Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Gerhard Wagner

Potenzialanalyse der GPU-Beschleunigung beim Raytracing

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Gerhard Wagner

Potenzialanalyse der GPU-Beschleunigung beim Raytracing

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 10. April 2018

Gerhard Wagner

Thema der Arbeit
Potenzialanalyse der GPU-Beschleunigung beim Raytracing

Stichworte
Raytracing, CUDA, GPGPU

Kurzzusammenfassung
Dieses Dokument befasst sich mit der Performanceanalyse eines Raytracers implementiert in CUDA. Zunächst zeigt dieses Dokument die Grundlagen des Raytracing Algorithmus. Da der Algorithmus Performance-Optimierungen benötigt um in Echtzeit-Anwendungen verwendung zu finden, werden Verbesserungen vorgestellt. Die Beschreibung des implementierten Raytracers zeigt, wie die Schnittpunktberechnungen implementiert wurden und wie die Bounding Volume Hierarchie erstellt wird. Im letzten Kapitel des Dokuments werden die Ergebnisse einiger Experimente mit dem Raytracer vorgestellt. Es werden auch Analysen des Nvidia Nsight Profilers gezeigt und erläutert.
Gerhard Wagner

Title of the paper
Performanceanalysis for GPU-based Raytracing

Keywords
Raytracing, CUDA, GPGPU

Abstract
This document is about performance analysis of an raytracer implemented in CUDA. Before explaining the implemented raytracer the document talks about the basics one needs to understand the raytracing algorithm. Since the algorithm needs some improvements to work in real-time, the document explains some improvements used for the raytracer. The explanation of the implemented raytracer shows how the algorithm is doing intersection tests and how the bounding volume hierarchy is created. To the end the document shows the results of some experiments made with the raytracer. In the last part of the document there are some analyses made with the Nvidia Nsight profiler.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Raytracing ist ein Verfahren zur Berechnung von Bildern aus einer virtuellen Szene. Der Algorithmus orientiert sich dabei am physikalischen Verhalten von Licht. Das Verfahren erfordert eine große Zahl an Berechnungen, wie der Schnittberechnung zwischen einer Fläche und einem Strahl. Die Anzahl der Berechnungen wirkt sich stark auf die Laufzeit des Algorithmus aus.

Ein großes Anwendungsgebiet des Raytracings ist die Computergrafik. In Animationsfilmen wird das Verfahren angewandt um realistische Beleuchtung zu erzielen. Für Simulationen wie z. B. Partikelsimulation einer Explosion kann Raytracing ebenfalls verwendet werden. Bei Kollisionberechnungen wird oft auch eine Form des Raytracings angewandt. Die hohe Laufzeit des Raytracings hat das Verfahren in Computerspielen und AR-/VR-Anwendungen nur schwer verwendbar gemacht. Erst mit der Möglichkeit Grafikkarten zum Raytracing zu benutzen, ist das Verfahren verwendbar in Echtzeit-Anwendungen.

Eine Möglichkeit der Beschleunigung des Raytracings ist das Verwenden von speziell zu diesem Zweck entwickelte Hardware. Diese Hardware ermöglicht es Raytracing auch in Echtzeit-Anwendungen zu verwenden. Eine weitere Möglichkeit der Perfomanceverbesserung ist das Verwenden eines General Purpose GPU (GPGPU) Toolkits. Für das Entwickeln von GPU Programmen gibt es die SDKs CUDA® und OpenCL™. CUDA wird von der NVIDIA® Corporation entwickelt und kann nur mit NVIDIA GPUs verwendet werden. OpenCL wird bereitgestellt von der Khronos Group und ist nicht an Hardware gebunden. Mit OpenCL kann sowohl eine GPU als auch eine CPU verwendet werden.

Ziel dieser Arbeit ist es, einen GPU-basierten Raytracer zu analysieren, der mit einem CUDA Programm läuft. Dieser Raytracer wurde im Verlauf dieser Arbeit implementiert.

Dazu werden zunächst die Grundlagen des Raytracings erläutert. Anschließend werden Konzepte zur Verbesserung der Performance vorgestellt. Danach wird der implementierte Raytracer beschrieben. Schließlich wird die Performanceanalyse ausgewertet.

