



HOCHSCHULE BREMEN
FAKULTÄT 4 – ELEKTROTECHNIK UND INFORMATIK
INTERNATIONALER STUDIENGANG MEDIENINFORMATIK (B.Sc.)

BACHELORTHESES

Anwendung von Hardware-Raytracing zur Optimierung der Treffererkennung

– Eine Untersuchung zur Leistungssteigerung und
Präzisionsverbesserung in Echtzeit-Anwendungen –

Kevin Niclas Kügler (399027)

Prüfer: Prof. Dr. Martin Hering-Bertram

Zweitprüfer: Prof. Dr. Volker Paelke

26. August 2024 (Version 1.00)

Inhaltsverzeichnis

1 Einleitung	4
1.1 Problemstellung	4
1.2 Lösungsansatz	4
2 Grundlagen	6
2.1 Raytracing	6
2.2 Physik-Engine	6
2.3 Vulkan	8
3 Konzeption	12
3.1 Zusammenfassung	15
4 Prototypische Realisierung	16
4.1 Prism Engine-Implementation	16
4.2 Unreal Engine 5 Implementation	29
5 Evaluation	31
6 Zusammenfassung und Ausblick	37
7 Literaturverzeichnis	38
8 Listingverzeichnis	41
9 Abbildungsverzeichnis	42
10 Glossar	43
11 Akronyme	45
12 Anhangsverzeichnis	46

Folgenden Personen gebührt mein besonderer Dank

Alexander Stöwing, für seine unermüdliche Hilfe bei Fragen jeglicher Art

und

meiner wunderbaren Frau, Natalie, für ihre mentale Unterstützung, auch wenn es
zwischendurch sehr schwer wurde

und

meiner wundervollen Tochter, Lucy

1 Einleitung

In der Welt der Videospiele und erweiterten Realitätsanwendungen (XR) stellen die steigende Komplexität und die Anforderungen an die Treffererkennung eine Herausforderung dar. Traditionelle CPU-basierte Methoden stoßen zunehmend an ihre Grenzen, was die Systemperformance und die Spielerfahrung beeinträchtigt. Diese Arbeit erforscht die Möglichkeit, die Treffererkennung durch den Einsatz von Raytracing-Kernen (RT-Kerne) auf GPUs zu realisieren, um die Präzision zu steigern und die CPU-Last zu verringern.

1.1 Problemstellung

Die Problematik der Treffererkennung in interaktiven Echtzeit-Umgebungen verschärft sich mit der zunehmenden Komplexität von Spielwelten und der fortlaufenden Entwicklung realistischerer 3D-Modelle. Die CPU ist in einer modernen Game-Engine oft mit mehreren Aufgaben belastet, einschließlich KI-Steuerung, Spiellogik und der Verwaltung von Benutzerinputs, wodurch die Ressourcen für die Treffererkennung limitiert sind.

Die traditionelle Hit-Scan-Methode, die auf der CPU ausgeführt wird, stößt bei der Skalierung auf Grenzen. Typischerweise werden Hitboxen in Form von Quadern, Sphären und Kapseln zur Vereinfachung des 3D-Modells verwendet. Der Strahl von seinem Startpunkt in die gegebene Richtung projiziert. Trifft der Strahl auf eine Hitbox, wird die Berechnung beendet und Trefferinformationen zurückgegeben. Alternativ kann das 3D-Modell selber als Hitbox fungieren, allerdings mit deutlich erhöhtem Rechenaufwand.

Dies führt zu einem Dilemma: Die Erhöhung der Präzision der Treffererkennung verlangt mehr Rechenzeit, was wiederum die Bildwiederholrate und damit die Immersion und Interaktion beeinträchtigen kann.

1.2 Lösungsansatz

Im Rahmen der Arbeit wird das Hit-Scan-Verfahren auf die GPU verlagert, um die Rechenleistung der spezialisierten RT-Kerne moderner Grafikkarten zu nutzen. Diese Anpassung zielt darauf ab, die Geschwindigkeit und Präzision der Treffererkennung zu steigern, während gleichzeitig die Belastung der CPU reduziert wird.

Die Verlagerung des GPU-basierten Hit-Scan-Verfahrens soll mit der Programmierschnittstelle (API) Vulkan,¹ speziell Vulkan Ray Queries, realisiert werden. Ray Queries ermöglichen den Einsatz der RT-Kerne außerhalb von Rendering-Shadern. Sie können für beliebige Zwecke benutzt werden. So auch für Treffererkennung.[1]

Mehrere Testszenen mit variierender Anzahl von statischen 3D-Modellen sollen der Evaluierung dienen: Durch Vergleichen der Performance des Verfahrens auf der GPU mit dem bestehenden CPU-basierten Ansatz in Unreal Engine 5 (UE5),² einer 2022 veröffentlichten Game-Engine die von Epic Games entwickelt wurde, wird überprüft ob die GPU-

¹<https://www.vulkan.org/>

²<https://unrealengine.com/en-US/unreal-engine-5>

1 Einleitung

Implementierung eine schnellere Verarbeitung ermöglicht. Dieser Vergleich soll die Vorteile der GPU-Nutzung für Echtzeit-Treffererkennung in interaktiven Anwendungen verdeutlichen. Der GPU-basierte Ansatz wird in einer zuvor eigens entwickelten Rendering-Engine, genannt Prism, verwirklicht.

2 Grundlagen

Dieses Kapitel widmet sich den für diese Arbeit relevanten Themen und beschreibt Grundlagen dieser. Dazu gehört das Raytracing, eine Methode zur realistischen Bildsynthese, die Lichtstrahlen in einer 3D-Szene simuliert. Weiterhin wird die Anwendung von Raytracing in Physik-Engines zur Kollisionserkennung in Videospielen beschrieben. Zudem wird Vulkan, eine leistungsfähige und hardwarenahe Programmierschnittstelle für Grafikkarten, behandelt, die Entwicklern präzise Kontrolle über die Hardware ermöglicht.

2.1 Raytracing

Andrew Glassner beschreibt Raytracing als Technik zur Bildsynthese, das Erstellen eines 2D Bildes aus einer 3D Welt[2]. Ein Monitor besteht aus unzähligen Pixeln mit jeweils einem Farbwert und Glassner erklärt, dass jedes Pixel eine Lochkamera repräsentiert aus der die 3D Welt betrachtet wird. Raytracing ahmt Photonen aus der realen Welt nach und gibt so dem Pixel seinen Farbwert basierend auf den Farbwerten der Objekte auf die das digitale Photon, auch Strahl genannt trifft, abprallt und irgendwann zurück Richtung Lochkamera fliegt.

Seit Immel et al. und James Kajiya zur gleichen Zeit auf der SIGGRAPH³ 1986 verschiedene Rendertechniken mit ihrer Rendering-Gleichung vereinheitlicht haben[3, 4], hat sich Raytracing stetig weiterentwickelt und gilt in der Videospielbranche als heiliger Gral des Renderings[5]. Nichts in Spielen ist realistischer als eine virtuelle Welt die von unserer nicht mehr zu unterscheiden ist. Trotz unzähliger Optimierungstechniken sowie Durchbrüche bei der Hardwareentwicklung waren Grafikkarten nicht stark genug Raytracing in 30 oder mehr Frames per Second (FPS)⁴ in gängigen Bildschirmauflösungen darzustellen. Erst die Erfindung von Raytracing-Kernen in Kombination mit Machine Learning (ML) Algorithmen zum hochskalieren von Bildern, genannt Deep Learning Super Sampling (DLSS)[6] auf den Turing-Grafikkarten der Firma NVIDIA⁵ löste das Problem der niedrigen FPS[5]. Seitdem haben sich beide Technologien weiterentwickelt. DLSS Frame Generation[6] erlaubt die Generierung von Bildern anstatt diese nur hoch zu skalieren. Weiterhin ist mit DLSS Ray Reconstruction[6] eine ML-basierte Möglichkeit geschaffen um die Anzahl an Strahlen welche in die 3D Szene geschossen werden verringert, aber dennoch das finale Bild annähernd korrekt darstellt.

2.2 Physik-Engine

Doch Raytracing ist nicht nur relevant für die Bildsynthese. Physik-Engines nutzen die Technik in abgewandelter Form ebenfalls zum projizieren von Strahlen zur Treffererkennung. In Videospielen verwendete Waffen wie Gewehre oder Bögen sind oft nicht vollständig über Zeit hinweg simuliert, sondern werden innerhalb eines Berechnungszyklus, genannt Tick,

³<https://www.siggraph.org/>

⁴dt. Bilder pro Sekunde

⁵<https://www.nvidia.com>

durch die 3D-Szene projiziert. Trifft die Kugel dabei ein Objekt kann sie absorbiert werden, es durchstoßen oder abprallen. Ersterer Fall ist der relevanteste. Die Physik-Engine berechnet beim Aufprall zudem die Kräfte welche auf das Objekt einwirken und wendet sie auf selbiges an, je nach Konfiguration. Um den Berechnungsaufwand gering zu halten werden bei der Überprüfung ob die Kugel ein Objekt trifft nicht alle Polygone zur Strahl-Polygon-Verschneidung, wie beispielsweise durch den Möller-Trumbore-Schnittalgorithmus[7], herangezogen sondern lediglich vereinfachte geometrische Formen, genannt Hitboxen, wie Quadern, Sphären oder Kapseln. Diese lassen sich mathematisch darstellen ohne dass Polygone benötigt werden[8]. Dieses Verfahren der Treffererkennung ist seit Jahrzehnten Standard in der Videospielbranche.

Eine weitere Optimierung sind Bounding Volumes (BV). Anstatt den Strahl mit jedem einzigen 3D-Modell und dessen Polygonen, die in fotorealistischen Spielen in die Zehn- bis Hunderttausende gehen können, innerhalb der 3D-Szene auf einen Treffer zu prüfen, werden die 3D-Modelle mit Bounding Volumes umgeben. Die bekannteste Form sind Axis-aligned Bounding Boxes (AABB). Das 3D-Modell wird hierbei von einem an den Achsen des Koordinatensystems ausgerichteten Quaders vollständig umschlossen. Im ersten Schritt wird der Strahl nun auf Kollision mit allen AABBs geprüft. Gibt es einen Treffer werden die einzelnen Polygone mit dem Strahl auf einen Schnittpunkt überprüft. Dies verringert den Berechnungsaufwand signifikant. Dennoch ist der Aufwand bei sehr detaillierten 3D-Modellen weiterhin hoch, besonders in Anbetracht der immer steigenden Polygon-Anzahl. Als Lösung dafür gibt es ein Subset der BVs: Bounding Volume Hierarchies (BVH). Die 3D-Modelle werden in mehrere in einer Baumstruktur angeordneten Bounding Volumes unterteilt. Wenngleich es üblich ist, dass einzelne BVs sich nicht überlappen und die 3D-Modellgeometrie so in mehreren BVs enthalten ist, kann dies je nach Anwendungs- oder Optimierungsfall durchaus vorkommen. Anders als in beispielsweise Octrees, erfunden von Donald Meagher[9], sind BVHs weder an eine Unterteilungstiefe gebunden noch müssen die Kind-Knoten innerhalb des Vater-Knotens positioniert sein.[8]

Bounding Volume Hierarchies sind Gegenstand vieler Forschungen. Aktuelle Veröffentlichungen umfassen die Nutzung von Machine Learning zur Erstellung von BVHs, wie Philippe Weier et al. zeigen[10] sowie Mochi, eine Collision Detection-Engine von Mandarapu et al. welche die RT-Kerne der Grafikkarte nutzt[11].

Die zuvor angesprochenen RT-Kerne bilden das Prinzip der Bounding Volume Hierarchies und der Strahl-Polygon-Verschneidung in Hardware ab. John Burgess beschreibt in seinem Artikel *RTX on - The NVIDIA Turing GPU* das Funktionsprinzip der RT-Kerne: Ein Shader mit Strahl-Informationen übergibt diese an die RT-Kerne, welche den generierten BVH solange durchlaufen bis entweder eine Verschneidung mit einem Polygon festgestellt wurde oder der Strahl ins Nichts ging. Anschließend wird das Ergebnis an weitere Shader zurückgegeben. Abbildung 1 beschreibt diesen Prozess im unteren Teil. Die obere Hälfte zeigt den Prozess auf Grafikhardware ohne RT-Kerne. Laut Burgess ist der Performance-Gewinn gegenüber dem Raytracing ohne RT-Kerne Sieben Mal höher.[12] Auf Abbildung 11 im Anhang ist eine Visualisierung des BVH-Algorithmus zu finden.

Die Nutzung der RT-Kerne außerhalb des 3D-Renderings ist ebenfalls Gegenstand mehrerer Arbeiten. So haben Steinberger et al. 2019 die Verwendung von RT-Kernen zur Punkt-

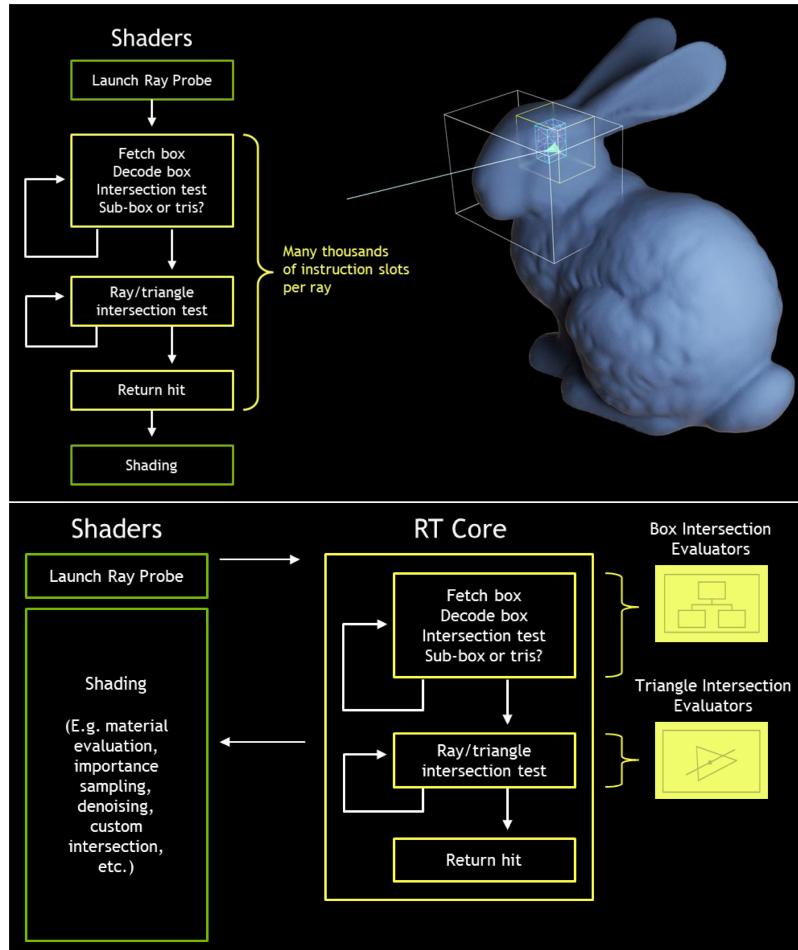


Abbildung 1: Funktionsweise der RT-Kerne; oberer Teil zeigt Raytracing ohne RT-Kerne, unterer zeigt diesen mit RT-Kernen[12]

lokalisierung innerhalb von unstrukturierten Tetraeder-Meshes erforscht und festgestellt, dass die Implementation mit RT-Kernen schneller war[13]. Yuhao Zhu hingegen hat 2022 eine Methode zur Leistungssteigerung der Nachbarn-Klassifikation unter Verwendung von RT-Kernen vorgeschlagen[14]. Experimentelle Ergebnisse zeigten eine 2,2-65-fache Steigerung der Performance gegenüber bestehenden Programmbibliotheken.

