gesamte Anzahl Threads in alle Richtungen angibt, welche im *trace*

rays-command festgelegt wird. Danach wird die Generierung der Zufallszahlen initialisiert, was in Abschnitt 5.5 gesondert beschrieben wird.

Das Berechnen der Schatten mit Ray Tracing funktioniert so, dass von der sichtbaren Szenengeometrie aus Strahlen zu allen Lichtquellen geschossen werden. Schneiden diese Strahlen auf dem Weg zur jeweiligen Lichtquelle andere Szenengeometrie, ist die Lichtquelle verdeckt, es gibt also Schatten. Werden mehrere Strahlen pro Lichtquelle verschickt und zählt diese als flächige Lichtquelle, wird wie in Abschnitt 2.5 beschrieben, aus dem Verhältnis von Strahlen, welche die Lichtquelle erreichen und Strahlen, welche von Szenengeometrie aufgehalten werden, ein Wert berechnet, der eine teilweise Verschattung (Penumbra) angibt. Dafür wird nun zunächst der `origin`-Parameter für das Ray Tracing, also der Punkt, von dem aus der Strahl verschickt wird, auf die Position im G-Buffer an dem jeweiligen Pixel gesetzt.

```
vec3 origin = texture(gbufferPosSampler, inUV).xyz;
```

Ebenso werden die *ray flags*, welche Einstellungen für das Ray Tracing repräsentieren, initialisiert.

```
uint rayFlags = gl_RayFlagsTerminateOnFirstHitNV |
                gl_RayFlagsOpaqueNV |
                gl_RayFlagsSkipClosestHitShaderNV;
```

Die Einstellung `gl_RayFlagsTerminateOnFirstHitNV` bedeutet, dass es egal ist, ob ein gefundener Schnittpunkt der vorderste Schnittpunkt ist, das Ray Tracing soll hier schon gestoppt werden. Dies ist hier gewünscht, da nur relevant ist, ob Szenengeometrie geschnitten wird oder nicht, nicht wo der Schnittpunkt ist. Normalerweise wird dann bei einem Schnittpunkt direkt der Closest Hit Shader aufgerufen, ein solcher existiert in dieser Pipeline aber nicht und muss deshalb mit der Einstellung `gl_RayFlagsSkipClosestHitShaderNV` explizit ausgeschlossen werden. So wird bei einem Schnittpunkt kein Shader aufgerufen, bei keinem gefundenem Schnittpunkt für einen Strahl allerdings der Miss Shader. Wird die *ray payload* nun vor dem Verschicken jedes Strahls auf 0 initialisiert und im Miss Shader auf 1 gesetzt, sollte dieser aufgerufen werden, bedeutet eine 0, dass der aktuelle Ausgangspunkt im Schatten liegt (Lichtquelle nicht erreicht), und eine1, dass er nicht im Schatten liegt (Lichtquelle erreicht). Die Einstellung `gl_RayFlagsOpaqueNV` bedeutet, dass es sich um undurchsichtige Geometrie handelt, bei der kein Any Hit Shader benötigt wird, der für jeden Schnittpunkt erst entscheiden muss, ob dieser Schnittpunkt wirklich benutzt werden soll.

Außerdem werden die folgenden Parameter initialisiert:

```
uint cullMask = 0xff;
float tmin = 0.001;
float tmax = 100000.0;
```

Die `cullMask` kann dazu benutzt werden, bestimmte Geometrien von einzelnen Strahlen auszuschließen. Da diese aber mit der Maske aller Instanzen

beim Erstellen der Beschleunigungsdatenstruktur übereinstimmt, wird keine Geometrie ausgeschlossen. Der Parameter `tmin` gibt den Startwert des Strahls wie in Abschnitt 2.3.1 an, ab dem gefundene Schnittpunkte auch benutzt werden. Dieser ist so gewählt, dass ein Schnittpunkt an dem Punkt, von dem der Strahl ausgeht, vermieden wird. Der Parameter `tmax` gibt die maximale Länge des Strahls an und wird je nach Lichtquelle später überschrieben.

Es folgen drei Schleifen, eine pro Lichtquellen-Typ, welche für jede Lichtquelle Strahlen verschickt. Diese wird am Beispiel der Schleife für die Punktlichtquellen erklärt. Die Schleife geht über alle Punktlichtquellen, initialisiert den Schatten-Mittelwert auf 0 und lädt die Punktlichtquellen-Struktur aus dem Buffer.

```
for(int i = 0; i < pointLights.length(); i++)
{
  float pointShadowValue = 0.0f;
  PBRPointLight currentLight = pointLights[i];
```

Dann wird in einer inneren Schleife über die Anzahl der Samples pro Lichtquelle iteriert. Dieser Parameter kann über die Benutzeroberfläche verändert werden.

```
for(int j = 0; j < currentLight.numShadowSamples; j++)
```

In dieser Schleife werden nun die Daten für die einzelnen Strahlen zur Lichtquelle gesammelt und der Strahl geschickt. Die maximale Länge des Strahls ist die Distanz vom Ausgangspunkt zur Lichtquelle:

```
tmax = length(currentLight.position - origin);
```

Der Strahl direkt zur Lichtquelle wäre der Vektor vom Ausgangspunkt auf der Geometrie zur Lichtquelle. Da diese aber eine flächige Lichtquelle, in diesem Fall eine Kugel ist, müssen Strahlen auf einen zufälligen Punkt der Lichtquelle verschossen werden, um die gesamte Verschattung zu integrieren, wie in Abschnitt 2.8 beschrieben ist. Dafür werden bei einer kugelförmigen Lichtquelle wie von Barré-Brisebois et al. [Bar+19] vorgeschlagen und von Shirley et al. [Shi+19b] beschrieben Strahlen in einem Kegel, dessen Grundfläche die Lichtquelle darstellt, verschossen.

```
vec3 toLight = normalize(currentLight.position - origin);

float hyp =
  sqrt(tmax * tmax + currentLight.radius * currentLight.radius);
float cosTheta = tmax/hyp;
vec3 directionCone = generateConeDirection(cosTheta);
vec3 direction = rotateToNormal(directionCone, toLight);
tmax += currentLight.radius;
direction = normalize(direction);
```

Die Funktion `generateConeDirection()` generiert analog zu Shirley et al. [Shi+19b] Vektoren in einem Kegel mit dem angegebenen maximalen Öffnungswinkel `cosTheta`. Diese werden dann mit der Funktion `rotateToNormal` in die Richtung der Lichtquelle gedreht.

Als nächstes wird die *ray payload* auf 0 initialisiert, was wichtig ist, da wie zuvor beschrieben nur ein Miss Shader benutzt wird, und der Strahl wird verschickt.

```
traceNV(topLevelAS, rayFlags, cullMask,
  0 /*sbtRecordOffset*/, 0 /*sbtRecordStride*/, 0 /*missIndex*/,
  origin, tmin, direction, tmax,
  0 /*payload*/ // X here is location = X of the payload
);
```

Die built-in-Funktion `traceNV(...)` startet das Verschicken des Strahls und damit die Suche nach einem Geometrieschnittpunkt inklusive Traversierung der Beschleunigungsdatenstruktur mit den Ray Tracing Cores. Der erste Parameter ist die zu nutzende Beschleunigungsdatenstruktur, danach folgen die vorhin erstellten *ray flags* und die Maske zum Auslassen bestimmter Geometrie. Die folgenden Parameter ermöglichen die Nutzung optionaler, spezieller Shader, indem sie einen Index in die *shader binding table* angeben, an dem der Shader steht, der im jeweiligen Fall aufgerufen werden soll. So können verschiedene `traceNV(...)`-Aufrufe mit unterschiedlichen Parametern verschiedene Shader aufrufen. In diesem Fall gibt es keinen Hit Shader, deshalb sind die ersten beiden dieser Parameter 0, und nur einen Miss Shader an Index 0, also ist dieser Parameter auch 0. Der letzte Funktionsparameter ist die `location` der *ray payload*, welche in diesem Fall wie zuvor beschrieben, 0 ist.

Nun wird der Strahl verfolgt und nach Schnittpunkten gesucht. Wird ein Schnittpunkt gefunden, passiert wegen der Abwesenheit aller Hit Shader nichts, `hitValue` behält den Wert 0. Wird kein Schnittpunkt gefunden, was bedeutet, dass die Lichtquelle erreicht wurde, wird der Miss Shader `softshadow.rmiss` aufgerufen.

```
#version 460
#extension GL_NV_ray_tracing : require

layout(location = 0) rayPayloadInNV uint hitValue;

void main()
{
  hitValue = 1U; // light hit!!
}
```

Dieser setzt die *ray payload* `hitValue` auf 1. Dann wird der ermittelte Wert, 0 oder 1 auf die Variable `pointShadowValue` aufaddiert. Die innere Schleife wird beendet und `pointShadowValue` wird durch die Anzahl der Samples geteilt. So wird ein Durchschnitts-Teilschattenwert berechnet. Dies geschieht, wie zuvor bereits erwähnt, einmal pro Lichtquelle und muss anschließend in das Ergebnisbild geschrieben werden.

Beim Schreiben in das Ergebnisbild werden, so lange nicht vom Benutzer anders gewünscht, die Ergebnisse der vorherigen Frames und des aktuellen Frames akkumuliert. Dies geschieht nur, wenn die Kamera und die Szene sich

nicht bewegen und wird über die im pro-Frame-Buffer übergebene Anzahl der Frames (`perFrameInfo.frameSampleCount`) gesteuert. Bewegt sich die Kamera oder deaktiviert der Nutzer das Akkumulieren oder wird das Animieren der Szene aktiviert, wird diese Anzahl immer auf 0 gesetzt und keine Akkumulation findet statt. Die Akkumulation der Ergebnisse reduziert das in den einzelnen Frames vorhandene Rauschen für ein besseres visuelles Ergebnis.

```
float oldValue =
  imageLoad(softShadowPointImage, ivec3(gl_LaunchIDNV.xy, i)).x;
pointShadowValue =
  (perFrameInfo.frameSampleCount * invSampleCount)
  * oldValue + invSampleCount * pointShadowValue;
imageStore(softShadowPointImage, ivec3(gl_LaunchIDNV.xy, i),
  vec4(pointShadowValue));
```

Wie bereits beschrieben geschieht der zuletzt beschriebene Ablauf für jeden Lichtquellen-Typ. Der makroskopische Ablauf des Shaders ist als der folgende:

```
for(int i = 0; i < dirLights.length(); i++)
{
  ...
  for(int j = 0; j < currentLight.numShadowSamples; j++)
  {
    ...
  }
  ...
}
for(int i = 0; i < pointLights.length(); i++)
{
  ...
  for(int j = 0; j < currentLight.numShadowSamples; j++)
  {
    ...
  }
  ...
}
for(int i = 0; i < spotLights.length(); i++)
{
  ...
  for(int j = 0; j < currentLight.numShadowSamples; j++)
  {
    ...
  }
  ...
}
```

Bei den gerichteten Lichtquellen (*directional lights* wird die Richtung der Lichtquelle leicht zufällig verändert, um auch hier weiche Schatten zu erzeugen. Bei den Spotlights wird pro Sample ein Punkt auf einer Scheibe generiert, zu dem dann ein Strahl verschickt wird.

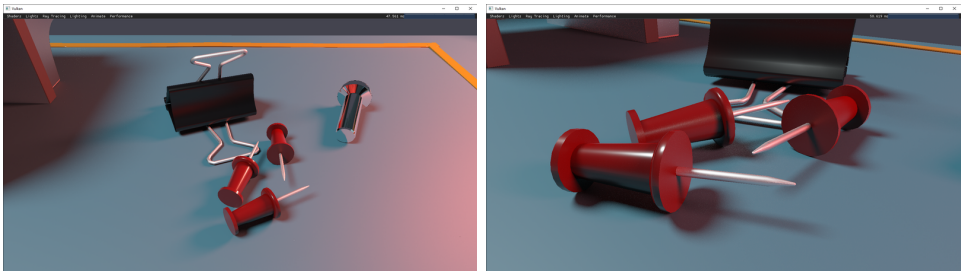


Abbildung 11: Schatten von zwei Lichtquellen mit akkumulierten Samples

Nach Ausführung des Shaders steht für jeden Punkt in der Szene, der im G-Buffer steht, ein Wert, der die volle, partielle oder nicht vorhandene Sichtbarkeit der jeweiligen Lichtquelle in jeder Schicht der Array-Texturen pro Lichtquellentyp angibt. In Abbildung 11 sind die Schatten zweier Lichtquellen in der in Abschnitt 6.1 beschriebenen Szene zu sehen.

Commands für die Berechnung der Umgebungsverdeckung mit Ray Tracing Analog zu den Schatten wird ein statischer, sekundärer *command buffer* in den primären *command buffer* aufgenommen.

```
m_commandBuffers.at(currentImage).executeCommands(
  m_rtAOSecondaryCommandBuffers.at(currentImage));
```

Dieser sekundäre *command buffer* bindet zuerst die zugehörige Pipeline. Als *descriptor sets* werden eins für die Eingabedaten, welches die Beschleunigungsdatenstruktur, die Positions- und Normalen-Texturen des G-Buffers, das Zufallszahlenbild und den pro-Frame-Buffer referenziert, und eins für die Ausgabedaten verwendet. Analog zu den Texturen, die die Schattenwerte speichern, wird auch die Textur, die die Umgebungsverdeckungswerte speichert, mit einer Barriere für das Schreiben im Ray Tracing-Shader synchronisiert, wie in Abbildung 10 gezeigt. Dann kommt auch hier der *trace rays-command*.

```
vkCmdTraceRays(m_rtAOSecondaryCommandBuffers.at(i),
  // ray gen
  m_rtAOSBTInfo.m_Buffer, 0,
  // miss
  m_rtAOSBTInfo.m_Buffer,
  2 * m_context.getRaytracingProperties().shaderGroupHandleSize,
  m_context.getRaytracingProperties().shaderGroupHandleSize,
  // hit
  m_rtAOSBTInfo.m_Buffer,
  1 * m_context.getRaytracingProperties().shaderGroupHandleSize,
  m_context.getRaytracingProperties().shaderGroupHandleSize,
  // callable
  nullptr, 0, 0,
  m_context.getSwapChainExtent().width,
  m_context.getSwapChainExtent().height, 1
);
```

In diesem wird im Gegensatz zum *command* bei den Schatten zusätzlich zum Miss Shader, der hier an Index 2 steht, der Index für die Hit Group der Shader angegeben, welche hier aus einem Closest Hit Shader besteht. Diese hat den Index 1. Die Schrittweite ist auch hier die Größe eines Shader-Handles im Buffer, da keine zusätzlichen Daten im Buffer stehen.