2. Traditionelles Raytracing

Bereits im Jahre 1525 wurde das Konzept des Raytracings verwendet wie Dürer in einem Holzschnitt zeigt. In der Abbildung 2.1 zeigt, wie eine Laute perspektivisch abgebildet wird. Dafür wird eine Schnur durch den Bilderrahmen an die Konturen der Laute geführt und mit einer Nadel ein Loch an der entsprechenden Stelle in die Leinwand gebohrt. Die Leinwand ist mit einem Scharnier am Bilderrahmen befestigt. Der Raytracing Algorithmus wurde von Appel et al 1968 veröffentlicht [2]. Whitted erweiterte 1980 den Raytracing Algorithmus um Reflexion und Brechung [15].

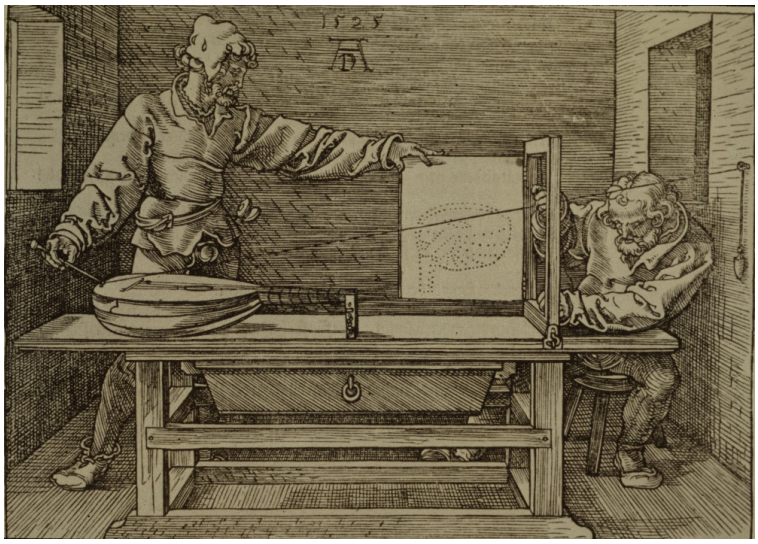

Abbildung 2.1.: Mann zeichnet Laute, Dürer 1525 https://commons.wikimedia.org/wiki/File:D%C3%BCrer_-_Man_Drawing_a_Lute.jpg

2.1. Grundlagen

Für die Berechnung eines einzelnen Bildes startet der Raytracer mit einem Strahl pro Pixel. Jeder Strahl wird bei einer direkten Implementierung mit jedem Objekt der Szene auf einen Schnittpunkt getestet. Für jeden Schnittpunkt wird ein Schattenstrahl je Lichtquelle erstellt und

diese werden wieder mit allen Objekten getestet. Bei Reflexion werden pro Schnittpunkt, an dem reflektiert wird, neue Strahlen erstellt, die wiederum mit allen Objekten getestet werden und Schattenstrahlen erzeugen können. Ein Beispiel soll zeigen wie viele Berechnungen bereits eine einfache Szene benötigt. Bei einer Bildauflösung von 1080p muss ein Raytracer $1920 \cdot 1080 = 2.073.600$ Strahlen erstellen. Die Szene in Abbildung 2.2 besteht nur aus der Kuh, welche 5.804 Dreiecke hat, es gibt keine Reflexion und nur eine Lichtquelle. Somit muss der Raytracing Algorithmus $2.073.600 \cdot 5.804 = 12.035.174.400$ Schnittberechnungen für die Primärstrahlen durchführen. Etwa 269.840 Strahlen treffen die Kuh und starten Schattenstrahlen. Somit müssen weitere $1.566.151.360$ Schnittberechnungen durchgeführt werden. Pro Bild werden also etwa $13.601.325.760$ Schnittberechnungen ausgewertet.