2.3 Vulkan

Vulkan ist eine 2016 veröffentlichte, quelloffene in C geschriebene Programmierschnittstelle (API⁶) zur Grafikkarte und wird von der Khronos Group⁷ entwickelt. Anders als OpenGL⁸ ist Vulkan hardwarenäher und erlaubt dem Programmierer genauere Kontrolle über die

⁶ Application Programmable Interface

⁷ <https://www.khronos.org/>

⁸ <https://www.opengl.org/>

Grafikkarte. Ältere APIs wie OpenGL übernehmen viele Funktionen wie Fehlerbehandlung, Synchronisation und Speichermanagement die in Vulkan vom Programmierer explizit gesteuert werden müssen und verringert so den Programmieraufwand. Dies hat jedoch zur Folge, dass Detailoptimierungen schwieriger sind. Der Programmierer ist daher auf die Qualität der API angewiesen.[15] Die Hardwareähnlichkeit von Vulkan haben jedoch den Nachteil, dass, wenn bestimmte Funktions-Parameter oder -Aufrufe falsch gesetzt sind oder fehlen, dies zu Programmabstürzen oder fehlerhaften Bildausgaben führen kann. Weiterhin erzeugt die Ausführlichkeit bereits bei kleinen Programmen wie dem *Hello World Polygon* für über 1000 Zeilen Quellcode. Andererseits wird Vulkan wegen diesen Eigenschaften von professionellen 3D-Programmierern geschätzt und gilt mittlerweile als Standard in der Spieleindustrie und ist auf fast jeder Betriebssystem-Plattform. Eine ähnliche API stellt Microsoft bereit: DirectX 12.⁹

Vulkan kann mehrere Grafikkarten unabhängig voneinander ansprechen, ein zentraler Unterschied zu OpenGL. Allerdings ist es nicht nur auf Grafikkarten beschränkt. Ähnliche Verarbeitungs-Hardware wie Digitale Signalprozessoren und reine Compute-Shader-Karten die universelle mathematische Berechnungen durchführen kann ebenfalls angesprochen werden[15]. Um mit Hardware zu kommunizieren bedarf es zweier Vulkan-Konstrukte. Das Physical Device repräsentiert eine oder mehrere Hardware-Komponenten und stellt lediglich einen Ansteuerpunkt im Code dar. Erst Mithilfe des Logical Device, welches tatsächliche Ressourcen allokiert und einen Zustand besitzt, kann das Physical Device gezielt abgefragt sowie gesteuert werden. Dies ist ein grundlegender Unterschied zu OpenGL bei dem die Ressourcen und der Zustand global sind. Ein Physical Device kann zu mehr als einem Logical Device gehören.[15, 16, 17]

Einer der Grundeigenschaften von Vulkan ist die Erweiterbarkeit. Die API ist in zwei Bereiche unterteilt: *Core* und *Extensions*. Jede Hardware die mit Vulkan interagieren kann, muss alle Inhalte des *Core* in ihrem Treiber implementiert haben. Dagegen sind *Extensions* optional. Hardware-Treiber können eine beliebige Anzahl an *Extensions* bereitstellen. Auch muss der Vulkan-Treiber selbst diese *Extension* anbieten.[15, 18]

Anders als OpenGL setzt Vulkan auf ein rechtshändiges kartesisches 3D-Koordinatensystem und besitzt weitere feine Unterschiede bei Koordinatensystem-Berechnungen.[19]

Um Arbeit auf der Hardware verrichten zu lassen, stellt Vulkan Queues bereit. Eine Queue ist einer Queue Family zugehörig welche mithilfe des Physical Device abgefragt und anschließend bei der Erstellung des Logical Device angefragt werden können. Da Vulkan Primär für das Rendering und Compute-Operationen entworfen wurde, sind die beiden bekanntesten Queue Families die Graphics Queue Family und Compute Queue Family[20]. Anders als viele andere Operationen erzeugt der Programmierer Queues nicht selbst. Der Treiber übernimmt dies beim Erstellen des Logical Device[15].

Konkrete Arbeitsbefehle werden durch Command Buffer aufgezeichnet und gespeichert. Diese Command Buffer können je nach gesetztem Flag bei Erstellung mehrmals abgespielt werden, was wertvolle Zeit spart. Die Command Buffer werden in die zuvor erklärten Queues eingereiht. Die Queues arbeiten die Command Buffer meist nach *First In - First Out* Prinzip

⁹<https://gpuopen.com/directx12/>

ab, die Grafikkarte kann allerdings diese Reihenfolge brechen wenn sich Optimierungsmöglichkeiten offenbaren[20].[15]

Die Verwaltung von RAM¹⁰ und Video-RAM (VRAM)¹¹ in Vulkan ist in zwei Arten unterteilt: Host Memory und Device Memory[15]. Beide Arten müssen durch den Programmierer verwaltet werden, ähnlich dem Speichermodell in C++. Ein Garbage-Collector existiert nicht, kann jedoch manuell implementiert werden. Dieses explizite Speichermodell erlaubt gezielte und performante Verwaltung des Grafikspeichers. Zuvor in OpenGL wurde der Speicher durch den Grafiktreiber allokiert und musste von diesem abgeschätzt werden. Um Speicherblöcke zwischen CPU und GPU zu transferieren wird Memory Mapping verwendet. Speicherzugriff innerhalb derselben Hardwarekomponente, beispielsweise des CPU L2 Cache, ist sehr schnell. Kommunikation zu einer anderen Hardwarekomponente ist bedeutend langsamer da hierbei anstatt auf internen Speicher innerhalb derselben Komponente auf den Peripheral Component Interconnect Express (PCI-E)-Bus als Trägermedium zurückgegriffen werden muss. Der PCI-E-Bus kann unter umständen von weiterer anderer Hardware benutzt werden welche ggf. viel Bandbreite des Bus in Besitz nehmen.[21]

Multithreading spielt in Vulkan eine wichtige Rolle. Viele Performance-kritische Operationen sind in Vulkan nicht synchron, d.h. dass mehrere Threads gleichzeitig auf dieselbe Ressource schreiben und lesen können[15]. Dies kann zu inkonsistenten Zuständen der Daten führen und so für Fehler und Abstürze sorgen. Um dies zu vermeiden gibt es mehrere Arten von Synchronisationsmechanismen in Vulkan[22]. Die einfachste Art ist auf das gesamte Logical Device zu warten. Eine solche Warteoperation ist teuer, denn die CPU muss in diesem Fall darauf warten dass das Logical Device komplett im Leerlauf ist. Eine zweite Möglichkeit stellt das Warten auf eine Queue dar. Hier muss lediglich darauf gewartet werden, dass alle Command Buffer innerhalb der Queue abgearbeitet wurden.

Für gezieltes Warten stellt Vulkan Barriers zur Verfügung. Es gibt mehrere Arten von Barriers, eine davon sind sogenannte Memory Barriers. Mithilfe dieser kann sichergestellt werden, dass Speicheroperationen wie der Transfer zwischen zwei Speicherpuffern abgeschlossen sind, ein Command Buffer beispielsweise darauf zugreifen darf.[23]

Shader stellen eine essentielle Komponente zum Verrichten von Arbeit auf Grafik- und Compute-Hardware dar. Sie bestehen aus Code einer C-ähnlichen Sprache, der OpenGL Shading Language (GLSL)[24]. Es gibt mehrere Shader-Arten. Darunter Vertex-Shader zum verarbeiten der 3D-Modelldaten, Fragment-Shader die einzelne Pixel berechnen und Compute-Shader mit denen beliebige mathematische Berechnungen und andere nicht Rendering-relevante Funktionen ausgeführt werden können.[15] Im Gegensatz zu OpenGL verlangt Vulkan Shader in einem Bytecode-Format, Standard Portable Intermediate Representation-V (SPIR-V). Dieses Zwischenformat ermöglicht es Vulkan komplexe High-Level Compiler aus den Hardware-Treibern auf externe Compiler-Programme auszulagern. Dadurch muss lediglich der standardisierte Bytecode auf der Hardware ausgeführt werden was so die Laufzeit-Performance steigert.[25]

Buffer sind eine fundamentale Ressource in Vulkan und bilden mit dem Buffer Memory die primäre Möglichkeit Daten zu verwalten. Buffer sind ein linearer Speicherblock und kön-

¹⁰ dt. Arbeitsspeicher

¹¹ dt. Grafikspeicher

nen vielseitig eingesetzt werden. Beispiele umfassen Vertex Buffer für 3D-Modelldaten sowie Index Buffer die Vertices in Vertex Buffer referenzieren. Auch gibt es Shader Storage Buffer Objects (SSBO) die zum Datentransfer zwischen CPU und Shader benutzt werden.[15] Um diese Memory Handles innerhalb von Vulkan und der Hardware zu nutzen sind Descriptors nötig. Sie dienen als Bindeglied zwischen Shader und Vulkan-API sowie VRAM. Descriptors sind in Descriptor Sets gebündelt welche wiederrum an einzelne Shader gebunden sind. In Shadern lassen sich die gebundenen Descriptor Sets ansteuern und Lese- sowie Schreiboperationen auf ihnen ausführen. Die ist ein Vorteil, da diese bis zum Binden an einen anderen Shader verfügbar bleiben und dadurch aktiv im Speicher verbleiben. Descriptor Sets werden auf Grundlage einer Art Blaupause, den Descriptor Set Layouts generiert. Durch die Vorgabe eines nicht veränderbaren Layouts kann der Hardware-Treiber die Ressourcennutzung optimieren. Zur weiteren Leistungssteigerung werden Descriptor Sets aus Descriptor Pools angefragt. Pooling ist in der Programmierung ein verbreitetes Mittel um die Erzeugung von Objekten zu optiminieren. Jede Speicherallokierung ist verhältnismäßig teuer. Um dies zu umgehen kann eine feste Menge an Objekten vorallokiert werden. Wird eines dieser Objekte, beispielsweise ein Descriptor Set benötigt, gibt der Descriptor Pool eines zur Verwendung frei. Wird es nicht mehr benötigt, nimmt der Pool das Set wieder an sich. Bei der Freigabe werden jegliche eventuell vorhandenen Zustände des Descriptor Sets zurückgesetzt um einen konsistenten Datenzustand zu gewährleisten.[15, 26]

Shader und Descriptor Sets sind aus Gründen der Performance an Pipelines gebunden. Jede Kombination aus Shadern wird einer Pipeline zugeordnet. Es gibt mehrere Arten von Pipelines, die wichtigsten sind Graphics Pipelines für das Rendering und Compute Pipelines für allgemeine mathematische Berechnungen. Soll beispielsweise ein 3D-Modell mit Texturen und prozedural generierten Polygonen gerendert werden, wird eine Graphics Pipeline mit einem Vertex-, Fragment- und Mesh-Shader benötigt. letzterer erzeugt Geometrie prozedural ohne vorige Erzeugung eines Vertex-Buffers. Der Vorteil an Pipelines besteht darin, dass sie bei Initialisierung von Vulkan vorab kompiliert werden. Dies erhöht die Performance zur Laufzeit und verhindert stottern durch Shader- und Pipeline-Kompilierung.[15, 27]

3 Konzeption

In modernen Game-Engines wird die Treffererkennung typischerweise durch die folgende Funktion realisiert:

Listing 1: *traceRay* Funktion in Pseudocode

```
1 void traceRay(Origin, Direction, minDistance, maxDistance)
```

Diese Funktion fragt die Physik-Engine ab, welche den Strahl durch die Szene projiziert und auf mögliche Kollisionen prüft. In den verbreiteten Game-Engines wie Unreal Engine und Unity¹² wird diese Berechnung auf der CPU ausgeführt und verwendet vereinfachte Hitboxen. Die Nutzung von Hitboxen birgt jedoch mehrere Nachteile. Hitboxen sind eine sehr rudimentäre Annäherung an das eigentliche 3D-Modell und können zu False-Negatives oder False-Positives führen. Insbesondere im Bereich des kompetitiven Multiplayerspiels, wie bei E-Sport-Turnieren, bei denen um hohe Geldsummen gespielt wird, können solche Fehler den Ausgang des Spiels und somit des Turniers entscheidend beeinflussen.

Ein prominentes Beispiel ist der First-Person-Shooter Apex Legends,¹³ entwickelt von Respawn Entertainment,¹⁴ in dem die Hitboxen mehrerer Charaktere nach erheblicher Kritik der Spielerschaft angepasst werden mussten[28, 29]. Solche Ungenauigkeiten beeinträchtigen nicht nur das Spielerlebnis, sondern erfordern auch einen zusätzlichen Entwicklungsaufwand. Um ein hoch detailliertes 3D-Modell korrekt darzustellen, werden oft viele Hitboxen benötigt. Diese einfachen geometrischen Formen können jedoch nur eine begrenzte Abdeckung des gesamten 3D-Modells bieten. Je mehr Hitboxen benötigt werden, desto mehr Zeit und Aufwand erfordert ihre Platzierung.

Als Alternative zu einfachen geometrischen Hitboxen könnten vereinfachte 3D-Modelle oder sogar die originalen 3D-Modelle selbst verwendet werden. Dies stellt jedoch herkömmliche Physik-Engines vor Herausforderungen, da sie nicht oder nur suboptimal für komplexe geometrische Hitboxen ausgelegt sind. Der Rechenaufwand steigt und die Laufzeitperformance sinkt signifikant, insbesondere in Szenen mit einer hohen Anzahl von 3D-Modellen. Die CPU wird durch die hohe Polygonanzahl und die damit verbundene Berechnung von Strahl-Polygon-Verschneidungen stark belastet.

Eine vielversprechende Lösung für diese Performance-Probleme bietet die Verlagerung der Berechnungen auf die GPU durch den Einsatz der RT-Kerne. Vulkan stellt hierfür Ray Queries zur Verfügung, um die RT-Kerne auch außerhalb der Rendering-Pipeline zu nutzen.

Das Konzept dieser Arbeit ist, die zuvor beschriebene Funktion *traceRay* auf der GPU umzusetzen. Die Implementierung erfolgt in einer zuvor eigens entwickelten Game-Engine namens Prism. Anschließend sollen verschiedene Testscenarien in dieser GPU-basierten Implementierung sowie in einer CPU-basierten Referenzimplementierung in UE5 durchlaufen und miteinander verglichen werden. Eine zentrale Anforderung hierbei ist, dass der Aufruf der *traceRay*-Funktion synchron erfolgt. Die CPU wartet, bis die GPU die Berechnung

¹²<https://unity.com>

¹³<https://www.ea.com/de-de/games/apex-legends>

¹⁴<https://www.respawn.com/>

3 Konzeption

abgeschlossen hat. In der Computergrafik ist dies unüblich, da GPUs auf Parallelisierung ausgelegt sind. Mit der Weiterentwicklung von GPU-basiertem Hardware-Raytracing besteht jedoch die Möglichkeit, dass ein einzelner Aufruf ähnlich schnell oder möglicherweise sogar schneller ist als ein entsprechender Aufruf auf der CPU.

Im Folgenden werden die einzelnen Komponenten des Konzepts detailliert beschrieben.

Hardwarebeschleunigtes Raytracing auf Grafikkarten kann auf verschiedene Weise umgesetzt werden. Der Hauptfokus dieser Technologie liegt traditionell auf dem fotorealistischen Rendering. Aus diesem Grund war die Nutzung der RT-Kerne auf die Rendering-Pipeline beschränkt. *Vulkan 1.2.162.0* hat diese Einschränkung mit der Einführung von Ray Queries aufgehoben und ermöglicht es, die RT-Kerne außerhalb der Fragment- und Vertex-Shader zu verwenden[1].

Ray Queries sind eine vereinfachte Form von Raytracing-Pipelines. Die Raytracing-Pipeline wird innerhalb der Rendering-Pipeline ausgeführt und bestehen aus mehreren Shadern, die beispielsweise Treffer oder Fehltreffer behandeln (Siehe Abbildung 3).

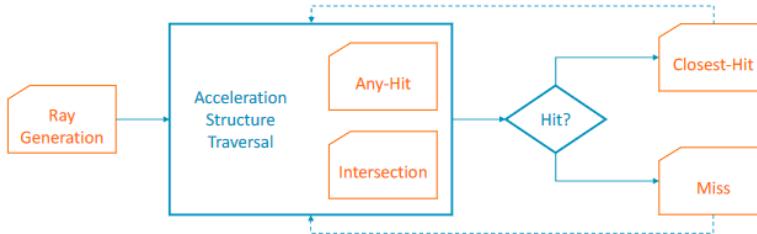


Abbildung 2: Fluss-ähnliches Diagramm einer Raytracing-Pipeline[30]

Im Gegensatz dazu lagern Ray Queries diese Shader-Funktionalität auf den Anwender aus, der innerhalb des Shaders entscheidet, was bei einem Treffer oder Fehltreffer geschieht (Siehe Abbildung 3).

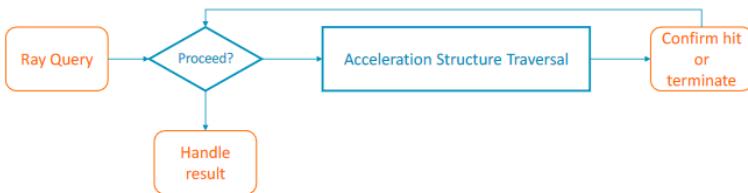


Abbildung 3: Fluss-ähnliches Diagramm einer Ray Query[30]

Ein weiteres Merkmal von Raytracing-Pipelines ist das Abprallen von Strahlen, das typischerweise Photonen nachahmt, die von einer Lichtquelle emittiert werden. Bei Ray Queries kann dieses Verhalten durch den Anwender optional implementiert werden, spielt jedoch in dieser Arbeit keine Rolle.

Neben den Strahlinformationen benötigen Ray Queries auch die Geometrie, gegen die

der Strahl projiziert werden soll. RT-Kerne verwenden eine spezielle Darstellungsform von Geometrie, sogenannte Acceleration Structures.

Acceleration Structures, die von Vulkan verwendet werden, bestehen aus zwei Komponenten: der Bottom-Level Acceleration Structure (BLAS) und der Top-Level Acceleration Structure (TLAS)[31]. Die BLAS enthält die tatsächliche Geometrie der 3D-Modelle der Szene. Die TLAS hingegen enthält Instanzen, also Transformationsinformationen eines 3D-Modells wie Position, Rotation und Skalierung. Eine häufige Optimierung in Game-Engines besteht darin, 3D-Modelle zu instanziieren, d. h., dieselbe Geometrie mehrmals mit unterschiedlichen Transformationsinformationen zu verwenden. Diese Instanzen referenzieren dieselben Geometriedaten, was dazu führt, dass die Geometrie nur einmal auf der GPU existieren muss, jedoch basierend auf den Instanzen mehrfach gerendert werden kann. Die TLAS ist eine Repräsentation dieser Instanzen, welche für die Nutzung durch Ray Queries optimiert ist. Abbildung 4 zeigt eine Visualisierung von BLAS und TLAS.