Nach dem Schreiben in die Textur im Ray Generation Shader wird auch diese wieder zurücktransformiert und synchronisiert für das Lesen im Fragment Shader. Es kommt ebenfalls ein Timer mit zwei Zeitstempeln zum Einsatz, um die Dauer der Shader-Ausführung zu messen.

Shader für die Berechnung der Umgebungsverdeckung mit Ray Tracing Im Ray Generation Shader `rtao.rgen` wird wie auch im Shader für die Schatten die Ray Payload deklariert, diesmal allerdings als `float`, da in dieser später der Abstand zum Schnittpunkt gespeichert wird.

```
layout(location = 0) rayPayloadNV float hitValue;
```

Dann werden in Analogie zum bereits vorgestellten Ray Generation Shader die Texturkoordinaten für den Zugriff auf den G-Buffer berechnet und die Zufallszahlenberechnung wird initialisiert. Zusätzlich zu `origin`, dem Ausgangspunkt des Strahls, wird hier die Normale aus dem G-Buffer geladen.

```
vec3 normal = normalize(texture(gbufferNormalSampler, inUV).xyz);
```

Als Einstellungen für den Strahl wird hier nur angegeben, dass es sich um undurchsichtige Geometrie handelt. Es wird im Gegensatz zum Schatten ein Closest Hit Shader eingesetzt und das Finden des vordersten Schnittpunkts ist hier relevant:

```
uint rayFlags = gl_RayFlagsOpaqueNV;
uint cullMask = 0xff;
float tmin = 0.001;
float tmax = perFrameInfo.RTAORadius;
```

Die Maske zum Auslassen von Geometrie ist hier wie bei den Schatten die, die bei Instanzen in der Beschleunigungsdatenstruktur angegeben wurde. Der Minimalwert für den Strahl ist wieder so gewählt, dass ein Schnittpunkt mit der Ausgangsgeometrie verhindert wird und der Maximalwert ist der Radius für die Umgebungsverdeckung und wird über die Benutzeroberfläche eingegeben.

Die Umgebungsverdeckung wird berechnet, indem von jedem Punkt in der Szene, dessen Position aus dem G-Buffer kommt, Strahlen in der oberen Hemisphäre verschickt werden. Wird bei einem Strahl ein Schnittpunkt gefunden, zählt dies als Verdeckung, wird kein Schnittpunkt bis zum maximalen Radius gefunden, zählt dies als offen. Das Verhältnis dieser Strahlen mit Verdeckung zu denen ohne Verdeckung ist der Umgebungsverdeckungswert. Zusätzlich wird bei einem Schnittpunkt der Strahl mit dem Abstand zum Schnittpunkt quadratisch gewichtet. Dies ist angelehnt an das Verfahren von Bavoil, Sainz und Dimitrov [BSD08].

Das eigentliche Berechnen der Umgebungsverdeckung geschieht in einer Schleife über die Anzahl der Samples, welche über die Benutzeroberfläche eingegeben wird.

```
float rtAOValue = 0.0f;

for(int i = 0; i < perFrameInfo.RTAOSampleCount; i++)
{
  vec3 direction =
    normalize(sampleRotatedCosineHemisphere(normal));

  traceNV(topLevelAS, rayFlags, cullMask,
    0 /*sbtRecordOffset*/, 0 /*sbtRecordStride*/, 0 /*missIndex*/,
    origin, tmin, direction, tmax,
    0 /*payload*/ // X here is location = X of the payload
  );

  if(hitValue > 0)
    rtAOValue += pow(hitValue / perFrameInfo.RTAORadius, 2);
  else
    rtAOValue += 1;
}

rtAOValue /= float(perFrameInfo.RTAOSampleCount);
```

Der Richtungsvektor für den zu verfolgenden Strahl wird von der Funktion `sampleRotatedCosineHemisphere(normal)` zurückgegeben, welche eine zufällige Richtung in der Halbkugel, die am eingegebenen Vektor ausgerichtet ist, generiert. Diese Richtungen sind cosinus-gewichtet, also befinden sich mehr im mittigen Teil der Halbkugel, wie von Barré-Brisebois et al. [Bar+19] vorgeschlagen. Die `traceNV(...)`-Funktion ist genau wie im Shader für die Schatten, da hier keine speziellen Shader angesprochen werden. Es wird nur ein Closest Hit Shader und einen Miss Shader verwendet, demnach haben beide den Index 0. Die restlichen Parameter wurden oben bereits eingeführt. Nachdem der Strahl verschickt wurde, wird bei einem Schnittpunkt der Closest Hit Shader `rtao.rchit` aufgerufen:

```
#version 460
#extension GL_NV_ray_tracing : require

layout(location = 0) rayPayloadInNV float hitValue;

void main()
{
  hitValue = gl_HitTNV; // ||S - P||
}
```

Dieser setzt die *ray payload* auf den Wert der built-in-Variable `gl_HitTNV`. Diese beinhaltet die Schrittweite auf dem Strahl, was der Faktor für die Multiplikation mit dem `direction`-Vektor ist, um vom Ausgangspunkt zum getroffenen Punkt zu kommen.

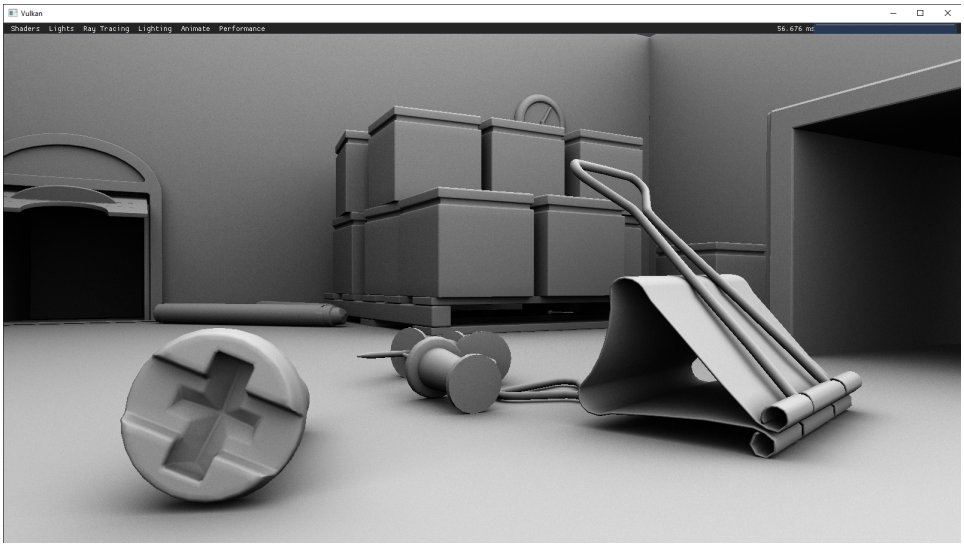


Abbildung 12: Umgebungsverdeckung einzeln mit akkumulierten Samples.

Wird kein Schnittpunkt gefunden, wird die *ray payload* vom Miss Shader `rtao.rmiss` auf -1 gesetzt.

```
#version 460
#extension GL_NV_ray_tracing : require

layout(location = 0) rayPayloadInNV float hitValue;

void main()
{
  hitValue = -1.0f;
}
```

Dies wird für die weiter oben gezeigte Fallunterscheidung am Ende der Schleife im Ray Generation Shader benötigt. Dann wird der Mittelwert aller Strahlen berechnet. Der finale Wert wird mit derselben Art der Akkumulation wie bei den Schatten in das Ausgabebild geschrieben.

In Abbildung 12 ist die Umgebungsverdeckung in der Szene, die in Abschnitt 6.1 beschriebenen ist, abgebildet. Auf dem Bild wir nur die Umgebungsverdeckung angezeigt mit einem Radius von 100, was hier über die Dimensionen der eigentlichen Szene hinausgeht. Demnach wird keine Geometrie von der Umgebungsverdeckungsberechnung an jeder beliebigen Stelle ausgeschlossen.

Commands für Reflexionen mit Ray Tracing Für die Reflexionen existieren jeweils zwei *command buffer* in dreifacher Ausführung, einer für das Berechnen der Relfexionen in hoher Auflösung in voller einer für die halbe Auflösung. Je nach Benutzereingabe wird einer der jeweiligen sekundären, statischen *command buffer* in den primären *command buffer* aufgenommen.

```
// execute command buffers for RT Reflections
if (m_useLowResReflections)
  m_commandBuffers.at(currentImage).executeCommands(
    m_rtReflectionsLowResSecondaryCommandBuffers.at(currentImage));
else
  m_commandBuffers.at(currentImage).executeCommands(
    m_rtReflectionsSecondaryCommandBuffers.at(currentImage);
```

Diese *command buffer* werden in einer dafür angelegten Lambda-Funktion erstellt, da sie bis auf die Auflösung, in der der Ray Tracing Shader ausgeführt wird, gleich sind. Es wird das Descriptor Set, in dem alle Ressourcen für die Berechnung der Reflexionen enthalten sind, gebunden. Dieses referenziert sowohl das Ausgabebild mit hoher als auch das Ausgabebild mit niedriger Auflösung. Außerdem wird das Descriptor Set mit den Daten zu den Lichqtuellen gebunden. Wie schon bei Schatten und Umgebungsverdeckung werden die Ergebnisbilder erst zum Schreiben im Ray Generation Shader synchronisiert und transformiert und nach dem *trace rays-command* wieder zurücktransformiert, wie in Abbildung 10 dargestellt ist. Außerdem wird wie zuvor vorher und nachher ein Zeitstempel geschrieben.

```
vkCmdTraceRaysNV(commandBuffer,
  // raygen
  m_rtReflectionsSBTInfo.m_Buffer, 0,
  // miss
  m_rtReflectionsSBTInfo.m_Buffer,
  2 * m_context.getRaytracingProperties().shaderGroupHandleSize,
  m_context.getRaytracingProperties().shaderGroupHandleSize,
  // closest hit
  m_rtReflectionsSBTInfo.m_Buffer,
  1 * m_context.getRaytracingProperties().shaderGroupHandleSize,
  m_context.getRaytracingProperties().shaderGroupHandleSize,
  // callable
  nullptr, 0, 0,
  extent.x, extent.y, 1
);
```

Hierbei ist der Closest Hit Shader an Index 1 und die zwei Miss Shader, starten an Index 2. Die Variable `extent` beinhaltet die Ausführungsdimensionen des Shaders, was bei der hochauflösenden Version die Größe des Fensters ist und bei der niedrig aufgelösten Version die halbe Seitenlänge, also ein Viertel der Pixel, beträgt.

Shader für Reflexionen mit Ray Tracing Das Prinzip, nachdem die Reflexionen berechnet werden, funktioniert folgendermaßen: Ausgehend von der Position im G-Buffer werden Reflexionsstrahlen je nach Oberflächenbeschaffenheit verschickt. Treffen diese auf weitere Geometrie, wird für diese die direkte Beleuchtung berechnet und ein Schattenfühler-Strahl zu jeder Lichtquelle geschickt. Die berechnete direkte Beleuchtung inklusive Schatten ist nun der Reflexionswert und wird in eine Textur geschrieben. So lassen

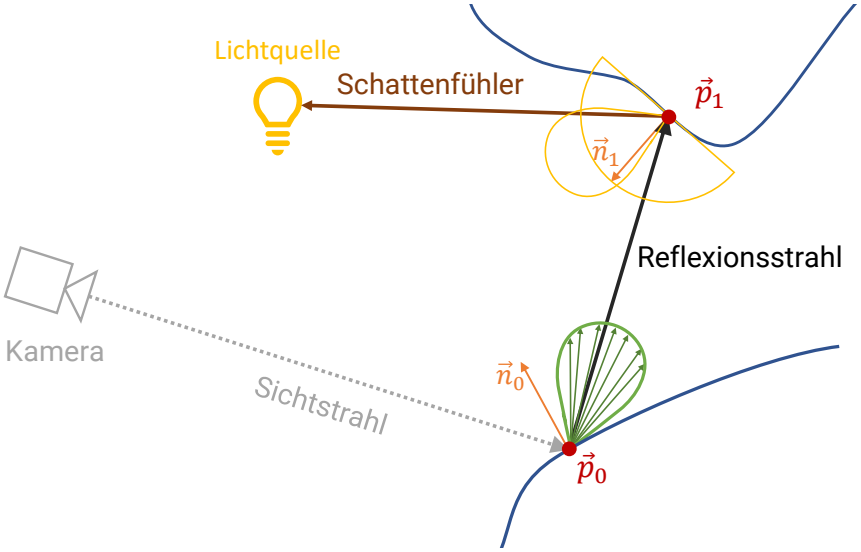


Abbildung 13: Schematische Darstellung der Reflexionsberechnung. Der Sichtstrahl dient hier nur der Verdeutlichung.

sich Reflexionen berechnen, in denen Beliebige Objekte der Szene zu sehen sein können. Dieser Ablauf ist in Abbildung 13 dargestellt. In dieser ist der Punkt $\vec{p_0}$ der Ausgangspunkt aus dem G-Buffer, der Sichtstrahl ist nur zur Verdeutlichung eingezeichnet. Der Punkt $\vec{p_1}$ ist der mittels Ray Tracing ermittelte Schnittpunkt, an dem die direkte Beleuchtung berechnet wird und von dem aus ein Schattenfühler zu jeder Lichtquelle verschickt wird.

Die genaue Umsetzung ist im Ray Generation Shader, gespeichert in der Datei `rtreflectionsPBR.rgen`, wie folgt: Die *ray payload* ist hier ein `vec3`, um einen Farbwert zu speichern.

```
layout(location = 0) rayPayloadNV vec3 hitValue;
```

Zuerst werden die Texturkoordinaten für den Zugriff in den G-Buffer analog zu den Shadern für Schatten und Umgebungsverdeckung berechnet, genau wie das initialisieren der Zufallszahlen. Damit werden die Position, die Objekt-ID, die Oberflächennormale und die Texturkoordinaten aus dem G-Buffer gelesen.

```
vec4 originAndID = texture(gbufferPosSampler, inUV);
vec3 normal = normalize(texture(gbufferNormalSampler, inUV).xyz);
vec4 uvLOD = texture(gbufferUVSampler, inUV);

vec3 origin = originAndID.xyz;
int MeshID = int(originAndID.w);
```

Wird in dem aktuell behandelten Pixel keine Geometrie angezeigt, sondern handelt es sich um einen Hintergrundpixel, bricht der Shader ab.

```
if(MeshID == -1) return; //background
```

Dies behebt ein Problem, welches `NaN`-Werte erzeugt, wenn ein Hintergrundpixel behandelt wird, welches nur auf Grafikkarten ohne Ray Tracing Cores auftrat.