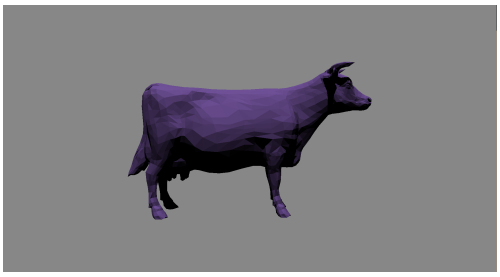

Abbildung 2.2.: Szene mit einer von oben beleuchteten lila Kuh

2.1.1. Objekte in der Szene finden

Um zu bestimmen ob ein Strahl mit Ursprung $\vec{O}$ und Richtung $\vec{D}$ ein Dreieck $\triangle_{ABC}$ trifft, nutzt der hier vorgestellte Raytracer eine auf Laufzeit optimierte Funktion von Möller und Trumbore [8]. Die Funktion von Möller und Trumbore verwendet baryzentrische Koordinaten $(u, v, w$ wobei $w = 1-u-v)$. Diese Koordinaten gelten für und beschreiben nur die Fläche eines bestimmten Dreiecks. Ein Schnittpunkt P mit dem Dreieck $\triangle_{ABC}$ erzeugt drei Teildreiecke mit den ursprünglichen Eckpunkten. Eine baryzentrische Koordinate ist das Verhältnis der Fläche eines Teildreiecks zur Gesammtfläche. Für die Berechnung der Unbekannten u, v, t, wobei t die Distanz zwischen Ursprung des Strahls und dem Schnittpunkt P ist, wird eine Determinante benötigt. Diese wird mittels Spatprodukt und den Kanten $\vec{E_1} = \vec{B} - \vec{A}$ und $\vec{E_2} = \vec{C} - \vec{A}$ bestimmt.

$$det = (\vec{D} \times \vec{E_2}) \cdot \vec{E_1} \tag{2.1}$$

Wenn $\vec{T} = \vec{O} - \vec{A}$ ist, können die Unbekannten berechnet werden (Formel 2.2).

$$u = \frac{(\vec{D} \times \vec{E_2}) \cdot \vec{T}}{det}, v = \frac{(\vec{T} \times \vec{E_1}) \cdot \vec{D}}{det}, t = \frac{(\vec{T} \times \vec{E_1}) \cdot \vec{E_2}}{det} \tag{2.2}$$