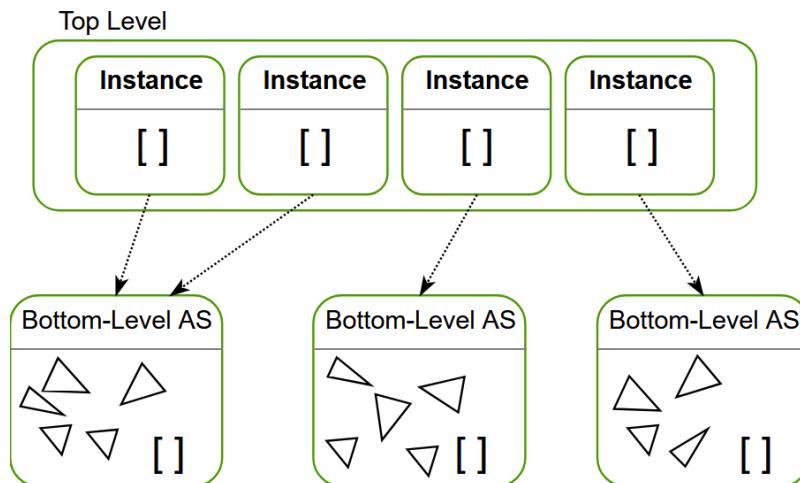


Abbildung 4: Übersicht einer Acceleration Structure[32]

Moderne GPUs bieten eine Vielzahl von Shadern, darunter auch Compute-Shader. Compute-Shader sind universelle Werkzeuge, die es ermöglichen, Berechnungen hochparallelisiert auf der GPU durchzuführen, ohne dass diese Rendering-basiert sein müssen. Ray Queries können in solchen Compute-Shadern ausgeführt werden.

Um die Acceleration Structures und die Strahlinformationen an die Ray Query weiterzugeben, wird ein Compute-Shader erstellt und mit den notwendigen Daten versorgt. Die Ray Query wird dann ausgeführt, und das Ergebnis wird an den Compute-Shader zurückgegeben. Dieser übermittelt das Ergebnis in Form eines booleschen Wahrheitswerts an die CPU und somit an den Aufrufer.

Zum Messen der Performance beider Implementierungen wird eine einfache *Stopwatch*-Klasse verwendet um die Zeit an zwei Punkten zu messen und voneinander subtrahieren. Die Differenz wird in Millisekunden angegeben.

Als Referenzimplementierung dient die Unreal Engine 5. Diese Engine enthält die ei-

gens entwickelte Physik-Engine *Chaos* und repräsentiert den aktuellen Stand der Technik. Aufgrund ihrer weiten Verbreitung in der Videospielbranche eignet sie sich ideal für einen Vergleich.

Um die beiden Ansätze vergleichen zu können, werden mehrere Testszenarien definiert, die aus einer variablen Anzahl von 3D-Modellen bestehen, die von Strahlen beschossen werden.

Um sicherzustellen, dass die Szenarien in beiden Implementierungen identisch sind, wird ein CSV-Datenformat definiert. Jede Zeile entspricht einem 3D-Modell sowie dessen Position, Rotation und Skalierung, dem Strahltyp (einzelne oder kegelförmig), Strahlursprung und -richtung. Für jedes Szenario wird eine separate CSV-Datei erstellt. Die Implementierung lädt beim Start die entsprechende CSV-Datei, führt das Szenario automatisch aus und speichert die Messwerte ebenfalls in einer CSV-Datei ab. Die Auswertung der Messdaten erfolgt mithilfe von Microsoft Excel.

Unreal Engine 5 bietet zudem die Möglichkeit die Treffererkennung basierend auf dem eigentlichen 3D-Modell durchzuführen anstatt mit Hitboxen. Daher werden die Testszenarien in der UE5 Implementation jeweils für Hitboxen sowie 3D-Modelle durchgeführt.

Testszenarien werden in UE5 mithilfe von selbstgeschriebenen Skripten erstellt und in CSV-Dateien exportiert.

3.1 Zusammenfassung

In modernen Game-Engines erfolgt die Treffererkennung typischerweise durch eine Funktion, die Strahlen in die Szene projiziert und auf Kollisionen prüft. Diese Berechnungen werden meist auf der CPU durchgeführt und nutzen vereinfachte Hitboxen, was jedoch zu Ungenauigkeiten wie False-Negatives oder False-Positives führen kann. Diese Probleme sind besonders in kompetitiven Spielen relevant, da sie das Spielerlebnis beeinträchtigen und zusätzlichen Entwicklungsaufwand erfordern. Eine vielversprechende Alternative ist die Verlagerung der Berechnungen auf die GPU, insbesondere durch den Einsatz von Raytracing-Kernen und Ray Queries, wie sie von der Vulkan-API bereitgestellt werden. In dieser Arbeit wird die Funktion *traceRay* auf der GPU implementiert, um die Leistung und Genauigkeit gegenüber einer CPU-basierten Referenzimplementierung in der Unreal Engine 5 zu vergleichen. Hierfür werden mehrere Testszenarien definiert, in denen die Berechnungszeiten der beiden Ansätze miteinander verglichen werden.

4 Prototypische Realisierung

Das Kapitel gliedert sich in zwei Bereiche.

Der erste Bereich behandelt die Implementierung des Konzepts innerhalb einer eigens entwickelten Prism Game-Engine und legt dabei den Schwerpunkt auf Vulkan-bezogene Themen. Hier werden spezifische technische Vulkan-Details und ihre Anwendung in der Engine beschrieben. Die Implementierung erfolgt mit dem offiziellen C++ Wrapper *Vulkan-Hpp*[33].

Der zweite und letzte Bereich konzentriert sich auf die CPU-basierte Implementierung des Konzepts in Unreal Engine 5. Dabei wird der Fokus auf die wesentlichen Unterschiede bei der Umsetzung gelegt.

4.1 Prism Engine-Implementation

Um die im Konzept angesprochene *traceRay*-Funktion zu implementieren, bedarf es mehrerer Schritte zuvor. Vulkan besteht aus einer Kernkomponente *Core* sowie optionale *Extensions*. Fordert der Benutzer eine *Extension* in Vulkan an, gibt der Grafiktreiber Rückmeldung ob diese implementiert ist und bereitgestellt werden kann. Ray Queries in Vulkan sind eine solche *Extension*. *Extensions* müssen bei Initialisierung des Logical Device angefragt werden (Siehe Listings 2-3).

Listing 2: Definition der Raytracing Extensions in VulkanRenderer.hpp

```
1   ...
2   std::vector<const char*> deviceExtensions = {
3     ...
4     VK_KHR_ACCELERATION_STRUCTURE_EXTENSION_NAME,
5     ...
6     VK_KHR_RAY_QUERY_EXTENSION_NAME,
7     VK_KHR_BUFFER_DEVICE_ADDRESS_EXTENSION_NAME
8   };
9   ...
```

Listing 3: Anfragen der Raytracing Extensions in VulkanRenderer.cpp

```
1   ...
2   vk::PhysicalDeviceFeatures deviceFeatures;
3   ...
4   vk::PhysicalDeviceAccelerationStructureFeaturesKHR
5   accelerationStructureFeatures;
6   accelerationStructureFeatures.accelerationStructure = VK_TRUE;
7   vk::PhysicalDeviceRayQueryFeaturesKHR rayQueryFeatures;
8   rayQueryFeatures.rayQuery = VK_TRUE;
9   rayQueryFeatures.pNext = &accelerationStructureFeatures;
10  vk::PhysicalDeviceBufferDeviceAddressFeatures
11  bufferDeviceAddressFeatures;
12  bufferDeviceAddressFeatures.bufferDeviceAddress = VK_TRUE;
13  bufferDeviceAddressFeatures.pNext = &rayQueryFeatures;
14  auto createInfo = vk::DeviceCreateInfo(
15    vk::DeviceCreateFlags(),
```

```

14     static_cast<uint32_t>(queueCreateInfos.size()) ,
15     queueCreateInfos.data()
16 );
17     createInfo.pEnabledFeatures = &deviceFeatures;
18     createInfo.pNext = &bufferDeviceAddressFeatures;
19     createInfo.enabledExtensionCount = static_cast<uint32_t>(
20         deviceExtensions.size());
21     createInfo.ppEnabledExtensionNames = deviceExtensions.data();
22     ...
23     logicalDevice = physicalDevice.createDeviceUnique(createInfo);
24     ...

```

Als nächstes wird der Compute-Shader geladen. Zunächst definiert die Funktion *createComputeDescriptorSetLayout* das Layout des Descriptor Sets für die Compute-Pipeline. Hierbei werden drei Binding erstellt: eines für die TLAS, eines für den Eingabe-Storage-Buffer und eines für den Ausgabe-Storage-Buffer. Diese Bindings sind essentiell, um später die entsprechenden Ressourcen mit der Compute-Pipeline zu verknüpfen. Siehe dazu Listing 4.

Listing 4: Erzeugen des *Descriptor Set Layouts* in VulkanRenderer.cpp

```

1 void VulkanRenderer::createComputeDescriptorsetLayout()
2 {
3     vk::DescriptorSetLayoutBinding tlasLayoutBinding;
4     tlasLayoutBinding.binding = 0;
5     tlasLayoutBinding.descriptorType = vk::DescriptorType::eAccelerationStructureKHR;
6     tlasLayoutBinding.descriptorCount = 1;
7     tlasLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eCompute;
8     tlasLayoutBinding.pImmutableSamplers = nullptr;
9     vk::DescriptorSetLayoutBinding inputStorageBufferLayoutBinding;
10    inputStorageBufferLayoutBinding.binding = 1;
11    inputStorageBufferLayoutBinding.descriptorType = vk::DescriptorType::eStorageBuffer;
12    inputStorageBufferLayoutBinding.descriptorCount = 1;
13    inputStorageBufferLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eCompute;
14    inputStorageBufferLayoutBinding.pImmutableSamplers = nullptr;
15    vk::DescriptorSetLayoutBinding outputStorageBufferLayoutBinding;
16    outputStorageBufferLayoutBinding.binding = 2;
17    outputStorageBufferLayoutBinding.descriptorType = vk::DescriptorType::eStorageBuffer;
18    outputStorageBufferLayoutBinding.descriptorCount = 1;
19    outputStorageBufferLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eCompute;
20    outputStorageBufferLayoutBinding.pImmutableSamplers = nullptr;
21    const std::array bindings = {
22        tlasLayoutBinding, inputStorageBufferLayoutBinding,
23        outputStorageBufferLayoutBinding
24    };
25    vk::DescriptorSetLayoutCreateInfo layoutInfo = {};
26    layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
27    layoutInfo.pBindings = bindings.data();
28    ...

```

```

28     computeDescriptorSetLayout = logicalDevice->createDescriptorSetLayout(
29         layoutInfo);
30 }
```

Im nächsten Schritt wird in der Funktion *createComputePipeline* die Compute-Pipeline selbst erstellt. Dazu wird zunächst das SPIR-V Shader-Modul für den Compute-Shader geladen und konfiguriert. Anschließend erfolgt die Erstellung des Pipeline-Layouts, das dass zuvor definierte Descriptor Set Layout einbezieht. Danach wird die Compute-Pipeline aufgebaut und entsprechend konfiguriert, um für die Berechnungen bereit zu sein. Siehe dazu Listing 12 im Anhang.

Daraufhin erstellt die Funktion *createComputeStorageBuffers* die Buffer, die innerhalb der Compute-Pipeline verwendet werden (Siehe Listing 5). Es werden sowohl ein Eingabebuffer als auch ein Ausgabebuffer angelegt.

Listing 5: Erstellen der Shader Storage Buffer Objects in VulkanRenderer.cpp

```

1 void VulkanRenderer::createComputeStorageBuffers()
2 {
3     constexpr auto inputBufferSize = sizeof(ComputeInputBufferObject);
4     createBuffer(inputBufferSize, vk::BufferUsageFlagBits::eStorageBuffer,
5                 vk::MemoryPropertyFlagBits::eHostVisible |
6                 vk::MemoryPropertyFlagBits::eHostCoherent, inputComputeBuffer,
7                 inputComputeBufferMemory);
8     constexpr auto outputBufferSize = sizeof(ComputeOutputBufferObject);
9     createBuffer(outputBufferSize, vk::BufferUsageFlagBits::eStorageBuffer,
10                 vk::MemoryPropertyFlagBits::eHostVisible |
11                 vk::MemoryPropertyFlagBits::eHostCoherent, outputComputeBuffer,
12                 outputComputeBufferMemory);
13 }
```

Die *Memory Flags*, die in dieser Funktion verwendet werden, sind *eHostVisible* und *eHostCoherent*. Sie definieren wichtige Eigenschaften des zugewiesenen Speichers:

- **vk::MemoryPropertyFlagBits::eHostVisible**: Dieses Flag stellt sicher, dass der Speicherbereich von der CPU aus sichtbar ist, was bedeutet, dass die CPU direkt auf den Speicher zugreifen und Daten lesen oder schreiben kann. Dies ist notwendig, um Daten in den Buffer zu laden (im Fall des *Eingabebuffers*) oder um Ergebnisse aus dem Buffer auszulesen (im Fall des Ausgabebuffers).[34]
- **vk::MemoryPropertyFlagBits::eHostCoherent**: Dieses Flag garantiert, dass Änderungen am Speicherbereich sofort für beide Seiten – sowohl die CPU als auch die GPU – sichtbar sind. Ohne dieses Flag müsste man explizit Speicherbereiche *flushen* (d.h. sicherstellen, dass alle Änderungen von der CPU zur GPU übertragen werden) oder *invalidate* (d.h. sicherstellen, dass die CPU die aktuellen Daten von der GPU sieht). Mit *eHostCoherent* entfällt dieser zusätzliche Aufwand, da die Kohärenz zwischen Host- und Device-Memory automatisch gewährleistet ist.[34]

Beide Buffer werden zudem mit der Größe der Structs *ComputeInputBufferObject* (Anhang, Listing 13) und *ComputeOutputBufferObject* (Anhang, Listing 14) erzeugt. Beide

Shader Storage Buffer Objects geben das Speicher-Layout bzw. Speichergröße für Vulkan vor.

Anschließend wird in der Funktion *createComputeDescriptorPool* ein Descriptor Pool erstellt, aus dem die Descriptor Sets für die Compute-Pipeline allokiert werden. Dieser Pool beinhaltet Ressourcen für die TLAS und SSBOs. Siehe dazu Listing 15 im Anhang.

Die Funktion *createComputeDescriptorSet* allokiert daraufhin das Descriptor Set aus dem zuvor erstellten Pool und befüllt es mit den erforderlichen Informationen, darunter der TLAS und die SSBOs. Diese Descriptor Sets sind für die Compute-Pipeline unerlässlich, da sie den Zugriff auf die entsprechenden Daten während der Ausführung der Raytracing-Berechnungen ermöglichen. Siehe dazu im Anhang Listing 16.

Zuletzt erstellt die Funktion *createComputeCommandPool* einen Command Pool, der für die Compute-Pipeline verwendet wird. Dieser Pool dient der Allokation von Command Buffers, welche die Ausführung der Compute-Operationen auf der GPU steuern. Zusammengefasst sorgen diese Funktionen dafür, dass die grundlegenden Elemente einer Compute-Pipeline in Vulkan eingerichtet werden, wodurch die Durchführung von Raytracing-Berechnungen auf der GPU ermöglicht wird. Siehe Listing 17 im Anhang.

Nun sind alle Voraussetzungen für das Ausführen des Compute-Shaders erfüllt. Bevor jedoch die eigentliche *traceRay* Funktion implementiert werden kann, bedarf es der Erzeugung der BLAS und TLAS.

Die *Bottom-Level Acceleration-Structures* werden auf Basis der für Treffererkennung relevanten 3D-Modelldaten erzeugt (Siehe Listing 7). Zu diesem Zweck müssen diese Daten zuerst in ein Zwischenformat gespeichert werden. Dies erledigt die *collectAccelerationStructureGeometries* Funktion in Listing 6.

Listing 6: Sammeln der 3D-Modellgeometrie in VulkanRenderer.cpp

```

1 std::vector<BottomLevelAccelerationStructureInput> VulkanRenderer::
2     collectAccelerationStructureGeometries() const
3 {
4     std::vector<BottomLevelAccelerationStructureInput> blasInputs;
5     ...
6     for (const auto& vulkanMesh : meshes)
7     {
8         const auto meshAsset = assetManager->getAsset<Assets::
9             StaticMeshAsset>(vulkanMesh->getMeshAssetName());
10        ...
11        vk::AccelerationStructureGeometryTrianglesDataKHR triangleData;
12        triangleData.vertexFormat = vk::Format::eR32G32B32Sfloat;
13        triangleData.vertexData = getBufferDeviceAddress(vulkanMesh->
14            getVertexBuffer().getBuffer());
15        triangleData.vertexStride = sizeof(Vertex);
16        triangleData.maxVertex = static_cast<uint32_t>(meshAsset->
17            getVertices().size()) - 1;
18        triangleData.indexType = vk::IndexType::eUInt32;
19        triangleData.indexData = getBufferDeviceAddress(vulkanMesh->
20            getIndexBuffer().getBuffer());
21        triangleData.transformData = nullptr;
22        vk::AccelerationStructureGeometryKHR geometry;
23        geometry.setGeometryType(vk::GeometryTypeKHR::eTriangles);

```

```

19     geometry.setFlags(vk::GeometryFlagBitsKHR::eOpaque);
20     geometry.geometry.setTriangles(triangleData);
21     const uint32_t primitiveCount = static_cast<uint32_t>(meshAsset->
22     getIndices().size() / 3);
23     vk::AccelerationStructureBuildRangeInfoKHR buildRangeInfo;
24     buildRangeInfo.setPrimitiveCount(primitiveCount);
25     buildRangeInfo.setPrimitiveOffset(0);
26     buildRangeInfo.setFirstVertex(0);
27     buildRangeInfo.setTransformOffset(0);
28     BottomLevelAccelerationStructureInput blasInput;
29     blasInput.geometry = geometry;
30     blasInput.rangeInfo = buildRangeInfo;
31     blasInput.meshInstanceIds = vulkanMesh->getMeshIds();
32     blasInputs.emplace_back(blasInput);
33 }
34 }
```

Anschließend werden die Speicheranforderungen für jede BLAS berechnet, um die notwendige Größe für den temporären Speicher (Scratch Buffer) zu ermitteln, der während des Aufbaus benötigt wird.