Außerdem wird in diesem Shader das Material des aktuellen Objekts benötigt, welches aus dem Material-Buffer geladen wird. Dafür werden erst die *multi draw indirect*-Daten geladen, an welche die Material-ID angehängt ist.

```
PerMeshInfoPBR currentMesh = perMeshInfos.perMesh[MeshID];
MaterialInfoPBR currentMaterial =
  materials[currentMesh.assimpMaterialIndex];
```

Die Einstellungen für die zu verschickenden Strahlen sind hier wie bereits beschrieben. Die Geometrie zählt als undurchsichtig und alle Objekte in der Szene sollen berücksichtigt werden.

```
uint rayFlags = gl_RayFlagsOpaqueNV;
uint cullMask = 0xff;
float tmin = 0.001;
float tmax = 100000.0;
```

Danach werden die Materialparameter *albedo*, *roughness* und *metalness* geladen. Diese kommen entweder aus dem Material-Buffer oder aus der jeweiligen Textur und werden je nachdem, ob das geladene 3D-Modell das *FBX*- oder *glTF*-Format hat, unterschiedlich geladen. Das Script, was im Laufe dieser Arbeit zu dem Zweck geschrieben wurde, den Shader-Compiler *glslc* für jeden Shader im Projekt auszuführen, um die Shader von GLSL zu SPIR-V zu kompilieren, ruft für jeden Shader den Compiler zwei Mal mit jeweils einem anderen `define` auf und kompiliert so zwei verschiedene Versionen jedes Shaders, eine für *FBX*- und eine für *glTF*-Modelle.

Nachdem die *roughness* an dem jeweiligen Pixel feststeht, wird geprüft, ob diese unter dem Schwellwert liegt, falls dieser über die Benutzeroberfläche eingegeben wurde. Ist die *roughness* zu niedrig, wird in das aktuelle hoch oder niedrig aufgelöste Ausgabebild geschrieben und der Shader beendet.

```
if(perFrameInfo.RTReflectionRoughnessThreshold > roughness - 0.01)
{
  if(perFrameInfo.RTUseLowResReflections == 0)
    imageStore(reflectionImage, ivec2(gl_LaunchIDNV.xy),
      vec4(0.0));
  else
    imageStore(reflectionLowResImage, ivec2(gl_LaunchIDNV.xy),
      vec4(0.0));
  return;
}
```

Ansonsten wird der in Abschnitt 2.7 eingeführte Parameter F_0 initialisiert, der bei Nichtmetallen den Wert $0,04$ hat und bei Metallen im Parameter *albedo* steht.

```
vec3 F0 = vec3(0.04);
F0 = mix(F0, albedo, metallic);
```

Außerdem wird der View-Vektor, also der Vektor von der Kamera zum behandelten Punkt in der Szene, berechnet und dieser an der Oberflächennormale reflektiert.

```
vec3 viewVector = normalize(origin - perFrameInfo.cameraPosWorld);
vec3 reflectedViewVector = reflect(viewVector, normal);
```

Der Reflexionswert wird dann in einer Schleife über die Anzahl der Samples, eingegeben über die Benutzeroberfläche, berechnet.

```
vec3 reflectionValue = vec3(0.0f);

for(int i = 0; i < perFrameInfo.RTReflectionSampleCount; i++)
{
  vec3 direction =
    rotateToNormal(sampleHemisphere(importanceSampleGGX(
      vec2(rand(), rand()), roughness)), reflectedViewVector);

  // re-roll if direction is not in hemisphere
  while(dot(direction, normal) < 0.0f)
  {
    direction =
    rotateToNormal(sampleHemisphere(importanceSampleGGX(
      vec2(rand(), rand()), roughness)), reflectedViewVector);
  }

  traceNV(topLevelAS, rayFlags, cullMask,
    0 /*sbtRecordOffset*/, 0 /*sbtRecordStride*/, 0 /*missIndex*/,
    origin, tmin, direction, tmax,
    0 /*payload*/ // X here is location = X of the payload
  );

  vec3 fresnel = fresnelSchlickRoughness(
    max(dot(direction, normal), 0.0), F0, roughness);

  reflectionValue += fresnel * hitValue;
}
reflectionValue /= float(perFrameInfo.RTReflectionSampleCount);
```

In der Schleife wird zuerst die Richtung für den Reflexionsstrahl bestimmt. Die Funktion `importanceSampleGGX(...)` generiert einen Punkt in Kugelkoordinaten abhängig von der *roughness* mit der Wahrscheinlichkeit der in Abschnitt 2.7 vorgestellten *GGX*-Funktion. Diese Art und Weise, die Reflexionen zu berechnen, wird auch von Barré-Brisebois et al. [Bar+19] vorgeschlagen und nimmt als Grundlage, dass dies nur den glänzend spiegelnden Teil des indirekten Lichts approximieren soll und die diffuse indirekte Beleuchtung über andere Verfahren approximiert werden kann. Die Funktion `sampleHemisphere(...)` konvertiert die Kugelkoordinaten in einen dreidimensionalen Richtungsvektor. Die Funktion `rotateToNormal(...)` wird hier dazu genutzt, die berechnete Richtung am reflektierten View-Vektor auszurichten. Die `while(...)`-Schleife dient dazu, einen neuen Vektor zu berechnen, falls der davor generierte Vektor außerhalb der Hemisphäre ist,

also ins Innere der Geometrie zeigt. Dies wird ebenfalls von Barré-Brisebois et al. [Bar+19] empfohlen und führt auch in dieser Arbeit zu Artefakten, wenn es nicht vorgenommen wird. Die `traceNV(...)`-Funktion verschickt den Strahl, welcher bei einem Schnittpunkt den Closest Hit Shader aufruft, in welchem die direkte Beleuchtung und der Schatten am Schnittpunkt berechnet werden. Dieser wird im folgenden Text genauer vorgestellt. Ist der Reflexionswert berechnet, wird dieser noch mit dem Fresnel-Wert, der die „Reflektivität“ der Oberfläche an dem Punkt angibt, gewichtet. Der aufaddierte Wert für jeden Sample wird dann durch die Anzahl der Samples geteilt. Analog zu den anderen Ray Tracing Shadern wird der Reflexionswert mit den Werten der vorherigen Frames akkumuliert gespeichert, nur das hier eine Fallunterscheidung, welche Textur (hoch oder niedrig aufgelöst) beschrieben wird, durchgeführt wird.

Der Closest Hit Shader `rtreflectionsPBR.rchit` hat Zugriff auf zwei *ray payloads*, eine für den Strahl, aus dem der Shader aufgerufen wurde und eine für die verschickten Strahlen.

```
layout(location = 0) rayPayloadInNV vec3 hitValue;
layout(location = 1) rayPayloadNV int rtSecondaryShadow;
hitAttributeNV vec2 attribs;
```

Die mit `hitAttributeNV` deklarierte Variable `attribs` beinhaltet die baryzentrischen Koordinaten des Schnittpunkts im Dreieck, in dem der Schnittpunkt gefunden wurde. Diese sind in zwei Werten kodiert und der fehlende Wert wird auf die folgende Weise berechnet:

```
vec3 barycentrics =
  vec3(1.0 - attribs.x - attribs.y, attribs.x, attribs.y);
```

Danach müssen die Vertex-Daten aus dem Vertex-Buffer geladen werden. Wie in Abbildung 14 dargestellt ist, wird erst der Offset in den Index-Buffer aus dem Offset-Buffer geladen, mit dem dann auf den Index für jeden Vertex im getroffenen Dreieck zugegriffen wird. Zusammen mit diesen Indizes und dem Offset in den Vertex-Buffer aus dem Offset-Buffer wird dann auf die Vertex-Daten zugegriffen.

```
OffsetInfo currentOffset =
  offsetInfos.offsets[gl_InstanceCustomIndexNV];
uint index0 = indexInfos.indices[currentOffset.m_ibOffset +
  (3 * gl_PrimitiveID + 0)];
uint index1 = indexInfos.indices[currentOffset.m_ibOffset +
  (3 * gl_PrimitiveID + 1)];
uint index2 = indexInfos.indices[currentOffset.m_ibOffset +
  (3 * gl_PrimitiveID + 2)];
VertexInfo vertex0 =
  vertexInfos.vertices[currentOffset.m_vbOffset + index0];
VertexInfo vertex1 =
  vertexInfos.vertices[currentOffset.m_vbOffset + index1];
VertexInfo vertex2 =
  vertexInfos.vertices[currentOffset.m_vbOffset + index2];
```

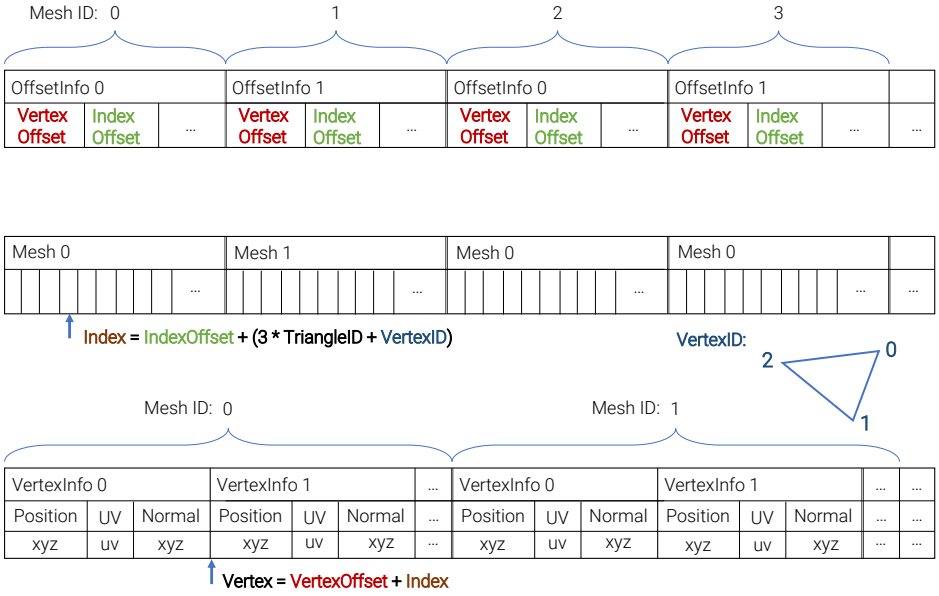


Abbildung 14: Das Zugriffschema für die Vertex-Daten im Closest Hit Shader für die Reflexionen.

Die built-in-Variable `gl_InstanceCustomIndexNV` ist hierbei die beim Erstellen der Beschleunigungsdatenstruktur in den Instanzen angegebene `instanceId` und gibt in diesem Fall die ID des getroffenen Objekts, also den Index in den Offset-Buffer, an. Die built-in-Variable `gl_PrimitiveID` gibt den Index des getroffenen Dreiecks in dem Objekt an. Für die Beleuchtung und Texturierung werden die Texturkoordinaten, die Normale, sowie die Position des getroffenen Punktes benötigt. Die ersten beiden werden aus den Vertex-Attributen mit den baryzentrischen Koordinaten interpoliert, die Position wird aus den builtin-Variablen, die die Ausgangsposition, die Richtung und die Schrittweite des Strahls bis zum Schnittpunkt angeben, berechnet:

```
const vec2 uv =
  barycentrics.x * vertex0.uv +
  barycentrics.y * vertex1.uv +
  barycentrics.z * vertex2.uv;

const vec3 N =
  normalize(
    barycentrics.x * vertex0.normal +
    barycentrics.y * vertex1.normal +
    barycentrics.z * vertex2.normal);

const vec3 WorldPos =
  gl_WorldRayOriginNV + gl_WorldRayDirectionNV * gl_HitTNV;
```

Für die Strahlen zur Lichtquelle zur Schattenberechnung werden hier die Einstellungen für die Strahlen so wie auch bei der Berechnung der Schatten in der ganzen Szene gewählt.

```
uint rayFlags = gl_RayFlagsTerminateOnFirstHitNV |
                gl_RayFlagsOpaqueNV |
                gl_RayFlagsSkipClosestHitShaderNV;;
uint cullMask = 0xff;
```

Danach werden die Materialinformationen aus dem Material-Buffer und aus den Texturen geladen. Dies geschieht wie zuvor im Ray Generation Shader. Ist dies geschehen, wird die direkte Beleuchtung für den Schnittpunkt berechnet, was analog zu der Beleuchtungsberechnung im Beleuchtungs-Shader durchgeführt wird. Diese wird in den nachfolgenden Abschnitten, in denen der finale Beleuchtungs-Shader erklärt wird, genau vorgestellt und wird deshalb an dieser Stelle nicht im Detail erklärt. Der Unterschied zur Beleuchtung im Beleuchtungs-Shader sind die Schattenstrahlen, welche hier zu jeder Lichtquelle geschickt werden. Im Gegensatz zu der Schattenberechnung werden die Lichtquellen nicht als flächige Lichtquellen abgetastet, es wird ein einzelner Strahl zum Zentrum der Lichtquelle verwendet, um eine einfache Verschattung zu approximieren. Dies ist eine approximativere Berechnung, bei der harte Schattenkanten entstehen. Diese sind in der Praxis in den Reflexionen nicht zu erkennen.