Siehe auch Abbildung 2.3

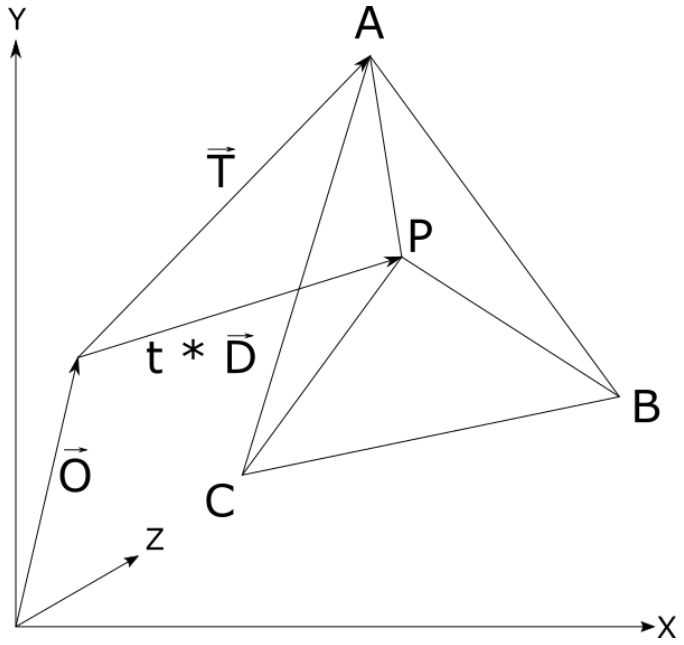

Abbildung 2.3.: Veranschaulichung baryzentrischer Koordinaten

Der Algorithmus von Möller und Trumbore ist so optimiert, dass vor dem Berechnen der ersten Unbekannten mit der Determinanten geprüft wird, ob der Strahl parallel zu dem Dreieck ist. Wenn der Winkel zwischen der Richtung des Strahls und der Normalen 90° ist, dann ist der Strahl paralell zum Dreieck und wird dieses auch nicht treffen. Nach dem Berechnen des Parameters u wird dieser geprüft. Alle baryzentrischen Koordinaten müssen die Bedingung $0 < u < 1$ bzw. $0 < v < 1$ erfüllen. Wenn u die Bedingung erfüllt, wird v berechnet. Es wird dann auf die Bedingung $u + v < 1$ geprüft. Die Bedingungen für u und v prüfen, ob sich der Schnittpunkt innerhalb des Dreiecks befindet. Zuletzt wird t berechnet und geprüft, ob $t > 0$ ist, da sich der Schnittpunkt ansonsten hinter dem Augpunkt befindet. Sollte eine der Bedingungen nicht erfüllt sein, wird das Dreieck abgelehnt und der Algorithmus kann weiter laufen. Wenn ein Dreieck getroffen wurde, kann dessen Farbe ausgelesen und als Farbe des Pixels gesetzt werden. Sollte kein Schnitt gefunden werden, wird der Pixel in der Farbe des

Hintergrunds dargestellt. Dies erzeugt ein einfaches Bild ohne Schatten und ohne Tiefe siehe Links in Abbildung 2.4.

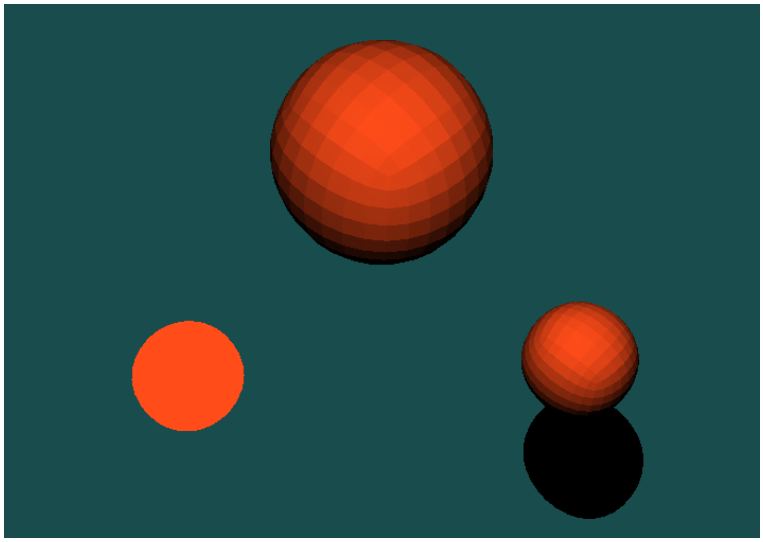

Abbildung 2.4.: Links: Objekt getroffen; Mitte: Leuchtdichte; Rechts: Schatten

2.1.2. Leuchtdichte

Wenn ein Treffer bestätigt wurde, wird der Winkel zwischen der Normalen am Schnittpunkt und der Richtung der Lichtquelle berechnet. Dieser Winkel beschreibt wie deutlich der Punkt von der Lichtquelle erleuchtet wird (s. Abbildung 2.5). Dieses Phänomen wird beschrieben durch das Lambertsches Gesetz [6] und ist in der Computergrafik bekannt als die diffuse Komponente des Phong-Beleuchtungsmodells [11]. Mit der Normalen $\vec{N}$ und dem Richtungsvektor zur Lichtquelle $\vec{L}$ wird der Winkel θ berechnet (Formel 2.3).

$$\theta = \cos^{-1}\left(\frac{\vec{N} \cdot \vec{L}}{|\vec{N}| \cdot |\vec{L}|}\right) \tag{2.3}$$

Da es sich bei $\vec{N}$ und $\vec{L}$ um normalisierte Vektoren handelt folgt $|\vec{N}| = |\vec{L}| = 1$. Daher kann die Formel 2.3 vereinfacht werden.

$$\theta = \cos^{-1}(\vec{N} \cdot \vec{L}) \tag{2.4}$$

Für den Algorithmus wird nicht θ sondern $\cos(\theta)$ benötigt. Somit vereinfacht sich die Formel weiter.

$$\cos(\theta) = \vec{N} \cdot \vec{L} \tag{2.5}$$

Sollte ein Strahl der Lichtquelle in einem Winkel von größer als $90°$ zur Normalen auf die Oberfläche treffen, dann liegt diese im Schatten. Da der Kosinus bei einem Winkel von $0 - 90°$ zwischen 1 und 0 liegt, kann der Kosinus als Gewicht für die Farbe verwendet werden. Zu sehen in Abbildung 2.4 Mitte und in Abbildung 2.5.