Sobald die benötigten Speichergrößen feststehen, wird ein Scratch Buffer erstellt, um den Aufbau der BLAS zu unterstützen. Daraufhin werden für jede gesammelte Geometrie leere BLAS-Strukturen im GPU-Speicher reserviert. Der eigentliche Aufbau der BLAS erfolgt durch das Laden der Geometrien in diese Strukturen. Hierfür wird ein Command Buffer verwendet, um die entsprechenden Operationen auf der GPU auszuführen. Nach der Ausführung des Command Buffers wird der Scratch Buffer wieder freigegeben, um somit den belegten Speicherplatz zu räumen. Abschließend werden die erstellten BLAS zurückgegeben, die dann für die Erzeugung der TLAS verwendet werden können.

Relevant ist hierbei das Flag *ePreferFastTrace*. Alternativ zu diesem gibt es unter anderem *ePreferFastBuild*. Dies signalisiert Vulkan entweder die Optimierung der BLAS für schnelleres Raytracing oder schnelleres Erstellen der BLAS, jeweils zum Nachteil des anderen. Ebenfalls wichtig ist der *BLAS build mode eBuild*. Anstatt die BLAS komplett neu zu erzeugen kann man diese auch update falls sich nur geringfügige Änderungen seit der letzten BLAS Erstellung ergeben haben.

Listing 7: Erzeugung der Bottom-Level Acceleration Structures in VulkanRenderer.cpp

```

1 std::vector<AccelerationStructure> VulkanRenderer::
2     buildBottomLevelAccelerationStructures()
3 {
4     ...
5     auto blasInputs = collectAccelerationStructureGeometries();
6     ...
7     vk::DeviceSize totalAccelStructureSizeBytes = 0;
8     vk::DeviceSize maxScratchSizeBytes = 0;
9     std::vector<BuildAccelerationStructure> buildAccelerationStructures;
10    buildAccelerationStructures.reserve(blasInputs.size());
11    ...
12    for (auto& [geometry, rangeInfo, meshInstanceIds] : blasInputs)
13    {
```

```

13     vk::AccelerationStructureBuildGeometryInfoKHR buildGeometryInfo{};
14     buildGeometryInfo.type = vk::AccelerationStructureTypeKHR::eBottomLevel;
15     buildGeometryInfo.flags = vk::BuildAccelerationStructureFlagBitsKHR::ePreferFastTrace;
16     buildGeometryInfo.mode = vk::BuildAccelerationStructureModeKHR::eBuild;
17     buildGeometryInfo.geometryCount = 1;
18     buildGeometryInfo.pGeometries = &geometry;
19     vk::AccelerationStructureBuildSizesInfoKHR sizeInfo{};
20     logicalDevice->getAccelerationStructureBuildSizesKHR(
21         vk::AccelerationStructureBuildTypeKHR::eDevice,
22         &buildGeometryInfo,
23         &rangeInfo.primitiveCount,
24         &sizeInfo
25     );
26     BuildAccelerationStructure buildAccelerationStructure{
27         buildGeometryInfo, sizeInfo, rangeInfo, {}, {}, {},
28         meshInstanceIds
29     };
30     buildAccelerationStructures.emplace_back(buildAccelerationStructure);
31     totalAccelStructureSizeBytes += sizeInfo.accelerationStructureSize;
32     maxScratchSizeBytes = std::max(maxScratchSizeBytes, sizeInfo.
33     buildScratchSize);
34 }
35 ...
36 vk::Buffer scratchBuffer;
37 vk::DeviceMemory scratchBufferMemory;
38 createBuffer(maxScratchSizeBytes,
39 vk::BufferUsageFlagBits::eShaderDeviceAddress | vk::BufferUsageFlagBits::eStorageBuffer,
40 vk::MemoryPropertyFlagBits::eDeviceLocal, scratchBuffer,
41 scratchBufferMemory);
42 vk::DeviceAddress scratchAddress = getBufferDeviceAddress(scratchBuffer);
43 std::vector<vk::AccelerationStructureBuildGeometryInfoKHR>
44 buildGeometryInfos;
45 buildGeometryInfos.reserve(buildAccelerationStructures.size());
46 std::vector<vk::AccelerationStructureBuildRangeInfoKHR> buildRangeInfos;
47 buildRangeInfos.reserve(buildAccelerationStructures.size());
48 std::vector<AccelerationStructure> blases;
49 blases.reserve(buildAccelerationStructures.size());
50 ...
51 for (auto& [buildInfo, buildSizesInfo, rangeInfo, accelerationStructure]
52 : buildAccelerationStructures)
53 {
54     vk::AccelerationStructureCreateInfoKHR createInfo{};
55     createInfo.type = vk::AccelerationStructureTypeKHR::eBottomLevel;
56     createInfo.size = buildSizesInfo.accelerationStructureSize;
57     createBuffer(createInfo.size,
58         vk::BufferUsageFlagBits::eAccelerationStructureStorageKHR |
59         vk::BufferUsageFlagBits::eShaderDeviceAddress, vk::
60         MemoryPropertyFlagBits::eDeviceLocal,

```

```

55     accelerationStructure.buffer,
56     accelerationStructure.memory);
57     createInfo.buffer = accelerationStructure.buffer;
58     ...
59     accelerationStructure.accelerationStructureKHR = logicalDevice->
createAccelerationStructureKHR(createInfo);
60     ...
61     buildInfo.dstAccelerationStructure = accelerationStructure.
62     accelerationStructureKHR;
63     buildInfo.scratchData.deviceAddress = scratchAddress;
64     buildGeometryInfos.emplace_back(buildInfo);
65     buildRangeInfos.emplace_back(rangeInfo);
66     blases.emplace_back(accelerationStructure);
67 }
68 ...
69 logicalDevice->resetCommandPool(computeCommandPool);
70 vk::CommandBufferAllocateInfo allocInfo = {};
71 allocInfo.level = vk::CommandBufferLevel::ePrimary;
72 allocInfo.commandPool = computeCommandPool;
73 allocInfo.commandBufferCount = 1;
74 vk::CommandBuffer commandBuffer;
75 ...
76 commandBuffer = logicalDevice->allocateCommandBuffers(allocInfo)[0];
77 ...
78 vk::CommandBufferBeginInfo beginInfo = {};
79 beginInfo.flags = vk::CommandBufferUsageFlagBits::eOneTimeSubmit;
80 commandBuffer.begin(beginInfo);
81 commandBuffer.buildAccelerationStructuresKHR(buildGeometryInfos,
buildRangeInfos.data());
82 commandBuffer.end();
83 vk::SubmitInfo submitInfo = {};
84 submitInfo.commandBufferCount = 1;
85 submitInfo.pCommandBuffers = &commandBuffer;
86 computeQueue.submit(submitInfo, nullptr);
87 computeQueue.waitIdle();
88 ...
89 logicalDevice->destroyBuffer(scratchBuffer);
90 logicalDevice->freeMemory(scratchBufferMemory);
91 ...
92 return blases;
93 }
```

Die Top-Level Acceleration Structure dient als übergeordnete Datenstruktur, die Referenzen auf mehrere BLAS enthält, wodurch eine hierarchische Organisation der Szenengeometrien ermöglicht wird.

Der Prozess beginnt mit der Erstellung einer Liste von Instanzen, wobei jede Instanz eine Referenz auf genau eine BLAS darstellt. Jede Instanz-TLAS enthält unter anderem die Transformationsmatrix der 3D-Modellinstanz sowie die Speicheradresse der BLAS.

Im nächsten Schritt wird ein Command Buffer initialisiert, der zur Ausführung der notwendigen GPU-Operationen dient. Es wird ein temporärer Staging Buffer erstellt, in dem die Instanzen vorübergehend im Host-Speicher abgelegt werden. Dieser Staging Buffer wird dann in einen Buffer auf der GPU kopiert, der die Instanzen für den weiteren Aufbau

der TLAS bereitstellt.

Nachdem die Instanzen in den GPU-Speicher kopiert wurden, wird eine Memory Barrier gesetzt, um sicherzustellen, dass alle Speicheroperationen abgeschlossen sind, bevor die TLAS aufgebaut wird. Die Funktion berechnet dann die notwendigen Speichergrößen für die TLAS und allokiert den benötigten Speicher auf der GPU. Zusätzlich wird ein Scratch Buffer erstellt, der temporär für den Aufbau der TLAS verwendet wird.

Schließlich werden die TLAS durch einen Aufruf des Command Buffers aufgebaut, der die Instanzen verarbeitet und die TLAS erstellt. Nach Abschluss des Aufbaus wird der Command Buffer beendet, und die GPU führt die entsprechenden Operationen aus. Nachdem der Aufbau abgeschlossen ist, wird auf das Beenden der GPU-Operationen gewartet, und die temporären Buffer werden freigegeben. Siehe dazu Listing 8.

Listing 8: Erzeugung der Top-Level Acceleration Structure in VulkanRenderer.cpp

```

1 AccelerationStructure VulkanRenderer::buildTopLevelAccelerationStructure(
2     const std::vector<AccelerationStructure>& blases)
3 {
4     std::vector<vk::AccelerationStructureInstanceKHR> instances;
5     instances.reserve(blases.size());
6     const auto activeScene = sceneManager->getActiveScene();
7     const auto staticMeshComponents = activeScene->getComponents<Core::
8         StaticMeshComponent>();
9     for (const auto& blas : blases)
10    {
11        auto matchingComponents = staticMeshComponents
12            | std::views::filter([](const auto& staticMeshComponent)
13            {
14                auto meshId = staticMeshComponent->getStaticMesh()->getMeshId()
15            ;
16                return std::ranges::any_of(blas.meshInstanceIds, [&](const auto
17                    & instanceId)
18                    {
19                        return instanceId == meshId && staticMeshComponent->
20                            isCollidable();
21                    });
22            });
23        for (auto staticMeshComponent : matchingComponents)
24        {
25            vk::AccelerationStructureInstanceKHR asInstance{};
26            asInstance.transform = glmMat4ToVkTransformMatrix(
27                staticMeshComponent->getAbsoluteTransform().toMatrix());
28            asInstance.instanceCustomIndex = 0;
29            asInstance.accelerationStructureReference =
30                getBlasBufferAddress(blas);
31            asInstance.setFlags(vk::GeometryInstanceFlagBitsKHR::
32                eTriangleCullDisable);
33            asInstance.mask = 0xFF;
34            asInstance.instanceShaderBindingTableRecordOffset = 0;
35            instances.emplace_back(asInstance);
36        }
37    }
38    const auto instanceCount = static_cast<uint32_t>(instances.size());
39    logicalDevice->resetCommandPool(computeCommandPool);

```

```

34     vk::CommandBufferAllocateInfo allocInfo = {};
35     allocInfo.level = vk::CommandBufferLevel::ePrimary;
36     allocInfo.commandPool = computeCommandPool;
37     allocInfo.commandBufferCount = 1;
38     vk::CommandBuffer commandBuffer;
39     ...
40     commandBuffer = logicalDevice->allocateCommandBuffers(allocInfo)[0];
41     ...
42     vk::CommandBufferBeginInfo beginInfo = {};
43     beginInfo.flags = vk::CommandBufferUsageFlagBits::eOneTimeSubmit;
44     commandBuffer.begin(beginInfo);
45     vk::Buffer instanceStagingBuffer;
46     vk::DeviceMemory instanceStagingBufferMemory;
47     const auto stagingBufferSize = instances.size() * sizeof(vk::
48     AccelerationStructureInstanceKHR);
49     createBuffer(stagingBufferSize, vk::BufferUsageFlagBits::eTransferSrc,
50     vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::
51     eHostCoherent,
52     instanceStagingBuffer, instanceStagingBufferMemory);
53     void* data = logicalDevice->mapMemory(instanceStagingBufferMemory, 0,
54     stagingBufferSize);
55     memcpy(data, instances.data(), stagingBufferSize);
56     logicalDevice->unmapMemory(instanceStagingBufferMemory);
57     vk::Buffer instanceBuffer;
58     vk::DeviceMemory instanceBufferMemory;
59     createBuffer(stagingBufferSize,
60     vk::BufferUsageFlagBits::eShaderDeviceAddress |
61     vk::BufferUsageFlagBits::eAccelerationStructureBuildInputReadOnlyKHR |
62     vk::BufferUsageFlagBits::eTransferDst,
63     vk::MemoryPropertyFlagBits::eDeviceLocal,
64     instanceBuffer, instanceBufferMemory);
65     copyBuffer(instanceStagingBuffer, instanceBuffer, stagingBufferSize,
66     commandBuffer);
67     vk::BufferMemoryBarrier instanceBufferMemoryBarrier{};
68     instanceBufferMemoryBarrier.srcAccessMask = vk::AccessFlagBits::
69     eTransferWrite;
70     instanceBufferMemoryBarrier.dstAccessMask = vk::AccessFlagBits::
71     eAccelerationStructureWriteKHR;
72     instanceBufferMemoryBarrier.srcQueueFamilyIndex =
73     VK_QUEUE_FAMILY_IGNORED;
74     instanceBufferMemoryBarrier.dstQueueFamilyIndex =
75     VK_QUEUE_FAMILY_IGNORED;
76     instanceBufferMemoryBarrier.buffer = instanceBuffer;
77     instanceBufferMemoryBarrier.size = stagingBufferSize;
    vk::PipelineStageFlags srcStageMask = vk::PipelineStageFlagBits::
    eTransfer;
    vk::PipelineStageFlags dstStageMask = vk::PipelineStageFlagBits::
    eAccelerationStructureBuildKHR;
    vk::DependencyFlags dependencyFlags = {};
    commandBuffer.pipelineBarrier(srcStageMask, dstStageMask,
    dependencyFlags, nullptr, instanceBufferMemoryBarrier,
    nullptr);
    auto instanceBufferAddress = getBufferDeviceAddress(instanceBuffer);
    vk::AccelerationStructureGeometryInstancesDataKHR instancesData{};
    instancesData.data.deviceAddress = instanceBufferAddress;

```

```

78     const vk::AccelerationStructureGeometryKHR tlasGeometry{vk::
79     GeometryTypeKHR::eInstances, instancesData};
80     vk::AccelerationStructureBuildGeometryInfoKHR buildGeometryInfo{};
81     buildGeometryInfo.flags = vk::BuildAccelerationStructureFlagBitsKHR::
82     ePreferFastTrace;
83     buildGeometryInfo.geometryCount = 1;
84     buildGeometryInfo.pGeometries = &tlasGeometry;
85     buildGeometryInfo.mode = vk::BuildAccelerationStructureModeKHR::eBuild;
86     buildGeometryInfo.type = vk::AccelerationStructureTypeKHR::eTopLevel;
87     buildGeometryInfo.srcAccelerationStructure = nullptr;
88     vk::AccelerationStructureBuildSizesInfoKHR buildSizesInfo{};
89     logicalDevice->getAccelerationStructureBuildSizesKHR(
90         vk::AccelerationStructureBuildTypeKHR::eDevice,
91         &buildGeometryInfo,
92         &instanceCount,
93         &buildSizesInfo
94     );
95     vk::AccelerationStructureCreateInfoKHR createInfo{};
96     createInfo.size = buildSizesInfo.accelerationStructureSize;
97     createInfo.type = vk::AccelerationStructureTypeKHR::eTopLevel;
98     AccelerationStructure tlas;
99     createBuffer(createInfo.size,
100      vk::BufferUsageFlagBits::eAccelerationStructureStorageKHR |
101      vk::BufferUsageFlagBits::eShaderDeviceAddress, vk::
102      MemoryPropertyFlagBits::eDeviceLocal,
103      tlas.buffer, tlas.memory);
104     createInfo.buffer = tlas.buffer;
105     tlas.accelerationStructureKHR = logicalDevice->
106     createAccelerationStructureKHR(createInfo);
107     vk::Buffer scratchBuffer;
108     vk::DeviceMemory scratchBufferMemory;
109     createBuffer(buildSizesInfo.buildScratchSize,
110      vk::BufferUsageFlagBits::eStorageBuffer | vk::BufferUsageFlagBits::
111      eShaderDeviceAddress,
112      vk::MemoryPropertyFlagBits::eDeviceLocal, scratchBuffer,
113      scratchBufferMemory);
114     const auto scratchBufferAddress = getBufferDeviceAddress(scratchBuffer)
115     ;
116     buildGeometryInfo.dstAccelerationStructure = tlas.
117     accelerationStructureKHR;
118     buildGeometryInfo.scratchData.deviceAddress = scratchBufferAddress;
119     vk::AccelerationStructureBuildRangeInfoKHR buildRangeInfo{instanceCount
120     , 0, 0, 0};
121     commandBuffer.buildAccelerationStructuresKHR({buildGeometryInfo}, &
122     buildRangeInfo);
123     commandBuffer.end();
124     vk::SubmitInfo submitInfo = {};
125     submitInfo.commandBufferCount = 1;
126     submitInfo.pCommandBuffers = &commandBuffer;
127     computeQueue.submit(submitInfo, nullptr);
128     // Wait for compute queue to finish
129     computeQueue.waitIdle();
130     logicalDevice->destroyBuffer(scratchBuffer);
131     logicalDevice->freeMemory(scratchBufferMemory);
132     logicalDevice->destroyBuffer(instanceStagingBuffer);

```

```
123     logicalDevice->freeMemory(instanceStagingBufferMemory);
124     logicalDevice->destroyBuffer(instanceBuffer);
125     logicalDevice->freeMemory(instanceBufferMemory);
126     ...
127     return tlas;
128 }
```

Die Funktion *traceRays* in Listing 9 dient der Durchführung der Raytracing-Operation innerhalb einer Vulkan-basierten Pipeline. Sie nimmt als Eingabe eine Liste an Structs, bestehend aus dem Strahlursprung, der Richtung eines Strahls sowie minimale und maximale Länge für die Strahlverfolgung. Der Hauptzweck dieser Funktion besteht darin, festzustellen, ob die gegebenen Strahlen ein Objekt in der Szene trifft.