Wie auch bei der Berechnung der Schatten für die gesamte Szene wird kein Hit Shader eingesetzt, sondern lediglich ein Miss Shader, um zu melden, wenn die Lichtquelle vom Strahl erreicht wurde ohne Szenengeometrie zu schneiden, weswegen die *ray payload*, hier `rtSecondaryShadow`, vor jedem `traceNV`-Auruf auf 0 gesetzt werden muss. Im folgenden Quellcode-Ausschnitt ist exemplarisch der `traceNV`-Aufruf für die Punktlichtquellen vorgestellt.

```
rtSecondaryShadow = 0;
traceNV(topLevelAS, rayFlags, cullMask,
  0 /*sbtRecordOffset*/, 0 /*sbtRecordStride*/, 1 /*missIndex*/,
  WorldPos, 0.001, L, length(currentLight.position - WorldPos),
  1 /*payload*/ // X here is location = X of the payload
);
```

Der `missIndex` ist auf 1 gesetzt, da der Miss Shader, der in der *shader binding table* an zweiter Stelle steht, genutzt werden soll. An Index 0 steht der Miss Shader, der für den ersten Strahl für die Reflexionen zum Einsatz kommt. Ebenso ist der Index der *ray payload* auf 1 gesetzt, da die *ray payload* mit `location = 1` genutzt wird. Der Miss Shader `rtreflectionsSecondaryShadow.rmiss` ist wie folgt:

```
layout(location = 1) rayPayloadInNV int rtSecondaryShadow;

void main()
{
  rtSecondaryShadow = 1;
}
```

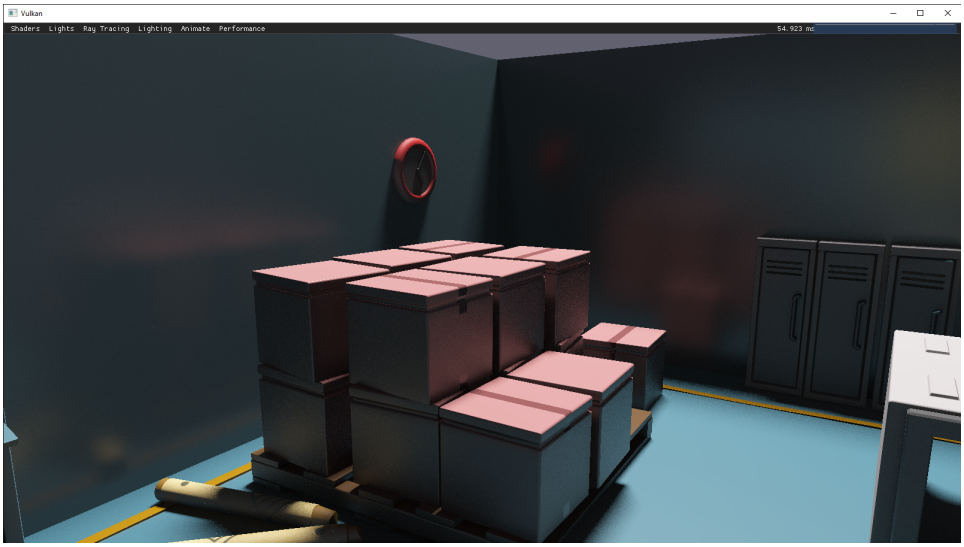


Abbildung 15: Glänzende Reflexionen mit akkumulierten Samples.

Der zurückgegebene Schattenwert ist folglich entweder 0 für Schatten oder 1 für keinen Schatten und wird dementsprechend in die direkte Beleuchtung von der Lichtquelle, von der die Verschattung so ermittelt wurde, eingerechnet. Dies geschieht für alle drei Lichtquellentypen in jeweils einer Schleife über alle Lichtquellen des Typs. Der so berechnete Beleuchtungswert wird noch mit dem diffusen Anteil der direkten Beleuchtung, also dem Faktor des diffusen Anteils, berechnet über den Fresnel-Term, multipliziert mit der ambienten Beleuchtung und mit dem *albedo*-Wert addiert und stellt dann die finale Reflexionsfarbe dar, welche in die *ray payload* des einfallenden Strahls geschrieben wird.

Falls ein Reflexionsstrahl keine Szenengeometrie trifft, wird der Reflexionswert im Miss Shader `rtreflections.rmiss` auf die Hintergrundfarbe gesetzt.

```
#version 460
#extension GL_NV_ray_tracing : require

layout(location = 0) rayPayloadInNV vec3 hitValue;

void main()
{
  hitValue = vec3(0.007, 0.007, 0.01);
}
```

In Abbildung 15 sind glänzende Reflexionen in der in Abschnitt 6.1 beschriebenen Szene abgebildet. Dabei ist deutlich zu erkennen, dass in den Reflexionen Teile der Szene zu sehen sind, die nicht im eigentlichen Bild abzüglich der Reflexionen vorhanden sind.

Versuch der Beschleunigung der Reflexionsberechnung mit Subgroup Operations Als Versuch der Beschleunigung der Reflexionsberechnung durch Ray Tracing durch weglassen nicht notwendiger Operationen wurde ein Ansatz nach dem von Mallett und Yuksel [MY18] beschriebenen Prinzip getestet. Die von Mallett und Yuksel [MY18] vorgestellte Methode ist für Rasterisierung gedacht und funktioniert so, dass erst nur wenige Pixel gerendert werden. Die Abweichung der Farbwerte dieser Pixel bestimmt, ob ein Pixel, der räumlich mittig zu diesen bereits berechneten Pixeln gelegen ist, als Farbe den Mittelwert dieser Pixel annimmt oder selbst berechnet wird. Dies geschieht mehrfach, zuerst mit wenigen, weiter voneinander entfernten Pixeln, dann mit dem Ergebnis nochmals, bis alle Pixel gefüllt sind. Die Idee war nun, dies auch für die Berechnung der Reflexionen zu nutzen. So würden an Stellen, an denen alle nah aneinander gelegenen Pixel einen ähnlichen Reflexionswert haben, wie zum Beispiel auf einer Oberfläche, in der sich lediglich ein einfarbiger Hintergrund oder beispielsweise ein blauer Himmel spiegelt, weniger Strahlen verschickt. Dies ist insbesondere bei den Reflexionen hilfreich für die Performance, da die Berechnung der Reflexionen der zeitaufwendigste Teil des Gesamtbilds ist, wie in Abschnitt 6.1 beschrieben ist.

Da die Implementierung von Mallett und Yuksel [MY18] laut deren Beschreibung in einem einzelnen Compute Shader abläuft, aus einem Compute Shader aber keine Strahlen mittels RTX-API verschickt werden können, war eine Implementierung mittels *subgroup operations* in Vulkan geplant. Dieses Feature erlaubt es, innerhalb von durch die Hardware bestimmten Gruppen von Threads, Daten zu teilen und arithmetische Operationen auf Daten innerhalb der Gruppe auszuführen. All dies geschieht ohne *shared memory*, welches in Ray Generation Shadern nicht zur Verfügung steht. So war die Idee, den Ergebniswert der Reflexion innerhalb der Thread-Gruppe zu vergleichen und nur falls nötig weitere Strahlen zu verschicken. Durch eine Auslassung in der Spezifikation der Shadersprache GLSL sind die dafür benötigten built-in-Variablen `gl_SubgroupSize` und `gl_SubgroupInvocationID` in Ray Tracing Shadern nicht verfügbar, trotz der vom Treiber gemeldeten Kompatibilität von Ray Tracing Shadern und Subgroup Operations und der Verfügbarkeit der Subgroup-Funktionen. Dies wurde im Rahmen der Arbeit als Fehler gemeldet und auch als solcher anerkannt, im Laufe der Arbeit jedoch nicht behoben[28][29]. Die Variablen sind für das Identifizieren des Threads innerhalb der Gruppe unerlässlich und die Nichtverfügbarkeit hat eine erfolgreiche Implementierung des vorgestellten Ansatzes verhindert. Damit wurde auch ein Test, ob die Performance auf diese Weise tatsächlich verbessert werden kann, da die Eigenheiten des *scheduling* des Hardware-Ray

[28] `https://github.com/KhronosGroup/GLSL/issues/65`, letzter Abruf: 5. Juni 2019

[29] `https://github.com/KhronosGroup/glslang/issues/1735`, letzter Abruf: 5. Juni 2019

Tracings, also das Ansteuern der Ray Tracing Hardware aus den einzelnen Threads und dessen Umgang mit wartenden Threads, nicht bekannt sind, verhindert.

Commands für die finale Beleuchtung Für den sekundären *command buffer* für die finale Beleuchtung und das Zusammensetzen der Ray Tracing-Ergebnisse muss wie auch schon beim Rendering in den G-Buffer der Renderpass im primären *command buffer* begonnen werden. Dabei wird die Ausgabetextur, welche auf dem Bildschirm angezeigt wird, und der Tiefenpuffer zurückgesetzt (*clear*). Die Ausgabe in die richtige Textur des *triple buffering* wird über das Binden des richtigen Framebuffers sichergestellt.

```
vk::ClearValue clearValue2(
  std::array<float, 4>{ 0.0f, 0.0f, 0.0f, 1.0f });
std::array<vk::ClearValue, 2> clearColors2 =
  { clearValue2, vk::ClearDepthStencilValue{1.0f, 0} };

vk::RenderPassBeginInfo renderpassInfo2(
  m_fullscreenLightingRenderpass,
  m_swapChainFramebuffers.at(currentImage),
  { {0, 0}, m_context.getSwapChainExtent() },
  static_cast<uint32_t>(clearColors2.size()),
  clearColors2.data());

m_commandBuffers.at(currentImage).beginRenderPass(
  renderpassInfo2, vk::SubpassContents::eSecondaryCommandBuffers);

m_commandBuffers.at(currentImage).executeCommands(
  m_fullscreenLightingSecondaryCommandBuffers.at(currentImage));

m_commandBuffers.at(currentImage).endRenderPass();
m_commandBuffers.at(currentImage).end();
```

Der hier verwendete sekundäre *command buffer* ist statisch und wurde deshalb schon in der Initialisierungsphase aufgenommen. Dieser muss den laufenden Renderpass referenzieren. In dem *command buffer* selbst wird nur ein Zeitstempel geschrieben, die Pipeline und die Descriptor Sets gebunden, dann der *draw call* aufgerufen und ein weiterer Zeitstempel geschrieben. Die gebundenen Descriptor Sets sind folgende: Das speziell für diesen Renderpass angelegte Descriptor Set mit allen pro-Mesh-Informationen, Materialien, Texturen und den G-Buffer-Texturen, das Descriptor Set mit den Daten der Lichtquellen und das Descriptor Set, in dem die Ray Tracing-Ergebnisbilder als `vk::CombinedImageSampler`, um daraus als Textur zu lesen, referenziert sind. Das *full screen triangle*, also ein Dreieck, welches den ganzen Bildschirm ausfüllt, wird hier nicht als Vertex-Buffer gebunden, sondern im Vertex Shader generiert. Der dazugehörige *draw call* lautet wie folgt, 3 ist die Anzahl der Vertices, 1 die Anzahl der Instanzen:

```
m_fullscreenLightingSecondaryCommandBuffers.at(i).draw(
  3, 1, 0, 0);
```

Shader für die finale Beleuchtung Im Vertex-Shader, der in der Datei `fullscreen.vert` gespeichert ist, werden je nach `gl_VertexIndex` die Ausgabepunkte und die dazugehörigen Texturkoordinaten für den Zugriff auf die G-Buffer-Texturen berechnet[30].

```
#version 460
#extension GL_ARB_separate_shader_objects : enable

layout (location = 0) out vec2 outUV;

void main()
{
  outUV = vec2((gl_VertexIndex << 1) & 2, gl_VertexIndex & 2);
  gl_Position = vec4(outUV * 2.0f + -1.0f, 0.0f, 1.0f);
}
```

Im Fragment Shader `fullscreenLightingPBR_RT.frag` wird zuerst die Position und die Objekt-ID aus dem G-Buffer geladen.

```
vec4 posAndID = texture(gbufferPositionSampler, inUV);

vec3 WorldPos = posAndID.xyz;

int drawID = int(posAndID.w);
```

Handelt es sich um einen Hintergrundpixel, wird die Hintergrundfarbe mit der Belichtung verrechnet und das *tone mapping* wie später beschrieben berechnet, das Ergebnis als Ausgabewert geschrieben und die Shader-Ausführung für diesen Pixel vorzeitig beendet.

Nach dieser Fallunterscheidung werden in Analogie zur Beleuchtung im Closest Hit Shader für die Reflexionen Oberflächennormale und Texturkoordinaten aus dem G-Buffer geladen und die Materialinformationen aus dem Buffer geladen und die Materialparameter *roughness*, *metalness* und *albedo* in Abhängigkeit des Formats des geladenen 3D-Modells geladen, wie bei dem Closest Hit Shader für die Reflexionen bereits beschrieben.