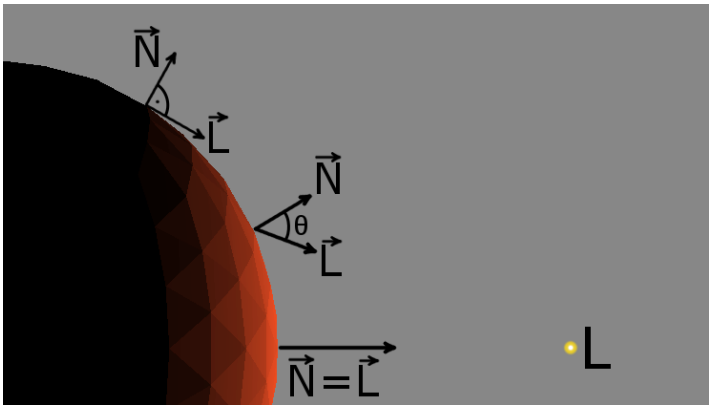

Abbildung 2.5.: Leuchtdichte mittels Lambertschen Gesetzes

2.1.3. Schatten

Am Schnittpunkt des Objekts kann ein neuer Strahl erstellt werden dessen Richtungsvektor auf die Lichtquelle zeigt. Diese Art Strahl nennt man Schattenstrahl. Wie bei anderen Strahlen wird ein Schnitt mit einem Objekt gesucht. Wenn ein Schattenstrahl auf dem Weg ein weiteres Objekt trifft, so liegt der Schnittpunkt auf dem ursprünglichen Objekt im Schatten und der Farbwert des Pixels muss modifiziert werden. Hierbei ist zu beachten, dass die Schatten dann noch hart sind (s. Abbildung 2.4 Rechts). Um weiche Schatten zu erhalten, muss der Raytracer mehrere Schattenstrahlen erzeugen und auswerten, deren Ziele um die Lichtquelle gestreut sind. Hier werden unter anderem Raster oder zufällig bestimmte Punkte verwendet. Bei mehreren Lichtquellen muss außerdem beachtet werden, dass die Ergebnisse aller Schattenstrahlen entsprechend verrechnet werden.

2.1.4. Reflexion und Brechung

Wenn ein Objekt getroffen wird, dessen Oberfläche reflektiert, wird ein weiterer Strahl in die reflektierte Richtung gestartet. Mit der Richtung $\vec{d}$ des Strahls und der Normalen $\vec{n}$ der Oberfläche ist die Formel für das Berechnen der Richtung $\vec{r}$ des reflektierten Strahls folgende:

$$normalize(\vec{r}) = \vec{d} - 2(\vec{d} \cdot \vec{n})\vec{n} \tag{2.6}$$

Diese reflektierten Strahlen werden ausgewertet wie die Primärstrahlen vom Augpunkt. Ihr Ergebnis wird dann je nach Reflexionsintensität mit dem bisherigen Ergebnis verrechnet.

Brechung wie zum Beispiel bei Glas oder einer Wasseroberfläche funktioniert im Prinzip ähnlich wie Reflexion. Die Richtung ist im Gegensatz zur Reflexion durch das Objekt hindurch und wird vom Brechnungsindex beeinflusst. Wenn n bzw n' der Brechungsindex des jeweiligen Mediums und $\vec{n}$ der Normalenvektor ist, kann mit der Strahlrichtung $\vec{d}$ die Brechungsrichtung $\vec{b}$ berechnet werden.

$$n(\vec{d} \times \vec{n}) = n'(\vec{b} \times \vec{n}) \tag{2.7}$$

Die Vektoren sind stets komplanar zueinander.

2.2. Andere Arbeiten

Neben vielen Optimierungen des Raytracing Algorithmus wurde auch optimierte Hardware entwickelt. Ein Beispiel dafür ist der SaarCOR [12], der eine gute Echtzeit-Leistung erzielen kann. Mit dem SaarCOR wurde versucht einen Ansatz zur Optimierung des Algorithmus auf Hardware Ebene zu unterstützen, was zu einer erheblichen Performance Verbesserung führte. Der SaarCOR Raytracer erreicht bei komplexen Szenen mit mehr als 30.000 Dreiecken eine Leistung von mehr als 25 FPS [12].