Zu Beginn der Funktion wird eine Bereinigung der vorherigen Raytracing-Daten durchgeführt, um sicherzustellen, dass keine veralteten Informationen den aktuellen Prozess beeinflussen.

Der nächste Schritt umfasst den Aufbau der notwendigen Beschleunigungsstrukturen. Zunächst werden die BLAS durch Aufrufen der zuvor beschriebenen Funktion *buildBottomLevelAccelerationStructures* erstellt. Diese Strukturen repräsentieren die grundlegenden geometrischen Daten der Szene. Daraufhin werden die TLAS mit der ebenfalls beschriebenen Funktion *buildTopLevelAccelerationStructure* generiert, welche die BLAS referenziert und eine hierarchische Organisation der Szene ermöglicht.

Anschließend werden SSBOs erstellt und mit den Parametern der Strahlen befüllt. Falls der Aufrufer mehr Strahlen projizieren will als der SSBO-Speicher fassen kann, wird dieser vergrößert.

Nachdem die Beschleunigungsstrukturen erstellt wurden und die SSBOs befüllt sind, wird das Descriptor Set für die Compute-Pipeline aktualisiert, um die TLAS und ggf. die neuen SSBOs einzubinden. Dies ermöglicht es der GPU, während der Raytracing-Operation effizient auf die Szenedaten zuzugreifen. Zudem werden die SSBOs in den Grafikspeicher kopiert, sodass der Compute-Shader auf die Daten zugreifen kann.

Ein Command Buffer wird initialisiert und konfiguriert, um die Compute-Pipeline auszuführen. Innerhalb dieses Buffers wird der entsprechende Compute-Shader gebunden, der für die Raytracing-Operation verantwortlich ist. Um von möglichst von der Multithreading-Fähigkeit der Grafikkarte zu profitieren wird auf Basis der Anzahl der zu projizierenden Strahlen die Anzahl der *Work Groups*, also der Arbeitspakete die auf den Compute Units¹⁵ der GPU verarbeit werden, definiert. Eine *Work Group* besteht aus 16 *Work Items*, welche jeweils einen Shader-Aufruf darstellen.

Durch den Aufruf von *dispatch* werden die *Work Groups* ausgeführt.

Nach der Ausführung des Command Buffers und dem Abschluss aller GPU-Operationen werden die Ergebnisse aus dem GPU-Speicher zurück in den Host-Speicher übertragen. Die Ausgabe-SSBOs werden ausgelesen, um festzustellen, ob die Strahlen ein Objekt getroffen haben. Die Ergebnisse werden anschließend an den Funktionsaufrufer zurückgegeben.

¹⁵Prozessoren zur Verarbeitung von Compute Shadern

Listing 9: *traceRay* Funktion in VulkanRenderer.cpp

```

1 std::vector<HitDetection::RayHitInfo> VulkanRenderer::traceRays(const std::
2   vector<HitDetection::RayInfo>& rayInfos)
3 {
4     std::vector<HitDetection::RayHitInfo> results;
5     results.reserve(rayInfos.size());
6     ...
7     cleanupRaytracingData();
8     ...
9     bottomLevelAccelerationStructures =
10    buildBottomLevelAccelerationStructures();
11    ...
12    const vk::WriteDescriptorSetAccelerationStructureKHR
13    tlasDescriptorWrite{
14        1, &topLevelAccelerationStructure.accelerationStructureKHR
15    };
16
17    // update compute input buffer
18    std::vector<HitDetection::ComputeInputBufferObject> inputBufferObjects;
19    inputBufferObjects.reserve(rayInfos.size());
20    for (size_t i = 0; i < rayInfos.size(); ++i)
21    {
22        const auto& rayInfo = rayInfos[i];
23        auto rayOriginInverted = rayInfo.getOrigin();
24        rayOriginInverted.y = -rayOriginInverted.y; // Invert Y axis for
25        Vulkan
26        inputBufferObjects.emplace_back(rayOriginInverted, rayInfo.
27            getDirection(), rayInfo.getMinDistance(),
28            rayInfo.getMaxDistance(), i);
29    }
30    ...
31    // If necessary, recreate the compute storage buffers
32    createComputeStorageBuffers(inputBufferObjects.size(), rayInfos.size(),
33        false);
34    // We have a new TLAS so we need the descriptorset to know about it.
35    Otherwise the old one will be used again.
36    vk::WriteDescriptorSet accelerationStructureDescriptorWrite{};
37    accelerationStructureDescriptorWrite.dstSet = computeDescriptorSet;
38    accelerationStructureDescriptorWrite.dstBinding = 0;
39    accelerationStructureDescriptorWrite.dstArrayElement = 0;
40    accelerationStructureDescriptorWrite.descriptorType = vk::
41        DescriptorType::eAccelerationStructureKHR;
42    accelerationStructureDescriptorWrite.descriptorCount = 1;
43    accelerationStructureDescriptorWrite.pNext = &tlasDescriptorWrite;
44    const vk::DescriptorBufferInfo inputBufferInfo{inputComputeBuffer, 0,
45        sizeof(HitDetection::ComputeInputBufferObject)};
46    vk::WriteDescriptorSet inputDescriptorWrite = {};
47    inputDescriptorWrite.dstSet = computeDescriptorSet;
48    inputDescriptorWrite.dstBinding = 1;
49    inputDescriptorWrite.dstArrayElement = 0;
50    inputDescriptorWrite.descriptorType = vk::DescriptorType::
51        eStorageBuffer;

```

```

44     inputDescriptorWrite.descriptorCount = 1;
45     inputDescriptorWrite.pBufferInfo = &inputBufferInfo;
46     const vk::DescriptorBufferInfo outputBufferInfo{outputComputeBuffer, 0,
47         sizeof(HitDetection::ComputeOutputBufferObject)};
48     vk::WriteDescriptorSet outputDescriptorWrite = {};
49     outputDescriptorWrite.dstSet = computeDescriptorSet;
50     outputDescriptorWrite.dstBinding = 2;
51     outputDescriptorWrite.dstArrayElement = 0;
52     outputDescriptorWrite.descriptorType = vk::DescriptorType::eStorageBuffer;
53     outputDescriptorWrite.descriptorCount = 1;
54     outputDescriptorWrite.pBufferInfo = &outputBufferInfo;
55     logicalDevice->updateDescriptorSets({
56         accelerationStructureDescriptorWrite, inputDescriptorWrite,
57         outputDescriptorWrite
58     }, nullptr);
59     const vk::DeviceSize bufferSize = sizeof(HitDetection::ComputeInputBufferObject) * inputBufferObjects.size();
60     void* data = logicalDevice->mapMemory(inputComputeBufferMemory, 0,
61     bufferSize);
62     memcpy(data, inputBufferObjects.data(), bufferSize);
63     logicalDevice->unmapMemory(inputComputeBufferMemory);
64     ...
65     logicalDevice->resetCommandPool(computeCommandPool);
66     vk::CommandBufferAllocateInfo allocInfo = {};
67     allocInfo.level = vk::CommandBufferLevel::ePrimary;
68     allocInfo.commandPool = computeCommandPool;
69     allocInfo.commandBufferCount = 1;
70     vk::CommandBuffer commandBuffer;
71     ...
72     commandBuffer = logicalDevice->allocateCommandBuffers(allocInfo)[0];
73     ...
74     vk::CommandBufferBeginInfo beginInfo = {};
75     beginInfo.flags = vk::CommandBufferUsageFlagBits::eOneTimeSubmit;
76     commandBuffer.begin(beginInfo);
77     commandBuffer.bindPipeline(vk::PipelineBindPoint::eCompute,
78         computePipeline);
79     commandBuffer.bindDescriptorSets(vk::PipelineBindPoint::eCompute,
80         computePipelineLayout, 0, 1,
81         &computeDescriptorSet, 0, nullptr);
82     uint32_t workgroupCountX = (rayInfos.size() + 15) / 16;
83     commandBuffer.dispatch(workgroupCountX, 1, 1);
84     commandBuffer.end();
85     ...
86     vk::SubmitInfo submitInfo = {};
87     submitInfo.commandBufferCount = 1;
88     submitInfo.pCommandBuffers = &commandBuffer;
89     ...
90     const vk::DeviceSize outputBufferSize = sizeof(HitDetection::ComputeOutputBufferObject) * rayInfos.size();
91     std::vector<HitDetection::ComputeOutputBufferObject> outputBuffers;

```

```

92     outputBuffers.resize(rayInfos.size());
93     const void* dataRead = logicalDevice->mapMemory(
94         outputComputeBufferMemory, 0, outputBufferSize);
95     memcpy(outputBuffers.data(), dataRead, outputBufferSize);
96     logicalDevice->unmapMemory(outputComputeBufferMemory);
97     ...
98 }

---



```

4.2 Unreal Engine 5 Implementation

Anders als die Prism Engine Implementierung fällt die der UE5 deutlich kleiner aus. Listing 10 zeigt die *ExecuteRaytraces* Funktion welche die Strahlinformationen zuallererst sortiert und anschließend wird das Array durchlaufen. Je nach Szenario-Typ wird entweder ein einzelner *ExecuteRaytrace* (Siehe Listing 11) Funktionsaufruf durchgeführt und die Zeit gemessen oder die Strahlen in Gruppen eingesortiert und anschließend mithilfe einer *ParallelFor*-Schleife[35] parallelisiert auf mehreren CPU-Kernen ausgeführt. Letztere ahmt die *Work Groups* aus Vulkan nach. Schlussendlich werden die Messwerte an den Aufrufer zurückgegeben.

Listing 10: *ExecuteRaytraces* Funktion in RTRaytracingSubsystem.cpp

```

1 TArray<double> URTRaytracingSubsystem::ExecuteRaytraces(TArray<
    URaytraceInfo*> RaytraceInfos)
2 {
3     TArray<double> Readings{};
4     int32 LastGroup = -1;
5     //Sort the raytrace infos by ascending group
6     Algo::Sort(RaytraceInfos, [] (const URaytraceInfo* A, const
    URaytraceInfo* B)
7     {
8         return A->GetRaytracingGroup() < B->GetRaytracingGroup();
9     });
10    TArray<URaytraceInfo*> GroupedRaytraceInfos{};
11    for (const auto& Info : RaytraceInfos)
12    {
13        if (Info->GetRaytracingType() == ERTRaytracingType::Static)
14        {
15            const auto Time = ExecuteRaytrace(Info, false);
16            Readings.Add(Time);
17        } else if (Info->GetRaytracingType() == ERTRaytracingType::
    StaticShotgun)
18        {
19            if (LastGroup != Info->GetRaytracingGroup() &&
GroupedRaytraceInfos.Num() > 0)
20            {
21                FStopWatch Stopwatch;
22                Stopwatch.Start();
23                ParallelFor(GroupedRaytraceInfos.Num(), [&] (const int32
Index)
24                {
25                    ExecuteRaytrace(GroupedRaytraceInfos[Index], false);
26                });
27            }
28        }
29    }
30 }
```

```
26         });
27         StopWatch.Stop();
28         Readings.Add(StopWatch.GetTimeInMilliseconds());
29         GroupedRaytraceInfos.Empty();
30         LastGroup = Info->GetRaytracingGroup();
31     }
32     GroupedRaytraceInfos.Add(Info);
33 }
34 }
35 return Readings;
36 }
```

Listing 11: *ExecuteRaytrace* Funktion in RTRaytracingSubsystem.cpp

```
1 double URTRaytracingSubsystem::ExecuteRaytrace(URaytraceInfo* RaytraceInfo,
2     bool bDrawDebug) const
3 {
4     FHitResult Result;
5     FCollisionObjectQueryParams CollisionObjectQueryParams;
6     CollisionObjectQueryParams.AddObjectTypeToQuery(ECC_WorldStatic);
7     FCollisionQueryParams CollisionQueryParams;
8     CollisionQueryParams.bTraceComplex = bUseComplexCollision;
9     ...
10    const bool bIsHit = GetWorld()->LineTraceSingleByObjectType(Result,
11        RaytraceInfo->GetOrigin(),
12        RaytraceInfo->GetOrigin() + RaytraceInfo->GetDirection()
13        * RaytraceInfo->GetMaxDistance(),
14        CollisionObjectQueryParams, CollisionQueryParams);
15    ...
16    return StopWatch.GetTimeInMilliseconds();
17 }
```

5 Evaluation

Um die Performance beider Ansätze zu evaluieren, sind 2 Testszenarien konzipiert. Beide Szenarien bestehen aus 1000 Instanzen des selben 3D-Modells, welches aus dem *Female Mannequin Character for Stylized Female Asset-Paket* für Unreal Engine 5 entnommen wurde[36]. Es bietet sich aufgrund seiner relativ hohen Polygonanzahl als Test-3D-Modell an. Aufgrund fehlender Hitboxen wurden diese händisch eingefügt. Abbildung 12 im Anhang zeigt das 3D-Modell mit Hitboxen.

In Testszenario Eins wird das 3D-Modell 1000-mal an verschiedenen vorgegebenen Positionen platziert. Jede Modellinstanz besitzt dieselbe Rotation sowie Skalierung und wird von einem Strahl mit vorgegebener Position und Richtung beschossen. Die Strahlen können treffen oder daneben gehen.

Testszenario Zwei gleicht dem ersten Szenario, jedoch wird anstelle eines einzigen Strahls pro 3D-Modell 24 Strahlen kegelförmig, wie eine Schrotflinte, in Richtung des 3D-Modells geschossen. Einzelne Strahlen können treffen oder daneben gehen.

Beide Szenarien werden 100-mal durchlaufen, und die Berechnungszeit jedes Strahls bzw. im Falle von Szenario Zwei jedes Schrotflinten-Schusses, wird gemessen. Nach Abschluss des Szenarios werden alle Zeitwerte in eine Datei exportiert.

Alle nachfolgenden Diagramme sind aus Gründen der Lesbarkeit ohne Ausreißer dargestellt. Zudem sind die Ausreißer mit hoher Wahrscheinlichkeit durch Lastspitzen von anderen Programmen bzw. Renderaufgaben der GPU entstanden. Es ist technisch nicht möglich diese Lastspitzen vollständig zu verhindern. Abbildungen 13-17 im Anhang stellen die Daten der Vollständigkeit halber mit Ausreißern dar.

Die Testszenarien wurden auf einem *AMD Ryzen 9 7900X* Prozessor¹⁶ und einer *NVIDIA RTX 3080* Grafikkarte¹⁷ ausgeführt.

Eine Auswertung der Messdaten beider Szenarien in den Konfigurationen *UE5 Complex* (volle 3D-Modelle), *UE5 Hitbox* (Hitboxen), *GPU Raytrace* und *GPU Total* in Abbildung 5 zeigt dass hardware-beschleunigtes GPU-Raytracing je nach Szenario durchschnittlich 48,5-63 mal langsamer ist als CPU Raytracing mit Hitboxen (*UE5 Hitbox*) oder vollen 3D-Modellen (*UE5 Complex*). Der unterste Balken *GPU Total* gibt die durchschnittliche Berechnungszeit inklusive aller zusätzlich notwendigen Operationen wie BLAS & TLAS Generierung oder Lesen der Output Buffer von der GPU an. Verglichen mit einfachen Hitboxen in UE5 ist GPU Raytracing insgesamt durchschnittlich 2600 mal langsamer. Trotz der 24 mal höheren Anzahl an Strahlen pro 3D-Modell in Szenario Zwei ist die durchschnittliche Berechnungszeit im GPU Raytracing annähernd identisch. Anders als GPU Raytracing schwankt das Ausführen der Treffererkennung auf der CPU in UE5 stark. Durchschnittliche Zeiten schwanken zwischen 9 bis 19 Mikrosekunden.