Der Umgebungsverdeckungswert wird aus der Textur geladen, in die dieser beim Ray Tracing geschrieben wird:

```
float ao = texture(rtaoImage, inUV).x;
```

Die restlichen Parameter für die Berechnung der direkten Beleuchtung werden wie bei der Beleuchtung im Closeset Hit Shader berechnet:

```
// viewing vector
vec3 V = normalize(matrices.cameraPos.xyz - WorldPos);

vec3 F0 = vec3(0.04);
F0 = mix(F0, albedo, metallic);
```

[30] https://www.saschawillems.de/blog/2016/08/13/vulkan-tutorial-on-rendering-a-fullscreen-quad-without-buffers/, letzter Abruf: 5. Juni 2019

Danach folgen drei Schleifen, eine für jeden Lichtquellen-Typ, in denen die jeweilige Beleuchtung berechnet und der Schattenwert für diese Lichtquelle geladen wird. Im folgenden Quellcode-Ausschnitt wird die Schleife für die Punktlichtquellen exemplarisch gezeigt:

```
vec3 Lo = vec3(0.0);
for(int i = 0; i < pointLights.length(); ++i)
{
  float pointShadow = texture(shadowPointImage, vec3(inUV, i)).x;

  // get light parameters
  PBRPointLight currentLight = pointLights[i];

  // light vector
  vec3 L = normalize(currentLight.position - WorldPos);

  // halfway vector
  vec3 H = normalize(V + L);

  // calculate per-light radiance
  float distance    = length(currentLight.position - WorldPos);
  float attenuation = 1.0 / (distance * distance);
  vec3 radiance     = currentLight.intensity * attenuation;

  // calculate the parts of the Cook-Torrance BRDF
  float NDF = DistributionGGX(N, H, roughness);
  float G   = GeometrySmith(N, V, L, roughness);
  vec3 F    = fresnelSchlick(max(dot(H, V), 0.0), F0);

  vec3 kS = F;
  vec3 kD = vec3(1.0) - kS;
  kD *= 1.0 - metallic;

  // Cook-Torrance specular BRDF term: DFG / 4(w0 . n)(wi . n)
  vec3 nominator    = NDF * G * F;
  float denominator =
    4 * max(dot(N, V), 0.0) * max(dot(N, L), 0.0) + 0.001;
  vec3 specular     = nominator / denominator;

  // add to outgoing radiance Lo
  float NdotL = max(dot(N, L), 0.0);
  Lo += (kD * albedo / PI + specular) *
    radiance * NdotL * pointShadow;
}
```

Zuerst wird der Schattenwert aus dem zur Lichtquelle korrespondierendem Array-Layer der Schatten-Arraytextur geladen. Dann werden die Lichtquelleninformationen für die aktuell behandelte Lichtquelle aus dem Buffer geladen. Der Lichtvektor und der *halfway vector* werden berechnet. Die Lichtintensität wird mit dem *inverse-square law*, also mit $\frac{1}{d^2}$, wobei d der Abstand zur Lichtquelle ist, abgeschwächt. Dies passiert bei der hier nicht gezeigten gerichteten Lichtquelle nicht. Danach werden mit den in Abschnitt 2.7 vorgestellten Funk-

tionen die einzelnen Komponenten des *Cook-Torrance*-Beleuchtungsmodells berechnet. Der diffuse Anteil `kD` wird mit dem Fresnel-Term abhängig vom Material berechnet und der `specular`-Teil des Beleuchtungsmodells wird aus den einzelnen Parametern ausgerechnet. In der letzten Zeile der Schleife wird die gesamte Beleuchtung berechnet und mit dem Schattenwert gewichtet. Dabei ist `kD * albedo / PI` der diffuse Teil. Bei der Beleuchtungsberechnung für die Spotlights wird zusätzlich der Lichtabfall zum Rand des Lichtkegels mit eingerechnet:

```
// spotlight intensity (falloff to sides of spot)
float theta = dot(L, normalize(-currentLight.direction));
float epsilon = currentLight.cutoff - currentLight.outerCutoff;
radiance *=
  clamp((theta - currentLight.outerCutoff) / epsilon, 0.0, 1.0);
```

Sind alle drei Schleifen über die Lichtquellen beendet, wird noch der ambiente Term und der Reflexionswert mit in die Beleuchtung einbezogen. Der Reflexionswert wird, je nachdem ob die hoch- oder niedrig aufgelösten Reflexionen ausgewählt sind, aus der jeweiligen Textur gelesen.

```
vec3 F =
  fresnelSchlickRoughness(max(dot(V, N), 0.0f), F0, roughness);
vec3 kS = F;
vec3 kD = vec3(1.0) - kS;
kD *= 1.0 - metallic;

vec3 reflectionColor;
if(matrices.useLowResReflections == 0)
  reflectionColor = texture(reflectionImage, inUV).xyz;
else
  reflectionColor = texture(reflectionLowResImage, inUV).xyz;

vec3 backgroundAmbient = vec3(0.003f);
vec3 color = (backgroundAmbient * (albedo / PI) *
  kD * ao + reflectionColor) + Lo;
```

Danach muss noch das *tone mapping* und die Gammakorrektur durchgeführt werden. Gamma ist hier auf den Wert $2,2$ festgelegt, die Belichtung (in der Variable `matrices.exposure`) wird über die Benutzeroberfläche eingegeben.

```
color = vec3(1.0) - exp(-color * matrices.exposure);
color = pow(color, vec3(1.0/2.2));
outColor = vec4(color, 1.0f);
```

Die Variable `outColor` ist hier die Ausgabe in das auf dem Bildschirm angezeigte Bild. Über dieses wird später noch wie in Abschnitt 5.4.2 beschrieben die Benutzeroberfläche gelegt. Damit ist die Berechnung des Ausgabebilds unter Zunahme aller Zwischenergebnisse abgeschlossen.

5.5 Die Zufallszahlen

Die Zufallszahlen werden wie in Abschnitt 5.3.15 beschrieben initialisiert und in eine dafür vorgesehene Textur in dreifacher Ausführung geschrieben. Bei jeder Verwendung, also den Ray Generation Shadern für Schatten, Umgebungsverdeckung und Reflexionen, wird der zum jeweiligen Pixel gehörige Wert als `seed` aus der Textur geladen. Bei jeder verwendeten Zufallszahl wird ein wie von Howes und Thomas [HT07] beschriebener Schritt eines linearen Kongruenzgenerators kombiniert mit einem Tausworthe-Step durchgeführt. Normalerweise wird am Ende jedes Shaders der dabei aus dem alten `seed` entstandene neue `seed`-Wert wieder in die Textur zurückgeschrieben. Da es genügt, einen Schritt pro Frame zu machen, da es nur darum geht, nicht die Zufallszahl aus demselben `seed` für denselben Verwendungszweck mehrfach zu verwenden, aber nicht, diese Zufallszahl für verschiedene Einsatzzwecke zu benutzen, wird nur im ersten Shader, der die Zufallszahlen benutzt, in diesem Fall der Ray Generation Shader für die Schatten, der neue `seed` in die Textur geschrieben. Ein reiner Lesezugriff in den anderen Shadern vereinfacht außerdem die Synchronisierung, die in Vulkan manuell erfolgen muss.

6 Evaluation

In den folgenden Abschnitten werden die Ergebnisse vorgestellt und die gewonnen Erkenntnisse und subjektiven Eindrücke über die RTX-API, deren Einsatzzweck, Ausgereiftheit und Umfang beschrieben. Außerdem wird ein kurzer Performance-Überblick gegeben.

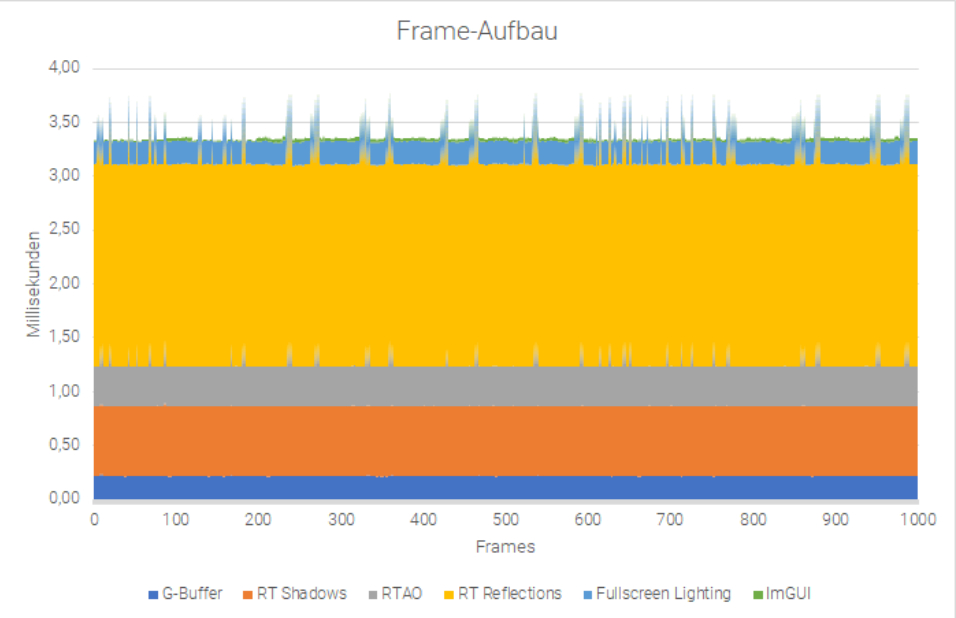


Abbildung 16: Dauer der einzelnen Schritte innerhalb eines Frames zusammengesetzt für 1000 Frames

6.1 Messungen

Aufgrund der in Abschnitt 6.3.1 beschriebenen Hardwareprobleme wurden Messungen nur bei einer Szene durchgeführt, dem 3D-Modell *PICA PICA – Mini Diorama 01*[31], zur Verfügung gestellt im Rahmen der in Abschnitt 3.1 vorgestellten Publikation. Diese besteht aus insgesamt 76274 Dreiecken in 170 einzelnen Meshes und wird als *glTF*-Format geladen.

Die Messungen sind mit einem Strahl pro Pixel für Reflexionen und Umgebungsverdeckung und einem Strahl pro Lichtquelle für Schatten mit einer Punkt-, einer Spot- und einer gerichteten Lichtquelle bei einer Auflösung von 1600×900 Pixeln durchgeführt worden. Bei der verwendeten Hardware handelt es sich um eine *NVIDIA RTX 2080 Ti*-Grafikkarte und einem *Intel Core i9-9900K* als Prozessor.

[31] `https://sketchfab.com/3d-models/pica-pica-mini-diorama-01-45e26a4ea7874c15b91bd659e656e30d`, letzter Abruf: 5. Juni 2019

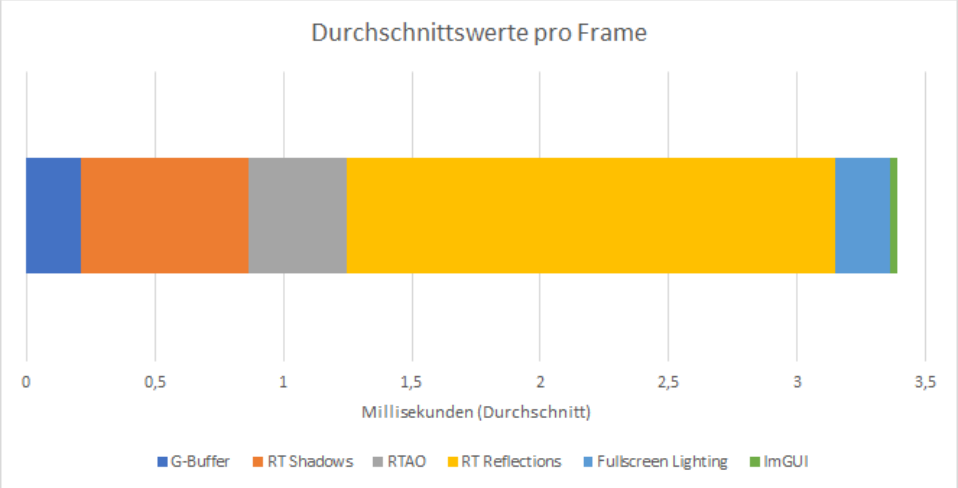


Abbildung 17: Durchschnittswerte der Teilschritte über 1000 Frames

Durchschnittswerte		
Schritt	Zeit (ms)	Anteil
G-Buffer	0,213	6,3 %
Schatten	0,653	19,3 %
Umgebungsverdeckung	0,381	11,3 %
Reflexionen	1,899	56,1 %
Beleuchtung, Komposition	0,214	6,3 %
Benutzeroberfläche	0,024	0,7 %
Gesamt	3,386	100 %

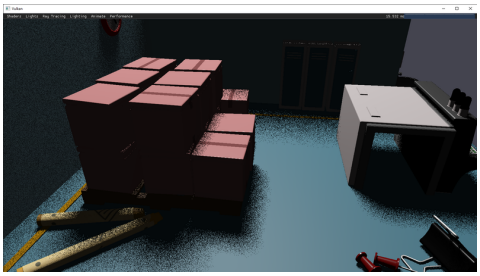
Tabelle 1: Durchschnittswerte der Teilschritte über 1000 Frames und deren Anteil am Gesamtwert

In Abbildung 16 sind die Ergebnisse der einzelnen im Implementationsteil der Arbeit erwähnten `Timer` dargestellt. Dabei wurden in insgesamt 1000 aufeinanderfolgenden Frames die Zeiten gemessen, welche auf der Grafikkarte für die einzelnen im Diagramm gezeigten Schritte gebraucht werden. Vertikal zusammengesetzt ergibt sich so eine Zeit für jeden Frame, in der die einzelnen Teilschritte farblich voneinander abgehoben sind.

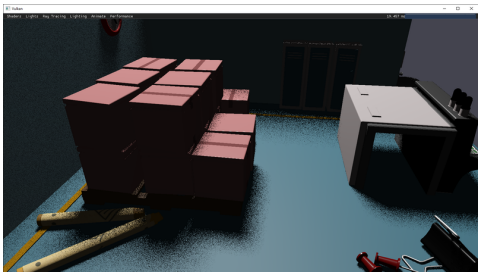
Zusammen mit der Tabelle 1 und der dazugehörigen Abbildung 17 lässt sich erkennen, dass die Berechnung der Reflexionen im Verhältnis zu den anderen Schritten den deutlich größten Zeitraum einnimmt. Dies bedeutet, dass hier das meiste Optimierungspotenzial vorhanden ist.

6.2 Präsentation der Ergebnisse

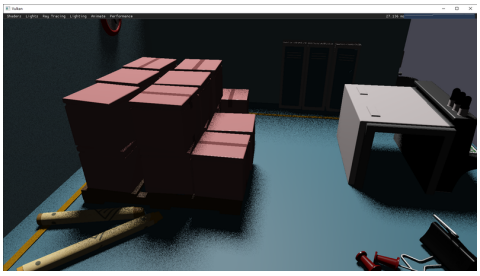
In den folgenden Abschnitten werden die Ergebnisse der einzelnen Rendering-Schritte und das Gesamtergebnis in Bildern gezeigt und erklärt. Die Szene, in der alle Bilder aufgenommen sind, wird in Abschnitt 6.1 vorgestellt.



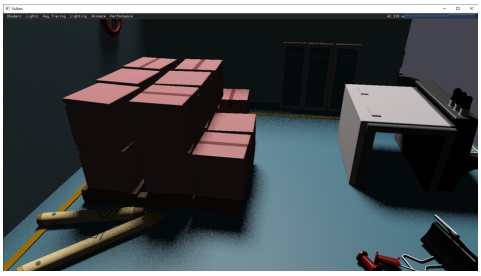
(a) 1 Sample pro Pixel



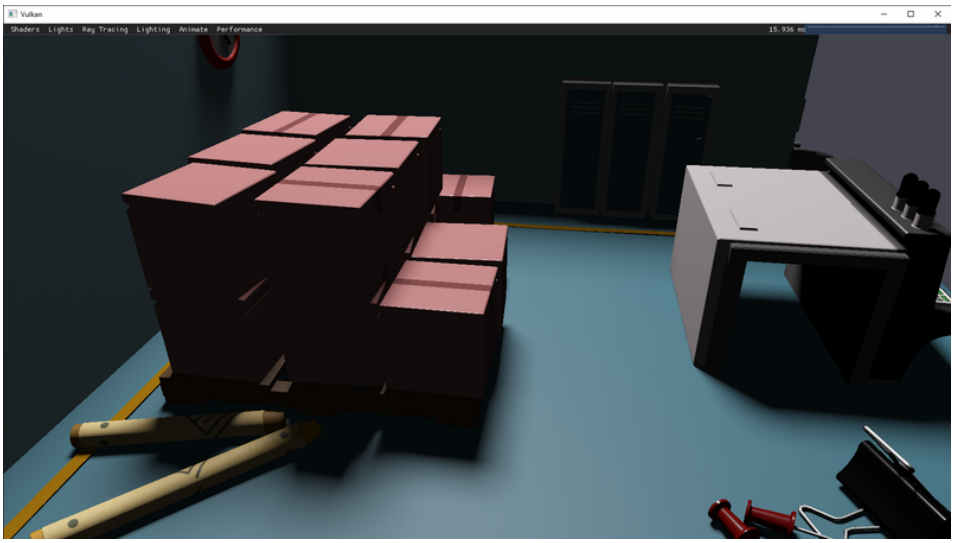
(b) 2 Samples pro Pixel



(c) 4 Samples pro Pixel



(d) 8 Samples pro Pixel



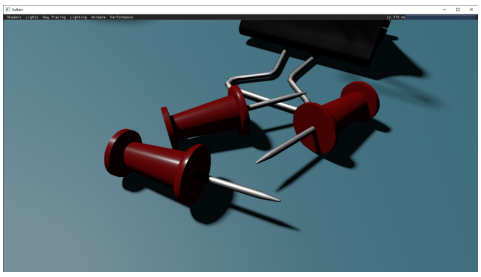
(e) 1 Sample pro Pixel, akkumuliert

Abbildung 18: Schatten einer Punktlichtquelle mit Radius 15 mit verschiedener Anzahl der Samples pro Pixel. Umgebungsverdeckung und Reflexionen sind ausgeschaltet, um kein zusätzliches Rauschen hinzuzufügen.

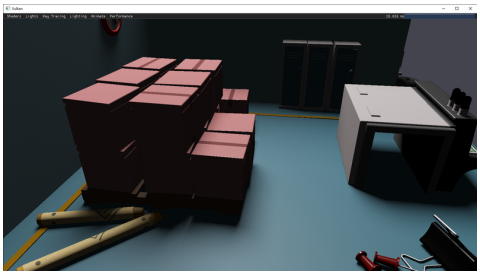
6.2.1 Schatten

In Abbildung 18 ist ein Vergleich der Anzahl der Samples für die Schattenberechnung gezeigt. Hierbei ist wie erwartet zu sehen, dass das Rauschen mit der Anzahl der Samples abnimmt. Ein Vergleich der Ergebnisse bei verschiedenen Radien der Lichtquelle ist in Abbildung 19 gegeben. Dabei ist zu sehen, dass die Berechnung der Schatten durch Ray Tracing für flächige Lichtquellen flexibel möglich ist und verschiedene Lichtquellen-Parameter entsprechend umgesetzt werden. Das Erhöhen des Radius einer Lichtquelle führt zu einer stärkeren Streuung der Schattenstrahlen und damit auch zu mehr Rauschen und einer längeren Zeit, bis sich das Rauschen durch Akkumulieren der vorherigen Frames ausgeglichen hat. Weitere Bilder der Schatten sind in Abbildung 11 zu sehen.

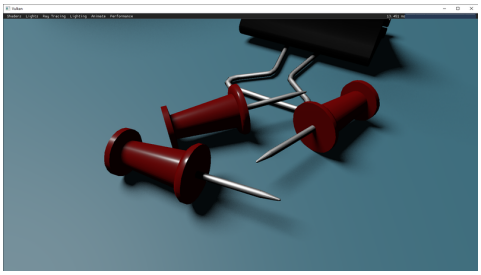
(a) Lichtquellen-Radius 5, akkumuliert



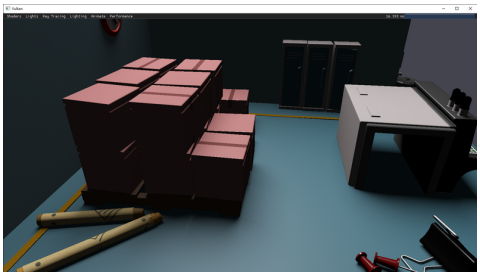
(b) Lichtquellen-Radius 5, akkumuliert



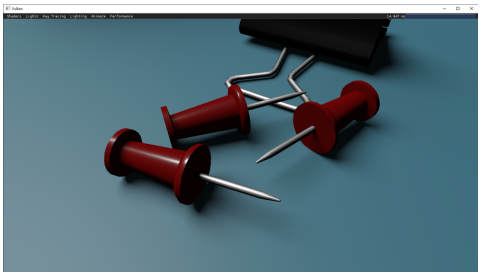
(c) Lichtquellen-Radius 25, akkumuliert



(d) Lichtquellen-Radius 25, akkumuliert

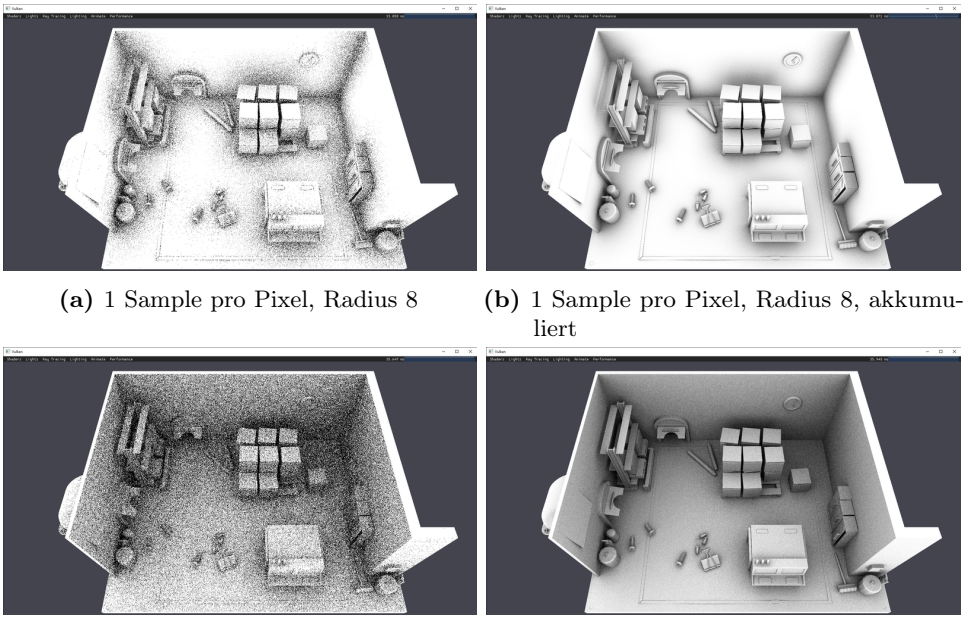


(e) Lichtquellen-Radius 40, akkumuliert



(f) Lichtquellen-Radius 40, akkumuliert

Abbildung 19: Schatten einer Punktlichtquelle mit dem Vergleich verschiedener Radien. Umgebungsverdeckung und Reflexionen sind ausgeschaltet, um kein zusätzliches Rauschen hinzuzufügen.



(a) 1 Sample pro Pixel, Radius 8

(b) 1 Sample pro Pixel, Radius 8, akkumuliert

(c) 1 Sample pro Pixel, Radius 100

(d) 1 Sample pro Pixel, Radius 100, akkumuliert

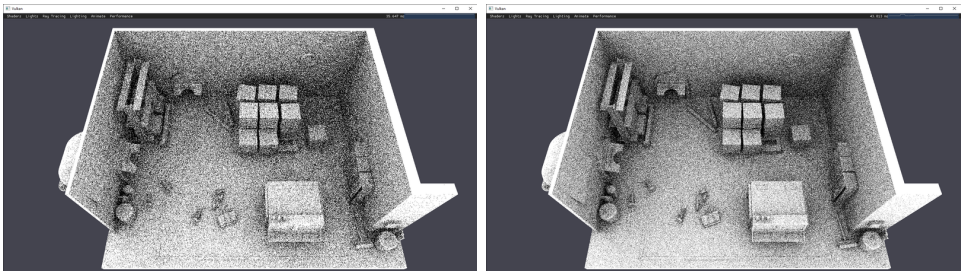
Abbildung 20: Umgebungsverdeckung mit einem Radius-Parameter von 8 und 100, nicht akkumuliert und akkumuliert als Vergleich.

6.2.2 Umgebungsverdeckung

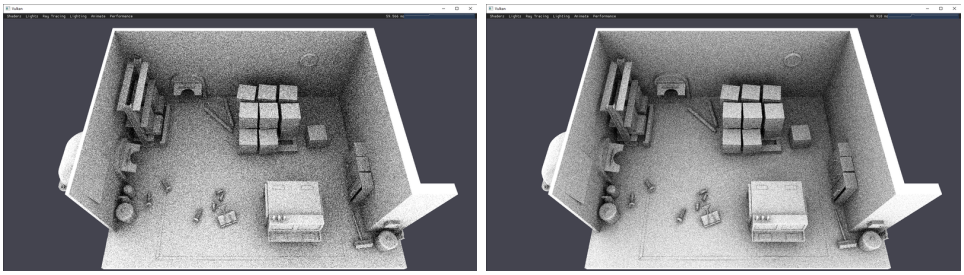
In Abbildung 20 wird der Einfluss des Radius-Parameters, welcher die maximale Strahl-Länge angibt, mit der nach Verdeckungen in der Umgebung gesucht wird, abgebildet. Bei den akkumulierten Bildern präsentiert sich bei dem höheren Radius das visuell ansprechendere Ergebnis. Es wird weniger eine lokale Verdeckung berechnet, sondern die ganze Szene mit einbezogen. Allerdings hat bei den nicht-akkumulierten Bildern das Bild mit dem geringeren Radius weniger auffällige Rausch-Artefakte. Die flexible Ray Tracing-Implementierung lässt das Einstellen der Parameter für das je nach Anwendungsfall gewünschte Ergebnis zu.

Der Einfluss der Anzahl der Samples pro Pixel auf das Endergebnis ist in Abbildung 21 zu erkennen. Wie erwartet nehmen die Rausch-Artefakte bei steigender Anzahl von Samples ab. Durch die Einstellung des Parameters kann die visuelle Qualität in Abhängigkeit von der Performance skaliert werden.

In Abbildung 22 ist ein Vergleich des Gesamtergebnisses mit und ohne Umgebungsverdeckung gezeigt. Das Bild ohne Umgebungsverdeckung sieht im Vergleich vor allem in Ecken heller aus, was das Bild mit Umgebungsverdeckung plastischer und realistischer erscheinen lässt. Eine weitere Detailaufnahme der Umgebungsverdeckung ist in Abbildung 12 zu sehen.



(a) 1 Sample pro Pixel, Radius 100 **(b)** 2 Samples pro Pixel, Radius 100



(c) 4 Samples pro Pixel, Radius 100 **(d)** 8 Samples es pro Pixel, Radius 100

Abbildung 21: Umgebungsverdeckung mit einem Radius-Parameter von 100, nicht akkumuliert, mit verschiedenen Anzahl von Samples pro Pixel.

(a) Gesamtergebnis inklusive Umgebungsverdeckung

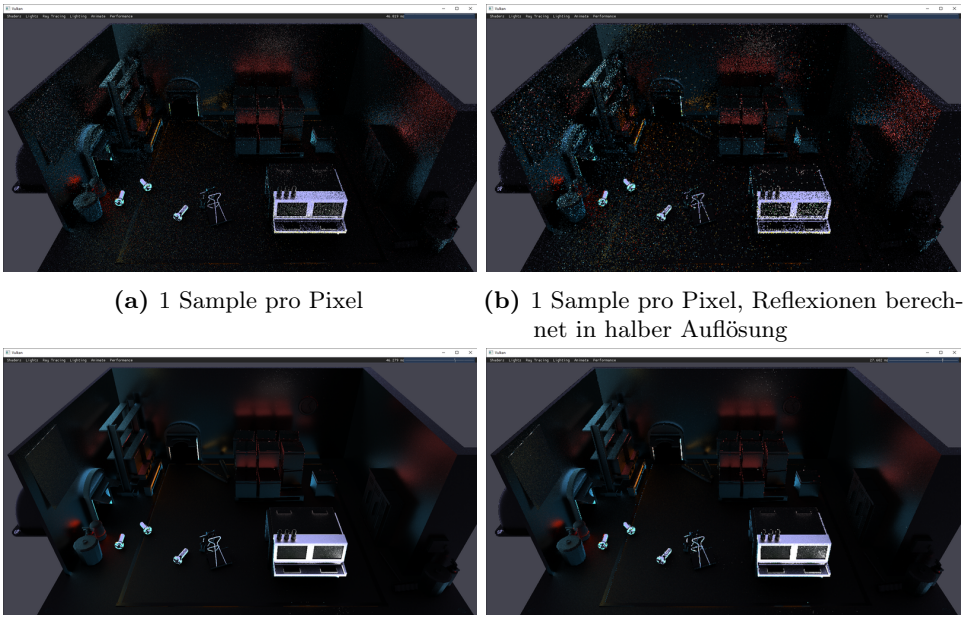
(b) Gesamtergebnis ohne Umgebungsverdeckung

Abbildung 22: Gesamtergebnis mit und ohne Umgebungsverdeckung als Vergleich

6.2.3 Reflexionen

In Abbildung 23 sind die Ergebnisse der Reflexionsberechnung gezeigt. Der Vergleich zwischen der Berechnung in halber und voller Auflösung wird insbesondere bei den Bildern ohne Akkumulation der vorherigen Frames deutlich. Ist das Ergebnis erst einmal akkumuliert, sind die eigentlichen glänzenden Reflexionen nur unauffällig gröber, allerdings sind die Kanten noch immer sehr deutlich niedriger aufgelöst. Insgesamt zeigt sich bei den nicht-akkumulierten Bildern das Rauschen sehr deutlich. Auch hier nimmt das Rauschen wie in Abbildung 24 gezeigt mit der Anzahl der Samples pro Pixel deutlich ab, selbst wenn einige helle Punkte, die auch wie beispielsweise von Pharr [Pha19a] *fireflies* genannt, trotz mehreren Samples pro Pixel deutlich erkennbar bleiben.