¹⁶<https://www.amd.com/de/products/processors/desktops/ryzen/7000-series/amd-ryzen-9-7900x.html>

¹⁷<https://www.nvidia.com/de-de/geforce/graphics-cards/30-series/rtx-3080-3080ti/>

5 Evaluation

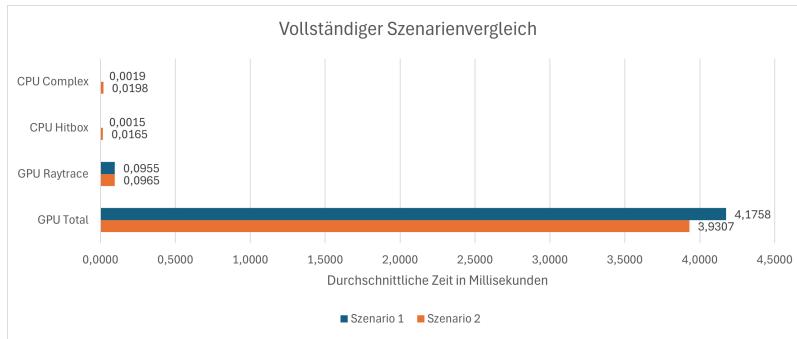


Abbildung 5: Die durchschnittliche Zeit aller Implementationen in beiden Szenarien

Die Zeit zum Erstellen der BLAS schwankt insgesamt zwischen beiden Szenarien stark. Wenngleich der Median beider wenig voneinander abweicht. Selbiges gilt für die TLAS. Extrema sind hier weniger stark voneinander abweichend. Siehe dazu Abbildung 6.

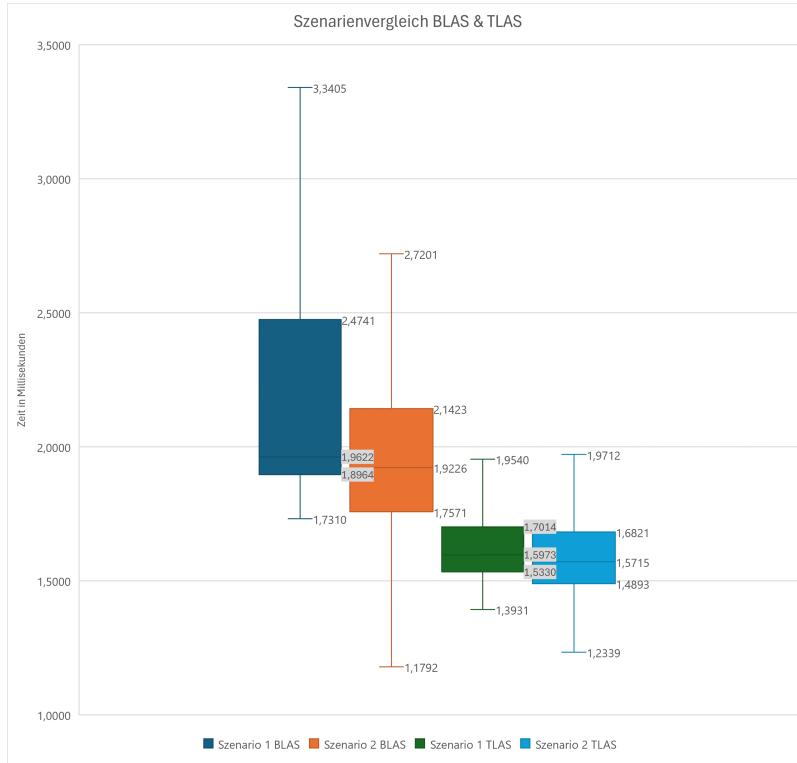


Abbildung 6: Boxplots der BLAS & TLAS Generierung beider Szenarien

5 Evaluation

Das Aufnehmen der Command Buffer in Abbildung 7 ist annähernd identisch mit minimal abweichenden Extrema und bewegt sich im Rahmen zwischen 1,9 bis 40 Mikrosekunden.

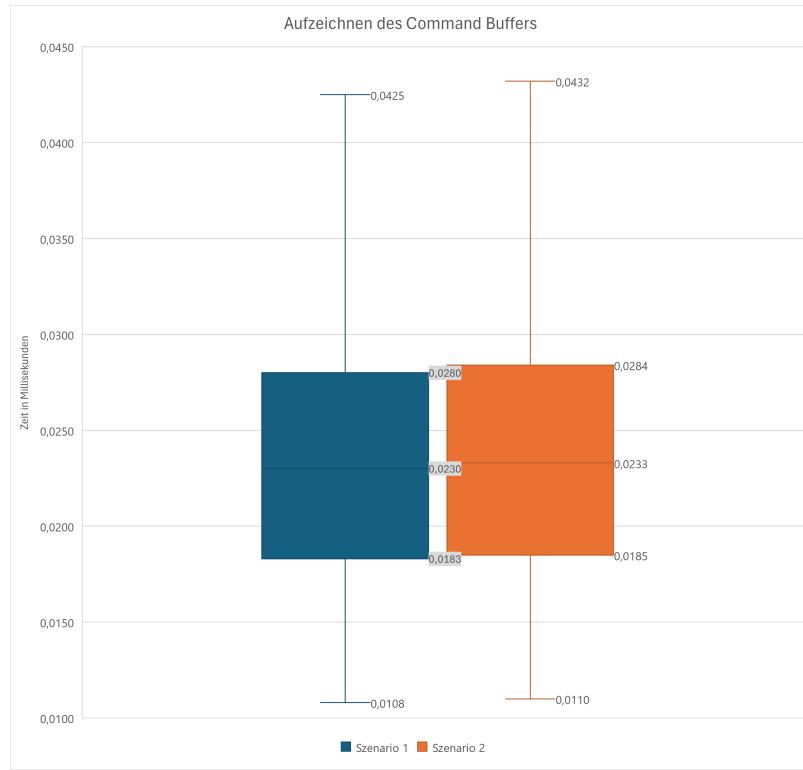


Abbildung 7: Boxplots des Aufzeichnen der Command Buffer beider Szenarien

5 Evaluation

Abbildung 8 zeigt eine 10-fache Steigerung der benötigten Zeit zum Transfer der Ergebnisse von der GPU zur CPU in Szenario Zwei gegenüber Szenario Eins.

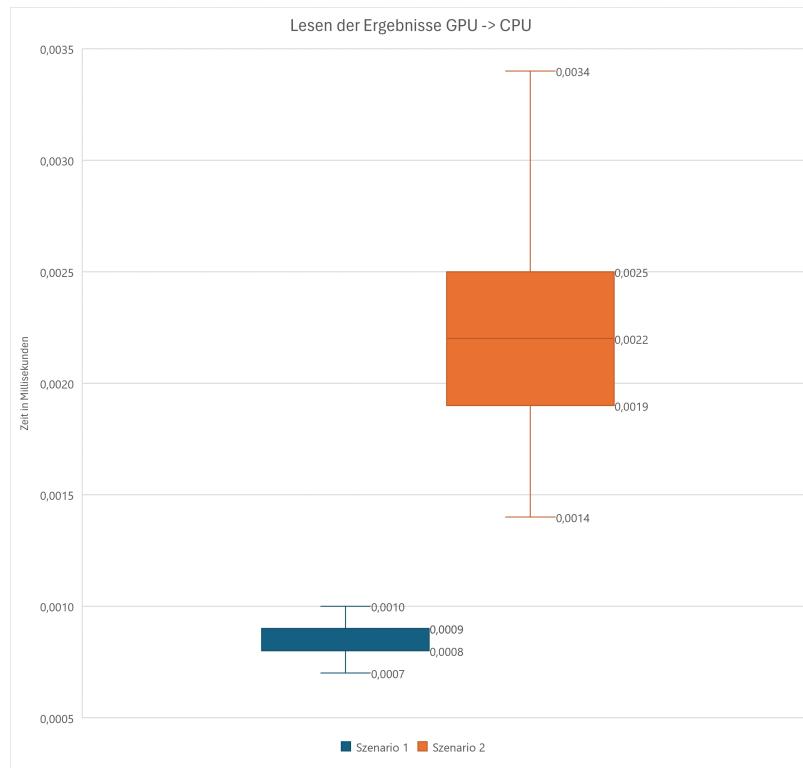


Abbildung 8: Boxplots der benötigten Zeit zum Transfer der Ergebnisse von GPU->CPU beider Szenarien

5 Evaluation

Die Aufräumzeit der BLAS & TLAS vom vorigen traceRays Durchlauf ist in beiden Szenarien annähernd gleich mit 0,2 Millisekunden, wie Abbildung 9 darstellt.

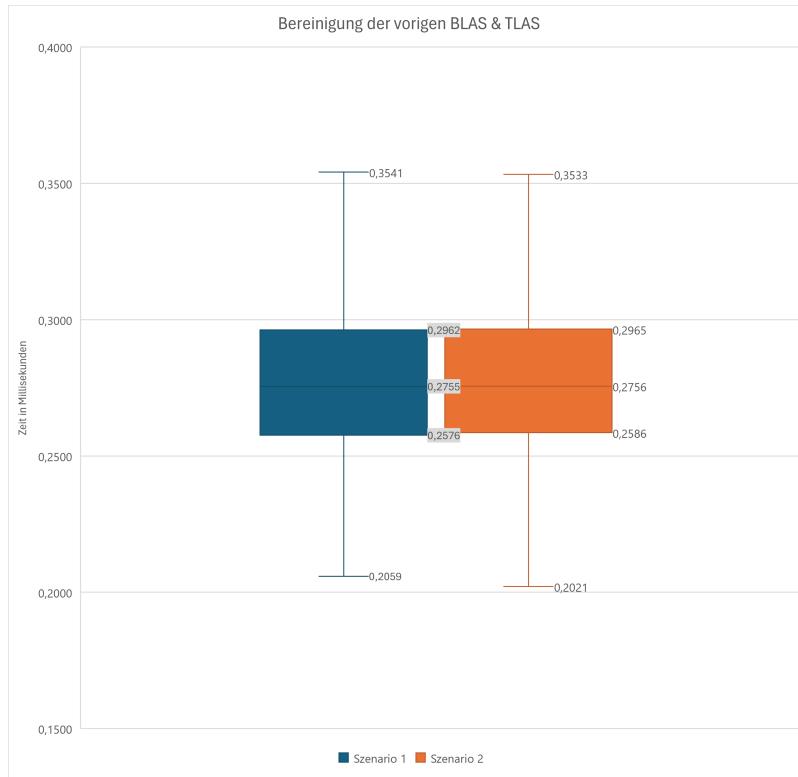


Abbildung 9: Boxplots der benötigten Zeit zum bereinigen der BLAS & TLAS des vorigen Durchlaufs

Eine Ausnahme von den zuvor erwähnten Ausreißern sind die im Anhang in Abbildung 13 zu sehenden Ausreißer auf der Nulllinie der Y-Achse keine Lastspitzen durch andere Programme, sondern ergeben sich aus der Tatsache, dass beim allerersten Programmdurchlauf keine vorigen BLAS & TLAS aufgeräumt werden müssen.

5 Evaluation

Abschließend schlüsselt Abbildung 10 die Boxplots der Szenarien in UE5 zusammen mit den *Hitbox* und *Complex* Durchläufen detailliert auf. Szenario Eins ist in beiden Varianten sowohl bei Median als auch Extrema in *einstelligen* Mikrosekundenbereich. Szenario Zwei hingegen zeigt eine größere Streuung. Dennoch bewegen sich beide Varianten im niedrigen *zweistelligen* Mikrosekundenbereich.

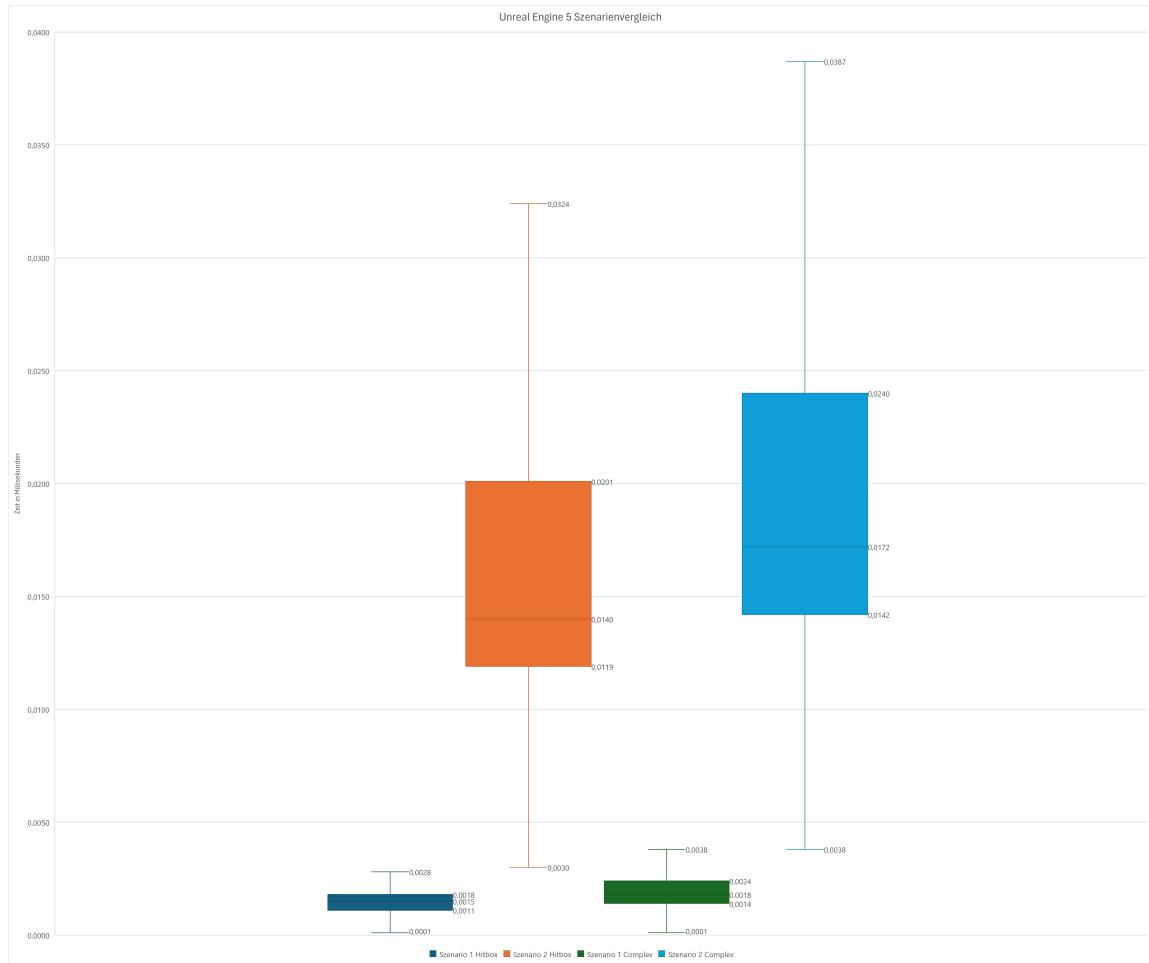


Abbildung 10: Boxplots der Szenarien und Varianten in UE5

6 Zusammenfassung und Ausblick

Wie die Evaluation eindeutig zeigt, ist das Hardware-basierte GPU-Raytracing mit Vulkan Ray Queries signifikant langsamer. Betrachtet man zudem die Zeit, welche benötigt wird um die BLAS & TLAS zu erzeugen, wird klar, dass diese Technik mehrere Millisekunden an Zeit kostet. In Videospielen ist dies zu lang. Die Ursachen können hierbei vielfältig sein. Einerseits könnte es an fehlenden oder inkorrekt gesetzten Vulkan-Flags liegen, andererseits könnte die PCI-E Schnittstelle nicht schnell genug sein. Auch überrascht es wie schnell die UE5 *Complex* Variante ist. Allerdings gilt es hierbei zu beachten dass lediglich ein einziges 3D-Modell getestet wurde. Bei dem Schrotflinten-Szenario wurde die Hochparallelisierung der Grafikkarte deutlich. Die reinen GPU-Raytracing Zeiten unterscheiden sich in beiden Szenarien minimal voneinander während die UE5 Testdaten eine Steigerung des Berechnungsaufwandes zeigen. In dieser Arbeit wurden nur statische 3D-Modelle betrachtet. Die Instanzen unterscheiden sich hierbei nur in Position, Rotation und Skalierung. In Videospielen kommen jedoch auch sehr viele animierte 3D-Modelle vor. Die Geometrie verformt sich. Hier könnte der GPU-Ansatz aufgrund der Parallelisierung einen Vorteil haben. Weiterführende Forschungen zu dieser Arbeit könnten die Nutzung von Machine Learning zur weiteren Performance-Steigerung der Treffererkennung analog dem existierenden DLSS Ray Reconstruction sowie die Integration von Machine Learning in existierende CPU-basierte Physik-Engines sein.

Abschließend lassen der aktuelle Forschungsstand sowie die technologische Weiterentwicklung der RT-Kerne und des Machine Learnings hoffen, dass Treffererkennung auf Grafikkarten die aktuell eingesetzten Hitboxen in näherer Zukunft ersetzt.