In Abbildung 25 ist der Unterschied zwischen den Reflexionen bei hoher und niedriger Auflösung am Gesamtergebnis zu sehen. Beide Bilder sind akkumuliert und auch hier wird der Unterschied zwischen den Auflösungen bei der Reflexionsberechnung insbesondere an den Kanten deutlich. Der Unterschied bei den glänzenden Reflexionen auf Flächen ist weniger auffällig. Ein weiteres Bild von den Reflexionen ist Abbildung 15 zu finden.



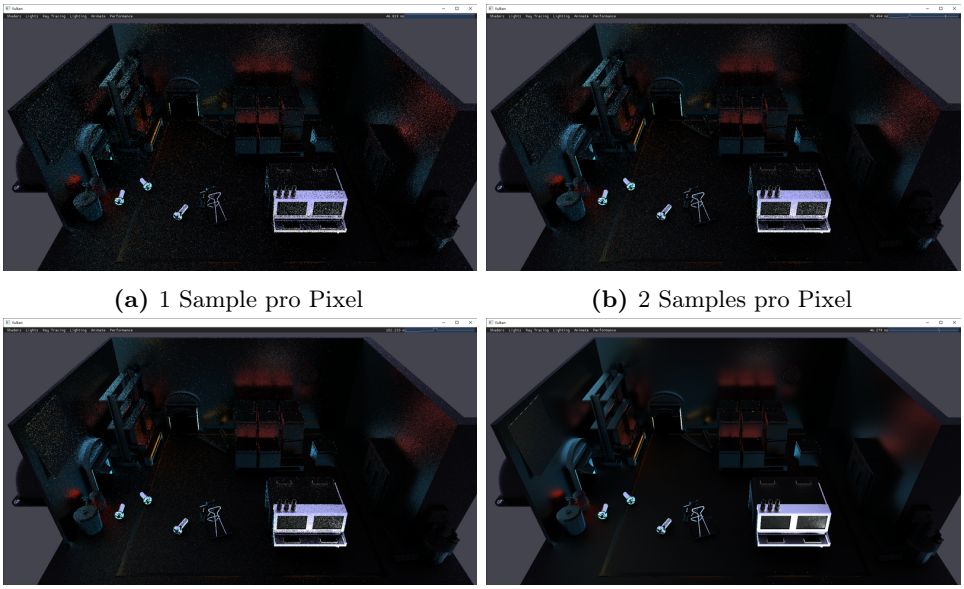
(a) 1 Sample pro Pixel

(b) 1 Sample pro Pixel, Reflexionen berechnet in halber Auflösung

(c) 1 Sample pro Pixel, akkumuliert

(d) 1 Sample pro Pixel, Reflexionen berechnet in halber Auflösung, akkumuliert

Abbildung 23: Reflexionen einzeln dargestellt, mit Vergleich zwischen der Berechnung der Reflexionen in voller und halber Auflösung.



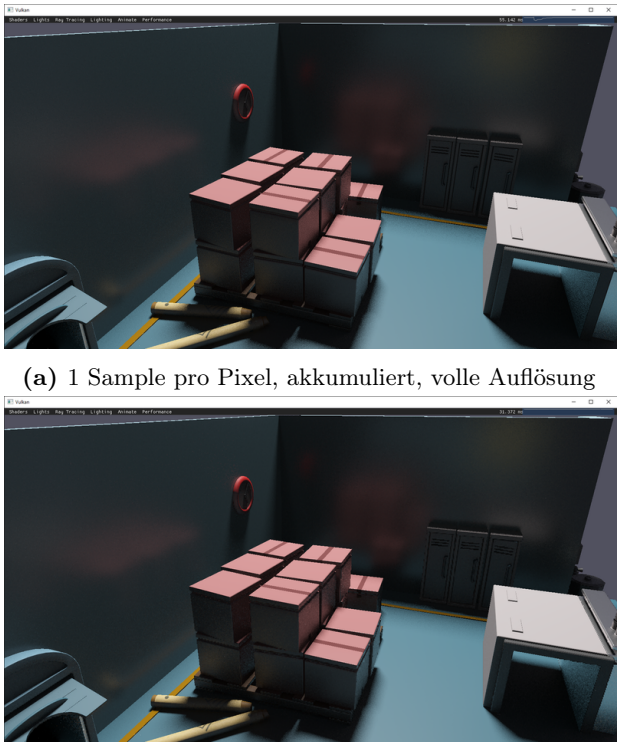
(a) 1 Sample pro Pixel

(b) 2 Samples pro Pixel

(c) 4 Samples pro Pixel

(d) 1 Sample pro Pixel, akkumuliert

Abbildung 24: Reflexionen einzeln dargestellt, mit verschiedener Sample-Anzahl.



(a) 1 Sample pro Pixel, akkumuliert, volle Auflösung

(b) 1 Sample pro Pixel, akkumuliert, halbe Auflösung

Abbildung 25: Reflexionen bei voller und halber Auflösung im Gesamtergebnis

Abbildung 26: Gesamtergebnis mit einer Punkt-Lichtquelle und einer Spot-Lichtquelle (grün), akkumuliert.

6.2.4 Gesamtergebnis

In den vorherigen Abschnitten wurde gezeigt, dass die Implementierung der Verfahren zur Berechnung von Schatten, Umgebungsverdeckung und Reflexionen flexibel ist und weitestgehend ohne Sonderbehandlung von Spezialfällen, wie das beispielsweise bei rasterisierungs-basierten Umsetzungen der Fall wäre, auskommt. Durch die Einstellungsmöglichkeiten der Parameter in der Benutzeroberfläche lassen sich die gewünschte visuelle Qualität beziehungsweise die Parameter, die zum gewünschten Zeit-Budget pro Frame bei der gegebenen Hardware-Performance führen, einstellen.

In Abbildung 26 ist das Gesamtergebnis mit einer Punktlichtquelle für die generelle Beleuchtung und einer zusätzlichen Spot-Lichtquelle. Schatten beider Lichtquellen lassen sich erkennen, die glänzenden Reflexionen an den Wänden und die spiegelnden Reflexionen in den Schrauben sind ebenfalls Teil der Szene. Durch die Umgebungsverdeckung werden Ecken, Objektkanten und scher erreichbare Stellen in der Szene abgedunkelt. Das Bild ist über einen längeren Zeitraum akkumuliert.

6.3 Erkenntnisgewinn über die RTX-API und deren Einsatz

In den folgenden Abschnitten werden die im Laufe dieser Arbeit gewonnenen Erkenntnisse über die RTX-API, deren Nutzung, deren Einsatzzwecke und die Zukunft der Technologie vorgestellt. Dabei handelt es sich um einen subjektiven Eindruck, welcher im Laufe der wissenschaftlichen Recherche und der Nutzung der Technologie im Rahmen dieser Arbeit gewonnen wurde.

6.3.1 Bewertung auf Programmebene

Die RTX-API hat sich als mächtiges Werkzeug für das Erstellen von Ray Tracing-Anwendungen herausgestellt. Ein Punkt, der dabei angesprochen werden muss, ist die Verfügbarkeit im Bezug auf Grafik-Programmierschnittstellen. Durch das Fehlen dieser in OpenGL muss entweder auf Vulkan oder auf DirectX 12 zurückgegriffen werden, welche beide als hardware-nahe Programmierschnittstellen einen signifikanten Mehraufwand beim Programmieren mit sich bringen, was im Implementationsteil dieser Arbeit deutlich wird.

Die Möglichkeiten der API haben für die Programmierung der Arbeit vollkommen ausgereicht und einen Punkt, an dem ein Feature der RTX-API, also in diesem Fall der Vulkan-Extension `VK_NV_ray_tracing` gefehlt hätte und wünschenswert gewesen wäre, gab es nicht. Der in Abschnitt 5.4.3 beschriebene Fehler in GLSL durch die Auslassung in der Spezifikation wird hierbei als Fehler im Einsatz von *subgroup operations*, nicht im Einsatz der Ray Tracing-Extension angesehen. Im Gegenteil gab es sogar Teile der RTX-API, welche in dieser Arbeit nicht zum Einsatz kamen und für speziellere Anwendungsfälle gedacht sind, wie Intersection Shader und Callable Shader. Daher werden die Möglichkeiten und Funktionen der API als mehr als Ausreichend eingeschätzt. Dem steht entgegen, dass der Aufbau der Beschleunigungsdatenstruktur und deren Traversierung, also die Hauptteile des Ray Tracing, die von der RTX-API übernommen werden, für den Benutzer nicht einsehbar und nur über undurchsichtige Einstellungen veränderbar sind. Dies schließt einen Einsatzzweck, bei dem es um eine spezielle Beschleunigungsdatenstruktur beziehungsweise eine Bewertung dieser geht, aus.

Die offizielle Dokumentation der Schnittstelle beschränkt sich auf die Spezifikation der Vulkan- und GLSL-Extensions, zusammen mit einigen Webseiten von *NVIDIA*, welche die Grundlagen und empfohlene Vorgehensweisen erklären. Durch das Unterstützen des Buches *Ray Tracing Gems*, herausgegeben von Haines und Akenine-Möller [HA19], welches ebenfalls eine Einführung in die RTX-API enthält, allerdings nur für DirectX 12, hat *NVIDIA* für eine für das Alter der Schnittstelle umfassende Dokumentationsgrundlage gesorgt. Diese Quelle, zusammen mit einigen der dazugehörigen Webseiten erschien allerdings erst im Laufe dieser Arbeit.

Einschätzung der Ausgereiftheit des Software-Ökosystems Die Software um die RTX-API in Kombination mit Vulkan umfasst in diesem Fall den Treiber, die Validation Layer und das Vulkan SDK, die Shader-Compiler und Debugging-Tools wie *NSight*. Das *NVIDIA*-hauseigene Debugging- und Profiling-Tool *NSight* unterstützt die Vulkan-Version der RTX-API erst seit Dezember 2018[32]. Das Vulkan SDK beinhaltet die Extension `VK_NV_ray_tracing`

[32] `https://docs.nvidia.com/nsight-graphics/2018.7/ReleaseNotes/index.html`, letzter Abruf: 5. Juni 2019

seit November 2018[33] und die Vorgängerversion, die als experimentell gekennzeichnete Extension `VK_NVX_raytracing` seit Oktober 2018[34]. Die Unterstützung für Fehlerfälle bei der Nutzung der Extension durch die Validation Layer hat sich im Laufe der Arbeit zwar weiterentwickelt, bis zum Ende dieser Arbeit gab es aber immer noch Absturz-Fälle, welche ohne eine Meldung durch die Validation Layer geschahen. Demnach sind diese als eventuell noch nicht als vollständig anzusehen, was die Unterstützung für die Ray Tracing Extension angeht. Der Treiber, im Laufe dieser Arbeit nur für das Windows-Betriebssystem getestet, hat bis zum Ende dieser Arbeit einige Instabilitäten gezeigt. So gab es mehrfach Ausfälle, gerade beim Ausführen der Anwendung mit besonders komplexen 3D-Modellen, welche nicht nur zum Absturz des Programms, sondern zum Absturz des Treibers, geäußert durch einen schwarzen Bildschirm bis zu einem Neustart geführt haben. Außerdem ist in einer solchen Situation die verwendete Grafikkarte nicht mehr in der Lage gewesen, in der Kombination mit dem Treiber ein Bild anzuzeigen und musste deshalb als Garantiefall ausgetauscht werden. Diesen unfertigen Eindruck bestätigt, dass bis zum Ende dieser Arbeit kein kommerzieller Einsatz der RTX-API in Zusammenhang mit Vulkan in einer nicht als experimentell oder als Vorabversion markierten Software bekannt ist.

Einschätzung der Eignung für bestimmte Szenarien Die Eignung der Nutzung der RTX-API lässt sich nach dieser Arbeit und der dazugehörigen Recherche wie folgt einschätzen: Der eigentliche Einsatzzweck, sowohl Software- als auch Marketing-technisch, liegt vor allem bei der Erweiterung bestehender Rendering-Systeme. So können Rasterisierungs-Pipelines wie zum Beispiel bei Computerspielen um Ray Tracing-Effekte erweitert werden, um eine bessere visuelle Qualität mit einem der Realität näherstehendem Ray Tracing-Algorithmus zu erreichen. Einzelne Bausteine der Pipeline können durch Ray Tracing-Ansätze ersetzt werden und pro Anwendung ist es den Entwicklern vorbehalten, für welche Effekte Ray Tracing eingesetzt werden soll und für welche nicht, um das optimale Ergebnis mit einer Mischung aus visueller Qualität und Performance zu erreichen. All dies kann mittels der RTX-API umgesetzt werden, ohne eine eigene Beschleunigungsdatenstruktur mit deren Aufbau, Traversierung und Aktualisierung zu implementieren und ein System zu erschaffen, in dem Ray Tracing-Anwendungen entwicklerfreundlich erstellt werden. Durch den Aufbau der RTX-API auf die Standard-Shadersprachen GLSL und HLSL mit eigenen Shadertypen muss keine neue Sprache gelernt oder eine Software-Architektur in bestehenden Shadertypen implementiert werden, bei welcher die Trennung der Belange dann wegen dem Wiederverwenden normaler Shader nicht vollständig gegeben ist.

[33] `https://vulkan.lunarg.com/doc/view/1.1.92.1/windows/release_notes.html`, letzter Abruf: 5. Juni 2019

[34] `https://vulkan.lunarg.com/doc/view/1.1.85.0/windows/release_notes.html`, letzter Abruf: 5. Juni 2019

Ein weiterer, sowohl Software- als auch Marketing-technisch geeigneter Einsatzzweck, zu dem die RTX-API bereits verwendet wird, ist die GPU-portierung bestehender Produktions-Renderings-Software, wie sie beispielsweise bei der Filmproduktion zum Einsatz kommt. Hier ist ebenfalls ein klassisches Ray Tracing-Szenario gegeben, welches so besonders einfach, also durch den Wegfall der Notwendigkeit der Implementation eigener Beschleunigungsdatenstrukturen und Ray Tracing-Softwarearchitektur, Grafik-Hardware und sogar die speziellen Ray Tracing-Cores eingesetzt werden kann.

Weniger geeignet scheint die RTX-API für Forschung und Lehre. Die nicht vorhandene Kontrolle des Nutzers über die Funktionsweise der Beschleunigungsdatenstruktur, mit der man sich beim Programmieren mit der RTX-API nicht oder nur wenig auseinander setzen muss, wäre dann mit dem Auslassen oder niedriger Priorisieren dessen in der Lehre verbunden. Eine Konzentration auf die Funktionalitäten, die mit der RTX-API selbst implementiert werden müssen, wie das Generieren der Strahlen, kann zwar von Nutzen sein, die Auswahl der Technologie in der Lehre muss aber wegen dieser Aufteilung sorgfältig getroffen werden. Die undurchsichtige Implementierung der Beschleunigungsdatenstruktur in der RTX-API suggeriert, dass an dieser Stelle die Forschung nur von den Implementierern des Treibers bei den jeweiligen Hardwareherstellern (im Falle eines kommenden gemeinsamen Standards) betrieben wird und sich zum Industriegeheimnis entwickelt, wenn sich mehrere Hersteller zu einem gemeinsamen Ray Tracing-Grafikschnittstellen-Standard zusammentun sollten, da dieser Punkt maßgeblich für Performance-Unterschiede zwischen Herstellern sorgen kann. Diese Suggestion und Entwicklung ist sicher nicht im Einklang mit wissenschaftlicher Lehre und Forschung. Außerdem sind die Notwendigkeit der Nutzung von Vulkan oder DirectX 12 und die hohen Hardware-Anforderungen eine weitere Hürde, die die Forschung und Lehre nehmen muss, um auf die RTX-API zurückgreifen zu können.

7 Fazit und Ausblick

In dieser Arbeit wurde ein Überblick über Ray Tracing und hybride Ray Tracing-Methoden gegeben und eine exemplarische Implementation eines hybriden Ray Tracing-Systems erstellt, erklärt und diese im Zusammenhang mit der verwendeten RTX-API evaluiert. Dabei musste aus Verfügbarkeitsgründen der RTX-API, welche zentrales Thema der Arbeit ist, auf Vulkan als Grafik-API zurückgegriffen werden. Die Nutzung von Vulkan hat sich insgesamt insofern positiv auf das Endergebnis ausgewirkt, dass die Vulkan-spezifischen Möglichkeiten, wie eine starke Entkopplung von GPU und CPU und die Initialisierung aller Ressourcen, der Rendering-Zustände und den statischen Teil der *commands* in der Phase vor dem eigentlichen Rendering, ausgenutzt wurden. Für das hybride Ray Tracing wurde ein Überblick über drei mögliche Anwendungsfälle für Ray Tracing innerhalb einer Grafik-Pipeline, Schatten, Umgebungsverdeckung und Reflexionen, gegeben und diese wurden exemplarisch implementiert und anhand dieser Implementierung wurde die RTX-API in ihren Einzelheiten gezeigt und erklärt. Alle drei dieser Anwendungsfälle haben die Gemeinsamkeit, dass sie durch das Ray Tracing ihre grundlegende Limitation bei einer Rasterisierungs-Umsetzung überkommen, nämlich die intuitive und physikalisch angelehnte Erweiterung von harten zu weichen Schatten bei der Schattenberechnung und das Verhindern von Randartefakten und Artefakten durch die begrenzten Geometrieinformationen im Bildraum bei der Umgebungsverdeckung und den Reflexionen. Dazu muss gesagt werden, dass das Ray Tracing nicht ohne Nachteile alle Artefakte behebt: Durch das Lösen von Integralen mit einzelnen Stichproben, in diesem Fall zum Beispiel Abtasten des Halbraumes für Umgebungsverdeckung, werden Rauschartefakte eingeführt, welche vor allem durch die begrenzte Anzahl an Samples innerhalb des Performance-Budgets auftreten und für ein visuell einwandfreies Ergebnis beseitigt werden müssen. Trotzdem erscheint eine Ray Tracing-Implementation der Techniken zur Berechnung solcher Effekte intuitiver und näher am grundlegenden Prinzip. Es werden also auch die Grenzen der neuartigen RTX-Technologie aufgezeigt und Kritik an der undurchsichtigen Natur der vorgegebenen Beschleunigungsdatenstruktur geübt. Wenn auch die Evaluation der Performance aus technischen Gründen kurz ausfallen musste, wird das Verhältnis des Aufwands der einzelnen Techniken zueinander klar. So stellt sich heraus, dass durch die deutlich längere Zeit, die die Reflexionsberechnung im Vergleich zu den anderen Techniken benötigt, hier der größte Anspruch an die Hardware gestellt wird, aber auch auf Anwendungsseite das größte Optimierungspotenzial vorhanden ist. Durch die zeitliche Nähe dieser Arbeit zu der Veröffentlichung der RTX-API wurde deren Entwicklung und die Verfügbarkeit der Software-Tools um die API herum beobachtet und auch dies mit in die Bewertung der Nutzbarkeit der API einbezogen.

So kann diese Arbeit dazu verwendet werden, Interessierten eine Einfüh-

rung in das Thema des hybriden Ray Tracings speziell mit der RTX-API zu geben, auf Grundlage derer beispielsweise einzelne Gesichtspunkte, welche nicht das Kernthema dieser Arbeit waren, vertieft und erweitert werden können. Denkbar wäre etwa eine Fokussierung auf die besonders performante beziehungsweise visuell ansprechende Implementierung der einzelnen Verfahren, das Finden neuer Optimierungsansätze für die Ray Tracing-Techniken oder ein Fokus auf das möglichst Artefakt-freie Entrauschen der Ergebnisbilder. Auch Optimierungen mit Level-of-Detail sind denkbar, besonders in Kombination von Messungen der Skalierbarkeit der Ray Tracing-Techniken in Abhängigkeit von der Szenenkomplexität im Vergleich zu einer Rasterisierungs-Implementierung. Der Einbau von einem oder mehreren Ray Tracing-Techniken in eine bestehende Rendering-Engine oder -Pipeline, welche auf Rasterisierung ausgelegt ist, ist ebenfalls ein interessanter Punkt, genau wie ein direkter Vergleich der Ray Tracing-Techniken mit dem Rasterisierungs-Pendant, um genaueres über das Verhältnis von gewonnener visueller Qualität zu verlorener Performance quantifizieren zu können. Für all diese Ideen kann diese Arbeit mit der grundlegenden Beispiel-Implementation einer hybriden Pipeline und der einzelnen Techniken hilfreich sein, auch um den Aufwand, der mit der Nutzung von Vulkan einhergeht durch das Vorstellen und Erklären der Implementation einzudämmen und in die Nutzung der API und deren Möglichkeiten und Verwendungszwecke einzuführen. Durch die kritische Beurteilung der API und deren Einsatzmöglichkeiten im Evaluationsteil der Arbeit kann diese Arbeit auch als Einschätzungshilfe zugezogen werden, wenn es darum gehen sollte, für einen bestimmten Einsatzzweck eine Technologie auszuwählen.

Literatur

[Aal18] Tatu Aalto. „The Latest Graphics Technology in Remedy's Northlight Engine". GDC 2018. 19. März 2018. URL: https://www.gdcvault.com/play/1025315/Advanced-Graphics-Techniques-Tutorial-The (besucht am 29. 05. 2019).

[Bar+19] Colin Barré-Brisebois et al. *Ray Tracing Gems*. Hrsg. von Eric Haines und Tomas Akenine-Möller. http://raytracinggems.com. Apress, 2019. Kap. 25.

[Bav+18] Louis Bavoil et al. *Hybrid Ray-Traced Ambient Occlusion*. Poster at HPG18 ACM SIGGRAPH / Eurographics High Performance Graphics. Aug. 2018. URL: https://casual-effects.com/research/Bavoil2018AO/index.html.

[BSD08] Louis Bavoil, Miguel Sainz und Rouslan Dimitrov. „Image-space Horizon-based Ambient Occlusion". In: *ACM SIGGRAPH 2008 Talks*. SIGGRAPH '08. Los Angeles, California: ACM, 2008, 22:1–22:1. ISBN: 978-1-60558-343-3. DOI: 10.1145/1401032.1401061. URL: http://doi.acm.org/10.1145/1401032.1401061.

[BWB19] Jakub Boksansky, Michael Wimmer und Jiri Bittner. *Ray Tracing Gems*. Hrsg. von Eric Haines und Tomas Akenine-Möller. http://raytracinggems.com. Apress, 2019. Kap. 13.

[CPC84] Robert L. Cook, Thomas Porter und Loren Carpenter. „Distributed Ray Tracing". In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '84. New York, NY, USA: ACM, 1984, S. 137–145. ISBN: 0-89791-138-5. DOI: 10.1145/800031.808590. URL: http://doi.acm.org/10.1145/800031.808590.

[CT82] R. L. Cook und K. E. Torrance. „A Reflectance Model for Computer Graphics". In: *ACM Trans. Graph.* 1.1 (Jan. 1982), S. 7–24. ISSN: 0730-0301. DOI: 10.1145/357290.357293. URL: http://doi.acm.org/10.1145/357290.357293.

[HA19] Eric Haines und Tomas Akenine-Möller, Hrsg. *Ray Tracing Gems*. http://raytracinggems.com. Apress, 2019.

[HS19] Eric Haines und Peter Shirley. *Ray Tracing Gems*. Hrsg. von Eric Haines und Tomas Akenine-Möller. http://raytracinggems.com. Apress, 2019. Kap. 1.

[HT07] Lee Howes und David Thomas. *Gpu Gems 3*. Hrsg. von Hubert Nguyen. First. Addison-Wesley Professional, 2007. Kap. 37. ISBN: 9780321545428.

[Kaj86] James T. Kajiya. „The Rendering Equation“. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, S. 143–150. ISBN: 0-89791-196-2. DOI: 10.1145/15922.15902. URL: http://doi.acm.org/10.1145/15922.15902.

[Kar13] Brian Karis. „Real Shading in Unreal Engine 4“. In: *ACM SIGGRAPH 2013 Courses: Physically Based Shading in Theory and Practice*. SIGGRAPH '13. Anaheim, California: ACM, 2013, 22:1–22:8. ISBN: 978-1-4503-2339-0. DOI: 10.1145/2504435.2504457. URL: http://doi.acm.org/10.1145/2504435.2504457.

[Mar+18] Adam Marrs et al. „Adaptive Temporal Antialiasing“. In: *ACM SIGGRAPH / Eurographics High Performance Graphics*. Aug. 2018, S. 4. URL: https://casual-effects.com/research/Marrs2018Antialias/index.html.

[McG19] Morgan McGuire. „Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields“. GDC 2019. 18. März 2019. URL: https://gdcvault.com/play/1026182/ (besucht am 29. 05. 2019).

[Mit07] Martin Mittring. „Finding Next Gen: CryEngine 2“. In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: ACM, 2007, S. 97–121. ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281671. URL: http://doi.acm.org/10.1145/1281500.1281671.

[MY18] Ian Mallett und Cem Yuksel. „Deferred Adaptive Compute Shading“. In: *Proceedings of the Conference on High-Performance Graphics*. HPG '18. Vancouver, British Columbia, Canada: ACM, 2018, 3:1–3:4. ISBN: 978-1-4503-5896-5. DOI: 10.1145/3231578.3232160. URL: http://doi.acm.org/10.1145/3231578.3232160.

[NVI18] NVIDIA. „NVIDIA Turing GPU Architecture“. 14. Sep. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (besucht am 03. 06. 2019).

[Par+10] Steven G. Parker et al. „OptiX: A General Purpose Ray Tracing Engine“. In: *ACM Trans. Graph.* 29.4 (Juli 2010), 66:1–66:13. ISSN: 0730-0301. DOI: 10.1145/1778765.1778803. URL: http://doi.acm.org/10.1145/1778765.1778803.

[Pha19a] Pharr. *Ray Tracing Gems*. Hrsg. von Eric Haines und Tomas Akenine-Möller. http://raytracinggems.com. Apress, 2019. Kap. 17.

[Pha19b] Matt Pharr. *Ray Tracing Gems*. Hrsg. von Eric Haines und Tomas Akenine-Möller. http://raytracinggems.com. Apress, 2019. Kap. 15.

[Sch+17] Christoph Schied et al. „Spatiotemporal Variance-guided Filtering: Real-time Reconstruction for Path-traced Global Illumination“. In: *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: ACM, 2017, 2:1–2:12. ISBN: 978-1-4503-5101-0. DOI: 10.1145/3105762.3105770. URL: http://doi.acm.org/10.1145/3105762.3105770.

[Sch94] Christophe Schlick. „An Inexpensive BRDF Model for Physically-based Rendering“. In: *Computer Graphics Forum* 13.3 (1994), S. 233–246. DOI: 10.1111/1467-8659.1330233. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/1467-8659.1330233. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.1330233.

[Shi+19a] Peter Shirley et al. *Ray Tracing Gems*. Hrsg. von Eric Haines und Tomas Akenine-Möller. http://raytracinggems.com. Apress, 2019. Kap. 2.

[Shi+19b] Peter Shirley et al. *Ray Tracing Gems*. Hrsg. von Eric Haines und Tomas Akenine-Möller. http://raytracinggems.com. Apress, 2019. Kap. 16.

[UL 19] UL Benchmarks. „3DMark Technical Guide“. 20. Mai 2019. URL: https://s3.amazonaws.com/download-aws.futuremark.com/3dmark-technical-guide.pdf (besucht am 01. 06. 2019).

[Wal+07] Bruce Walter et al. „Microfacet Models for Refraction Through Rough Surfaces“. In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. EGSR'07. Grenoble, France: Eurographics Association, 2007, S. 195–206. ISBN: 978-3-905673-52-4. DOI: 10.2312/EGWR/EGSR07/195-206. URL: http://dx.doi.org/10.2312/EGWR/EGSR07/195-206.

[Whi80] Turner Whitted. „An Improved Illumination Model for Shaded Display“. In: *Commun. ACM* 23.6 (Juni 1980), S. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882. URL: http://doi.acm.org/10.1145/358876.358882.

[ZIK98] S. Zhukov, A. Iones und G. Kronin. „An ambient light illumination model“. In: *Rendering Techniques '98*. Hrsg. von George Drettakis und Nelson Max. Vienna: Springer Vienna, 1998, S. 45–55. ISBN: 978-3-7091-6453-2.