7 Literaturverzeichnis

Gedruckte Quellen

- [2] Andrew S. Glassner. *An Introduction to Ray Tracing*. 1.3. Morgan Kaufmann, Apr. 2019, S. 353. ISBN: 9780122861604. URL: <https://www.realtimerendering.com/raytracing/An-Introduction-to-Ray-Tracing-The-Morgan-Kaufmann-Series-in-Computer-Graphics-.pdf> (besucht am 25.08.2024).
- [3] David S. Immel, Michael F. Cohen und Donald P. Greenberg. „A radiosity method for non-diffuse environments“. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, S. 133–142. ISBN: 0897911962. DOI: 10.1145/15922.15901. URL: <https://doi.org/10.1145/15922.15901> (besucht am 25.08.2024).
- [4] 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: Association for Computing Machinery, 1986, S. 143–150. ISBN: 0897911962. DOI: 10.1145/15922.15902. URL: <https://doi.org/10.1145/15922.15902>.
- [5] NVIDIA Corporation. *NVIDIA TURING GPU ARCHITECTURE*. Sep. 2018. URL: <https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf> (besucht am 27.05.2024).
- [7] Tomas Möller und Ben Trumbore. „Fast, Minimum Storage Ray-Triangle Intersection“. In: *Journal of Graphics Tools* 2.1 (1997), S. 21–28. DOI: 10.1080/10867651.1997.10487468. eprint: <https://doi.org/10.1080/10867651.1997.10487468>. URL: <https://doi.org/10.1080/10867651.1997.10487468> (besucht am 25.08.2024).
- [8] Christer Ericson. *Real-Time Collision Detection*. Taylor & Francis Group, 2004, S. 632. ISBN: 9781000750553.
- [9] Donald Meagher. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. Techn. Ber. Okt. 1980.
- [10] Philippe Weier u. a. „N-BVH: Neural ray queries with bounding volume hierarchies“. In: *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers '24*. SIGGRAPH '24. ACM, Juli 2024. DOI: 10.1145/3641519.3657464. URL: <http://dx.doi.org/10.1145/3641519.3657464>.
- [11] Durga Keerthi Mandarapu, Nicholas James und Milind Kulkarni. *Mochi: Fast & Exact Collision Detection*. 2024. arXiv: 2402.14801 [cs.GR]. URL: <https://arxiv.org/abs/2402.14801>.
- [12] John Burgess. „RTX on—The NVIDIA Turing GPU“. In: *IEEE Micro* 40.2 (2020), S. 36–44. DOI: 10.1109/MM.2020.2971677.

Online-Quellen

- [13] Ingo Wald u. a. „RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location“. In: *High-Performance Graphics - Short Papers*. Hrsg. von Markus Steinberger und Tim Foley. The Eurographics Association, 2019. ISBN: 978-3-03868-092-5. DOI: [10.2312/hpg.20191189](https://doi.org/10.2312/hpg.20191189).
- [14] Yuhao Zhu. *RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing*. 2022. arXiv: 2201.01366 [cs.DC].
- [15] Kessenich John Graham Sellers. *Vulkan™ Programming Guide: The Official Guide to Learning Vulkan*. Addison-Wesley, 2017. ISBN: 978-0-13-446454-1.
- [24] R.J. Rost, J.M. Kessenich und B. Lichtenbelt. *OpenGL Shading Language*. Graphics programming. Addison-Wesley, 2004. ISBN: 9780321197894. URL: https://books.google.de/books?id=kDXOXv_GeswC (besucht am 25.08.2024).
- [30] Jose-Emilio Muñoz-Lopez. *Ray Tracing: delivering immersivegaming experiences on mobile*. 2023. URL: https://vulkan.org/user/pages/09.events/vulkanised-2023/vulkanised_2023_ray_tracing_delivering_immersive_gaming_experiences_on_mobile.pdf (besucht am 25.08.2024).

Online-Quellen

- [1] Daniel Koch u. a. *Ray Tracing In Vulkan*. Dez. 2020. URL: https://www.khronos.org/blog/ray-tracing-in-vulkan/%5C#blog%5C_Ray%5C_Queries (besucht am 25.08.2024).
- [6] NVIDIA Corporation. *NVIDIA DLSS 3*. URL: <https://www.nvidia.com/de-de/geforce/technologies/dlss/> (besucht am 26.08.2024).
- [16] The Khronos Group. *VkPhysicalDevice(3) Manual Page*. 2024. URL: <https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkPhysicalDevice.html> (besucht am 24.08.2024).
- [17] The Khronos Group. *VkDevice(3) Manual Page*. 2024. URL: <https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkDevice.html> (besucht am 24.08.2024).
- [18] The Khronos Group. *Extending Vulkan*. 2024. URL: <https://docs.vulkan.org/spec/latest/chapters/extensions.html> (besucht am 24.08.2024).
- [19] Sean Harmer. *Projection Matrices with Vulkan – Part 1*. Nov. 2023. URL: <https://www.kdab.com/projection-matrices-with-vulkan-part-1/> (besucht am 17.08.2024).
- [20] The Khronos Group. *Queues*. 2024. URL: <https://docs.vulkan.org/guide/latest/queues.html> (besucht am 24.08.2024).
- [21] The Khronos Group. *Memory Allocation*. 2024. URL: <https://docs.vulkan.org/spec/latest/chapters/memory.html> (besucht am 24.08.2024).
- [22] The Khronos Group. *Synchronization and Cache Control*. 2024. URL: <https://registry.khronos.org/vulkan/specs/1.3/html/vkspec.html#synchronization> (besucht am 24.08.2024).

- [23] Raphael Mun u. a. *Understanding Vulkan Synchronization*. 2024. URL: <https://www.khronos.org/blog/understanding-vulkan-synchronization> (besucht am 24.08.2024).
- [25] The Khronos Group. *The Industry Open Standard Intermediate Language for Parallel Compute and Graphics*. Juli 2024. URL: <https://www.khronos.org/spir/> (besucht am 17.08.2024).
- [26] The Khronos Group. *Allocation of Descriptor Sets*. 2024. URL: <https://docs.vulkan.org/spec/latest/chapters/descriptorsets.html#descriptorsets-allocation> (besucht am 24.08.2024).
- [27] The Khronos Group. *Pipelines*. 2024. URL: <https://docs.vulkan.org/spec/latest/chapters/pipelines.html> (besucht am 17.08.2024).
- [28] SookieSpy. *Audiovisual explanation of hitbox issues in Apex Legends*. Feb. 2019. URL: <https://youtu.be/1EuZEmoRU4A?t=68> (besucht am 17.08.2024).
- [29] Jay Frechette. *Patchnotes of Season 1 of Apex Legends*. März 2019. URL: <https://www.ea.com/games/apex-legends/news/preseason-update-march-wild-frontier-update> (besucht am 17.08.2024).
- [31] The Khronos Group. *Ray-tracing: Extended features and dynamic objects*. 2024. URL: https://docs.vulkan.org/samples/latest/samples/extensions/ray_tracing_extended/README.html (besucht am 25.08.2024).
- [32] Nia Bickford. *vk mini path tracer*. URL: https://nvpro-samples.github.io/vk_mini_path_tracer/ (besucht am 27.05.2024).
- [33] The Khronos Group. *Vulkan-Hpp: C++ Bindings for Vulkan*. 2024. URL: <https://github.com/KhronosGroup/Vulkan-Hpp> (besucht am 25.08.2024).
- [34] The Khronos Group. *VkMemoryPropertyFlagBits(3) Manual Page*. 2024. URL: <https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkMemoryPropertyFlagBits> (besucht am 25.08.2024).
- [35] Isara Tech. *UE4 – Improving speed with ParallelFor*. Dez. 2019. URL: <https://isaratech.com/ue4-improving-speed-with-parallelfor/> (besucht am 25.08.2024).
- [36] PO-Art. *Female Mannequin Character for Stylized Female*. Feb. 2017. URL: <https://www.unrealengine.com/marketplace/en-US/product/female-mannequin> (besucht am 25.08.2024).

8 Listingverzeichnis

1	<i>traceRay</i> Funktion in Pseudocode	12
2	Definition der Raytracing Extensions in VulkanRenderer.hpp	16
3	Anfragen der Raytracing Extensions in VulkanRenderer.cpp	16
4	Erzeugen des <i>Descriptor Set Layouts</i> in VulkanRenderer.cpp	17
5	Erstellen der Shader Storage Buffer Objects in VulkanRenderer.cpp	18
6	Sammeln der 3D-Modellgeometrie in VulkanRenderer.cpp	19
7	Erzeugung der Bottom-Level Acceleration Structures in VulkanRenderer.cpp	20
8	Erzeugung der Top-Level Acceleration Structure in VulkanRenderer.cpp	23
9	<i>traceRay</i> Funktion in VulkanRenderer.cpp	27
10	<i>ExecuteRaytraces</i> Funktion in RTRaytracingSubsystem.cpp	29
11	<i>ExecuteRaytrace</i> Funktion in RTRaytracingSubsystem.cpp	30
12	Erstellen der Compute-Pipeline in VulkanRenderer.cpp	47
13	<i>ComputeInputBufferObject</i> -Klasse in ComputeInputBufferObject.hpp	48
14	<i>ComputeOutputBufferObject</i> -Klasse in ComputeOutputBufferObject.hpp	48
15	Erstellen des Descriptor Pools in VulkanRenderer.cpp	48
16	Erstellen des Descriptor Sets in VulkanRenderer.cpp	49
17	Erstellen des Compute Command Pools in VulkanRenderer.cpp	50

9 Abbildungsverzeichnis

1	Funktionsweise der RT-Kerne	8
2	Fluss-ähnliches Diagramm einer Raytracing-Pipeline	13
3	Fluss-ähnliches Diagramm einer Ray Query	13
4	Acceleration Structure	14
5	Vollständiger Szenarienvergleich Durchschnittszeit	32
6	Szenarienvergleich BLAS & TLAS Erstellung	32
7	Szenarienvergleich Command Buffer Aufzeichnung	33
8	Szenarienvergleich Transfer der Ergebnisse GPU->CPU	34
9	Szenarienvergleich Bereinigung voriger BLAS & TLAS	35
10	Detaillierter Szenarienvergleich in UE5	36
11	Darstellung des Bounding Volume Hierarchy-Algorithmus auf RT-Kernen	47
12	UE5 3D-Modell mit Hitboxen	51
13	Szenarienvergleich BLAS & TLAS Erstellung mit Ausreißern	52
14	Szenarienvergleich Command Buffer Aufzeichnung mit Ausreißern	53
15	Szenarienvergleich Transfer der Ergebnisse GPU->CPU mit Ausreißern	54
16	Szenarienvergleich Bereinigung voriger BLAS & TLAS mit Ausreißern	55
17	Detaillierter Szenarienvergleich in UE5 mit Ausreißern	56

10 Glossar

Acceleration Structures Spezielle Datenstrukturen, die zur Optimierung des Raytracings verwendet werden.

Bounding Volume Hierarchy (BVH) Eine Baumstruktur von Bounding Volumes zur schnellen Kollisionserkennung.

Command Buffer Speicher für eine Abfolge von Befehlen, die an die GPU gesendet werden.

Deep Learning Super Sampling (DLSS) Ein Machine-Learning-Algorithmus von NVIDIA zur Bildskalierung und -verbesserung.

Descriptor Pools Speicherpools zur Optimierung der Erstellung von Descriptor Sets in Vulkan.

Descriptor Sets Eine Sammlung von Ressourcenbindepunkten in Vulkan, die in Shadern verwendet werden.

Hitboxen Vereinfachte geometrische Formen zur Kollisionserkennung in Videospielen.

Logical Device Eine Abstraktion in Vulkan, die Ressourcen allokiert und einen Zustand besitzt.

Memory Barriers Synchronisationsmechanismen in Vulkan, die sicherstellen, dass Speicheroperationen abgeschlossen sind.

Physical Device Repräsentiert eine oder mehrere Hardware-Komponenten in Vulkan.

Physik-Engine Software zur Simulation physikalischer Effekte in Echtzeit-Anwendungen.

Pipeline Eine Kombination aus Shadern in Vulkan, die vorab kompiliert wird, um die Performance zu erhöhen.

Pipelines Grafische oder Berechnungspipelines, die in Vulkan verwendet werden, um Shader und andere Grafikanweisungen zu verwalten.

Queue Eine Warteschlange in Vulkan, die zur Verwaltung von Befehlen dient.

Ray Queries Eine vereinfachte Form von Raytracing-Pipelines, die innerhalb von Shadern ausgeführt werden.

Raytracing Eine Technik zur Bildsynthese, bei der Lichtstrahlen in einer 3D-Szene simuliert werden.

Shader Programme, die auf Grafikhardware ausgeführt werden, um verschiedene Berechnungen durchzuführen.

Shader Storage Buffer Object (SSBO) Ein Speicherobjekt, das zum Datentransfer zwischen CPU und Shader verwendet wird.

Vulkan Eine hardwarenahe Programmierschnittstelle für Grafikkarten, die Entwicklern präzise Kontrolle über die Hardware ermöglicht.

11 Akronyme

AABB Axis-aligned Bounding Box.

API Application Programmable Interface.

BLAS Bottom-Level Acceleration Structure.

BV Bounding Volume.

BVH Bounding Volume Hierarchy.

Compute Shader Shader für allgemeine Berechnungen auf der GPU.

CPU Central Processing Unit.

DLSS Deep Learning Super Sampling.

FPS Frames per Second.

GLSL OpenGL Shading Language.

GPU Graphics Processing Unit.

L2 Cache Second-Level Cache der CPU.

ML Machine Learning.

PCI-E Peripheral Component Interconnect Express.

Queue Family Gruppe von Queues in Vulkan, die ähnliche Aufgaben ausführen.

RT-Kerne Raytracing-Kerne.

SPIR-V Standard Portable Intermediate Representation-V.

SSBO Shader Storage Buffer Object.

TLAS Top-Level Acceleration Structure.

UE5 Unreal Engine 5.

VRAM Video Random Access Memory.

12 Anhangsverzeichnis

1	BVH-Algorithmus Visualisierung	47
2	Compute-Pipeline Initialisierung	47
3	ComputeInputBufferObject-Klasse	48
4	ComputeOutputBufferObject-Klasse	48
5	Descriptor Pool Erzeugung	48
6	Descriptor Set Erzeugung	49
7	Compute Command Pool Erstellung	50
8	UE5 3D-Modell mit Hitboxen	51
9	Szenarienvergleich BLAS & TLAS Erstellung mit Ausreißern	52
10	Szenarienvergleich Command Buffer Aufzeichnung mit Ausreißern	53
11	Szenarienvergleich Transfer der Ergebnisse GPU->CPU mit Ausreißern	54
12	Szenarienvergleich Bereinigung voriger BLAS & TLAS mit Ausreißern	55
13	Detaillierter Szenarienvergleich in UE5 mit Ausreißern	56
14	Erklärung über das eigenständige Erstellen der Arbeit	57

BVH-Algorithmus Visualisierung

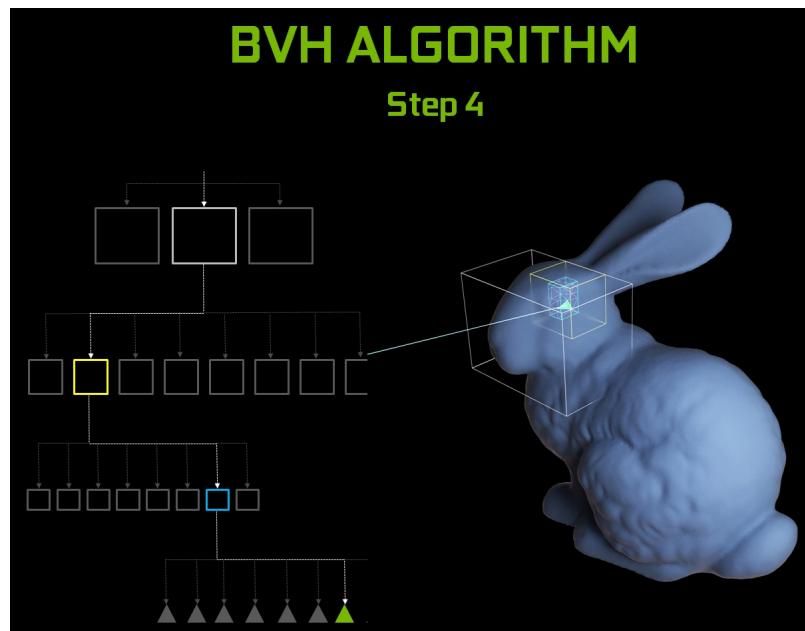


Abbildung 11: Darstellung des Bounding Volume Hierarchy-Algorithmus auf RT-Kernen[5]

Compute-Pipeline Initialisierung

Listing 12: Erstellen der Compute-Pipeline in VulkanRenderer.cpp

```
1 void VulkanRenderer::createComputePipeline()
2 {
3     ...
4     const auto computeShaderAsset = assetManager->getAsset<Assets::
5         ShaderAsset>("Assets/Shaders/compute.comp.spv");
6     ...
7     auto computeShaderModule = createShaderModule(computeShaderAsset->
8         getShader());
9     vk::PipelineShaderStageCreateInfo computeShaderStageInfo = {};
10    computeShaderStageInfo.stage = vk::ShaderStageFlagBits::eCompute;
11    computeShaderStageInfo.module = *computeShaderModule;
12    computeShaderStageInfo.pName = "main";
13
14    vk::PipelineLayoutCreateInfo pipelineLayoutInfo = {};
15    pipelineLayoutInfo.setLayoutCount = 1;
16    pipelineLayoutInfo.pSetLayouts = &computeDescriptorsetLayout;
17    ...
18    computePipelineLayout = logicalDevice->createPipelineLayout(
19        pipelineLayoutInfo);
20    ...
21    vk::ComputePipelineCreateInfo computePipelineCreateInfo = {};
```

```

19     computePipelineCreateInfo.stage = computeShaderStageInfo;
20     computePipelineCreateInfo.layout = computePipelineLayout;
21
22     const auto computePipelineResult = logicalDevice->
23     createComputePipeline(nullptr, computePipelineCreateInfo);
24     ...
25     computePipeline = computePipelineResult.value;
26     ...
}

```

ComputeInputBufferObject-Klasse

Listing 13: *ComputeInputBufferObject*-Klasse in ComputeInputBufferObject.hpp

```

1 struct ComputeInputBufferObject
2 {
3     alignas(16) glm::vec3 origin;
4     alignas(16) glm::vec3 direction;
5     alignas(4) float tMin;
6     alignas(4) float tMax;
7     alignas(4) uint32_t rayId;
8 };

```

ComputeOutputBufferObject-Klasse

Listing 14: *ComputeOutputBufferObject*-Klasse in ComputeOutputBufferObject.hpp

```

1 struct ComputeOutputBufferObject
2 {
3     alignas(8) uint32_t isHit;
4     alignas(8) uint32_t rayId;
5 };

```

Descriptor Pool Erzeugung

Listing 15: Erstellen des Descriptor Pools in VulkanRenderer.cpp

```

1 void VulkanRenderer::createComputeDescriptorPool()
2 {
3     constexpr vk::DescriptorPoolSize storageBufferpoolSize{vk::
4         DescriptorType::eStorageBuffer, 1};
5     constexpr vk::DescriptorPoolSize accelerationStructurePoolSize{
6         vk::DescriptorType::eAccelerationStructureKHR, 1
7     };
8     constexpr std::array poolSizes = {accelerationStructurePoolSize,
9         storageBufferpoolSize};
10    const vk::DescriptorPoolCreateInfo poolInfo{

```

```

9         {}, 1, static_cast<uint32_t>(poolSizes.size()), poolSizes.data()
10    };
11    ...
12    computeDescriptorPool = logicalDevice->createDescriptorPool(poolInfo);
13    ...
14 }

```

Descriptor Set Erzeugung

Listing 16: Erstellen des Descriptor Sets in VulkanRenderer.cpp

```

1 void VulkanRenderer::createComputeDescriptorSet()
2 {
3     vk::DescriptorSetAllocateInfo allocInfo = {};
4     allocInfo.descriptorPool = computeDescriptorPool;
5     allocInfo.descriptorSetCount = 1;
6     allocInfo.pSetLayouts = &computeDescriptorsetLayout;
7     ...
8     computeDescriptorSet = logicalDevice->allocateDescriptorSets(allocInfo)
9     [0];
10    ...
11    const vk::WriteDescriptorSetAccelerationStructureKHR
12        tlasDescriptorWrite{
13        1, &topLevelAccelerationStructure.accelerationStructureKHR
14    };
15    vk::WriteDescriptorSet accelerationStructureDescriptorWrite{};
16    accelerationStructureDescriptorWrite.dstSet = computeDescriptorSet;
17    accelerationStructureDescriptorWrite.dstBinding = 0;
18    accelerationStructureDescriptorWrite.dstArrayElement = 0;
19    accelerationStructureDescriptorWrite.descriptorType = vk::
20        DescriptorType::eAccelerationStructureKHR;
21    accelerationStructureDescriptorWrite.descriptorCount = 1;
22    accelerationStructureDescriptorWrite.pNext = &tlasDescriptorWrite;
23    const vk::DescriptorBufferInfo inputBufferInfo{inputComputeBuffer, 0,
24        sizeof(ComputeInputBufferObject)};
25    vk::WriteDescriptorSet inputDescriptorWrite = {};
26    inputDescriptorWrite.dstSet = computeDescriptorSet;
27    inputDescriptorWrite.dstBinding = 1;
28    inputDescriptorWrite.dstArrayElement = 0;
29    inputDescriptorWrite.descriptorType = vk::DescriptorType::
30        eStorageBuffer;
31    inputDescriptorWrite.descriptorCount = 1;
32    inputDescriptorWrite.pBufferInfo = &inputBufferInfo;
33    const vk::DescriptorBufferInfo outputBufferInfo{outputComputeBuffer, 0,
34        sizeof(ComputeOutputBufferObject)};
35    vk::WriteDescriptorSet outputDescriptorWrite = {};
36    outputDescriptorWrite.dstSet = computeDescriptorSet;
37    outputDescriptorWrite.dstBinding = 2;
38    outputDescriptorWrite.dstArrayElement = 0;
39    outputDescriptorWrite.descriptorType = vk::DescriptorType::
40        eStorageBuffer;
41    outputDescriptorWrite.descriptorCount = 1;
42    outputDescriptorWrite.pBufferInfo = &outputBufferInfo;

```

```
36     const std::array descriptorWrites = {
37         accelerationStructureDescriptorWrite, inputDescriptorWrite,
38         outputDescriptorWrite
39     };
40     logicalDevice->updateDescriptorSets(descriptorWrites, nullptr);
41 }
```

Compute Command Pool Erstellung

Listing 17: Erstellen des Compute Command Pools in VulkanRenderer.cpp

```
1 void VulkanRenderer::createComputeCommandPool()
2 {
3     const QueueFamilyIndices queueFamilyIndices = findQueueFamilies(
4         physicalDevice);
5     vk::CommandPoolCreateInfo poolInfo = {};
6     poolInfo.flags = vk::CommandPoolCreateFlagBits::eTransient;
7     poolInfo.queueFamilyIndex = queueFamilyIndices.computeFamily.value();
8     ...
9     computeCommandPool = logicalDevice->createCommandPool(poolInfo);
10 }
```

UE5 3D-Modell mit Hitboxen



Abbildung 12: Darstellung des verwendeten 3D-Modells mit Hitboxen

Szenarienvergleich BLAS & TLAS Erstellung mit Ausreißern

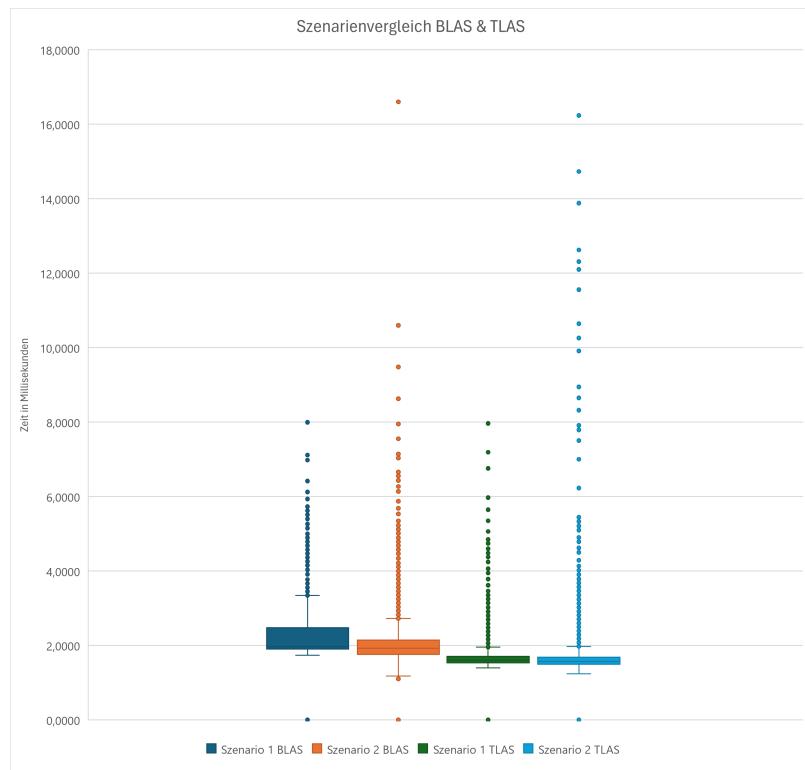


Abbildung 13: Boxplots der BLAS & TLAS Generierung beider Szenarien mit Ausreißern

Szenarienvergleich Command Buffer Aufzeichnung mit Ausreißern

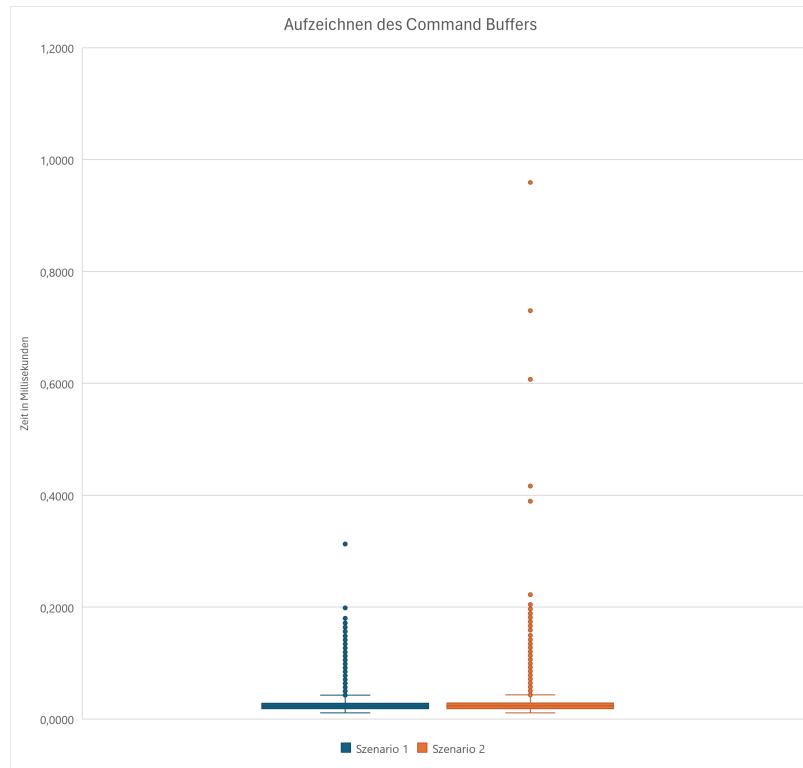


Abbildung 14: Boxplots des Aufzeichnen der Command Buffer beider Szenarien mit Ausreißern

Szenarienvergleich Transfer der Ergebnisse GPU->CPU mit Ausreißern

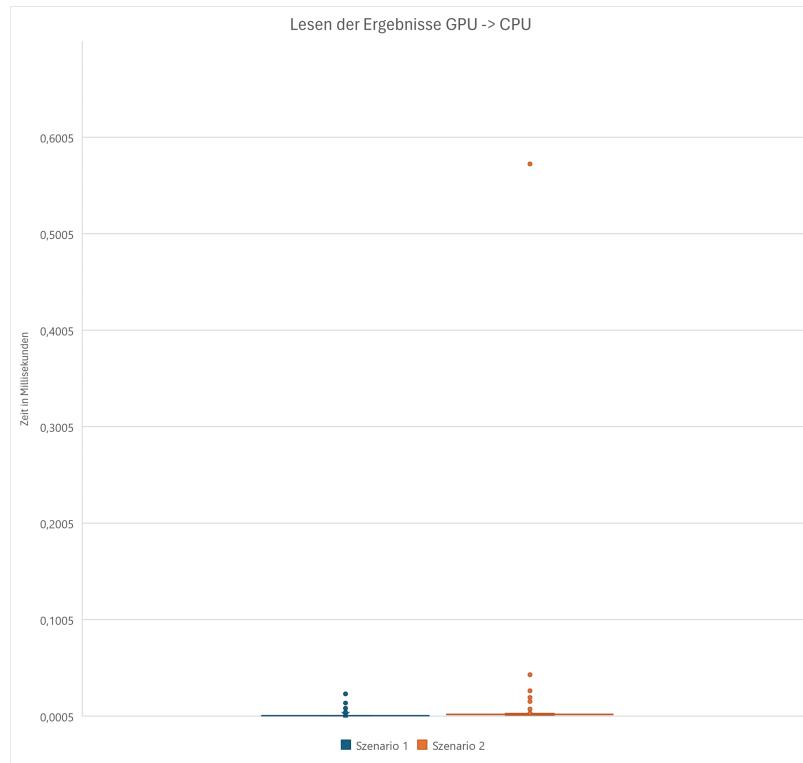


Abbildung 15: Boxplots der benötigten Zeit zum Transfer der Ergebnisse von GPU->CPU beider Szenarien mit Ausreißern

Szenarienvergleich Bereinigung voriger BLAS & TLAS mit Ausreißern

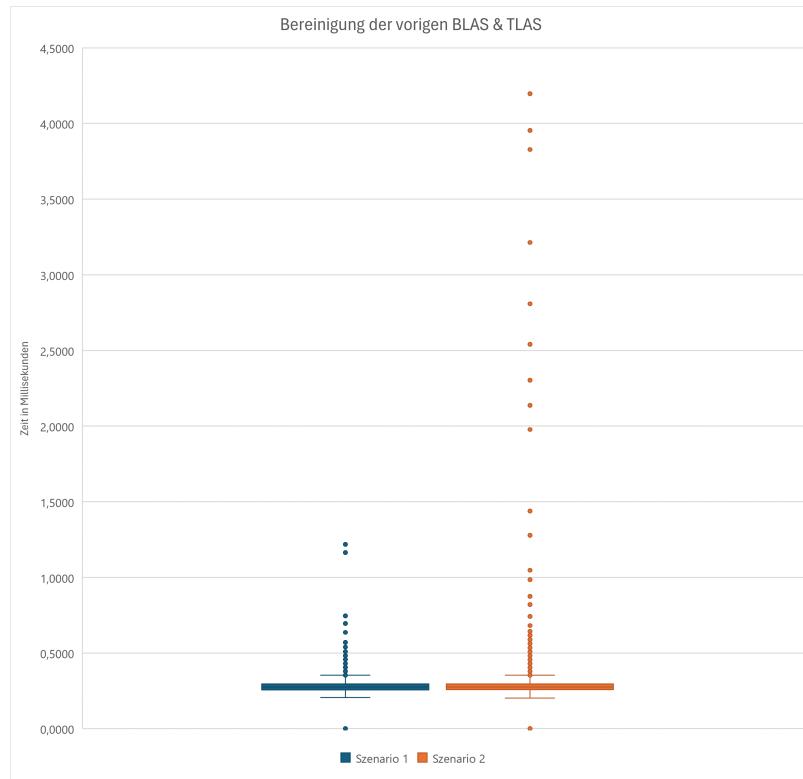


Abbildung 16: Boxplots der benötigten Zeit zum bereinigen der BLAS & TLAS des vorigen Durchlaufs mit Ausreißern

Detaillierter Szenarienvergleich in UE5 mit Ausreißern

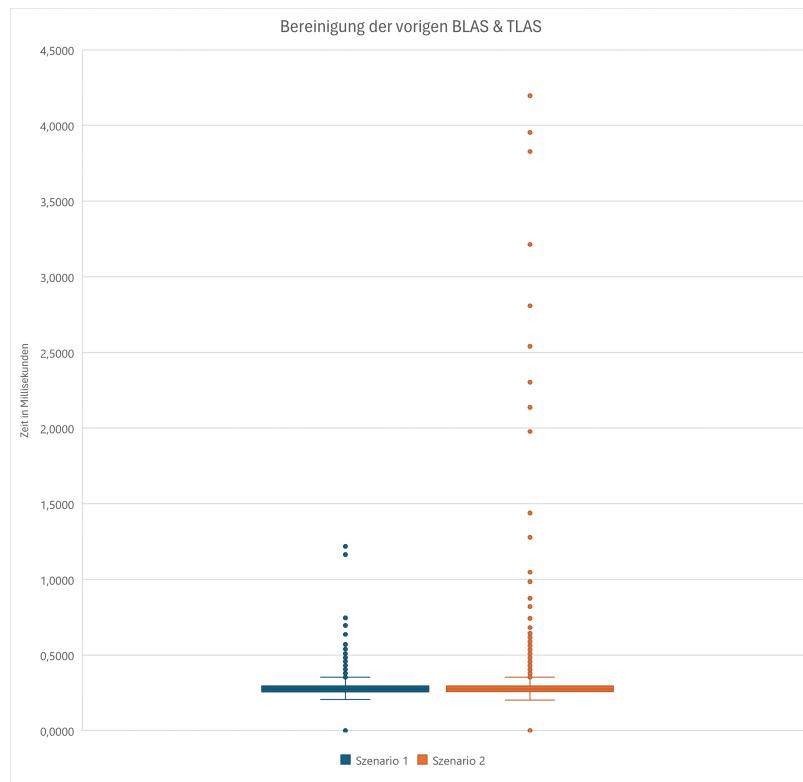


Abbildung 17: Boxplots der Szenarien und Varianten in UE5 mit Ausreißern

Erklärung über das eigenständige Erstellen der Arbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf in der Arbeit enthaltene Grafiken, Skizzen, bildliche Darstellungen sowie auf Quellen aus dem Internet. Die Arbeit habe ich in gleicher oder ähnlicher Form auch auszugsweise noch nicht als Bestandteil einer Prüfungs- oder Studienleistung vorgelegt.

Ich versichere, dass die eingereichte elektronische Version der Arbeit vollständig mit der Druckversion übereinstimmt.

Name: Kevin Niclas Kügler

Matrikelnummer: 399027

Bremen, den

.....
(Unterschrift)