3. Konzept

In diesem Kapitel werden Konzepte vorgestellt, die es ermöglichen sollen einen Raytracer in Echtzeit zu verwenden. Der größte Rechenaufwand beim Raytracing ist die Schnittberechnung mit den Objekten der Szene. Um dieses Problem zu bewältigen wird eine Möglichkeit zur Reduzierung der Anzahl an Schnittberechnungen vorgestellt. Des Weiteren wird eine Möglichkeit zur Parallelisierung des Algorithmus und die Verwendung der GPU mittels CUDA gezeigt.

3.1. Bounding Volume (BV)

Ein BV (zu deutsch Hüllkörper) ist ein primitiver Körper, der einen komplexeren umschließt. BVs finden ihre Anwendung vor allem bei der Kollisionsberechnung und beim Raytracing. Der Vorteil von BVs ist, dass die Schnittberechnung mit einem anderen Objekt oder einem Strahl in der Regel schneller erfolgt. Bei einer einfachen Schnittberechnung von Strahl und Objekt wird eine Berechnung für jedes Dreieck des Objekts ausgeführt. Durch den Schnitt mit dem BV des Objekts kann mit nur einer Schnittberechnung entschieden werden, ob das Objekt potenziell getroffen werden kann. Ein Treffer bei dem BV ist allerdings noch keine Garantie für das Treffen des Objekts. Dafür müssen weitere Berechnungen erfolgen. Es gibt verschiedene Arten von BVs (s. Abbildung 3.1): Eine Sphere, die das Objekt einfach umschließt. Die Oriented Bounding Box (OBB), eine Box so gedreht, dass sie das Objekt so eng wie möglich umschließt. Die Axis Aligned Bounding Box (AABB), eine Box deren lokale Achsen parallel zu den globalen sind. Es existieren noch weitere Arten von BVs wie z.B. k-DOPs, welche eine Erweiterung der AABBs sind [14]. Je besser das BV passt, desto höher ist die Wahrscheinlichkeit bei einem BV Treffer ebenfalls einen Treffer bei dem Objekt zu bestätigen. Besser passende BVs haben höhere Kosten in der Schnittberechnung. BVs die einfacher auf einen Treffer zu testen sind, passen meistens nicht sehr eng um das Objekt. In dem hier vorgestellten Verfahren werden AABBs verwendet, da diese günstig in der Erstellung, dem Speicherplatz und der Schnittberechnung sind.

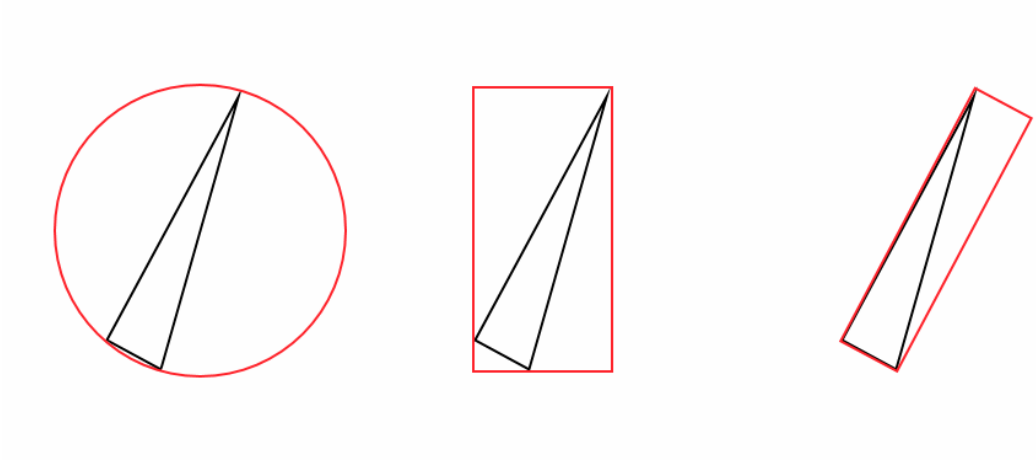

Abbildung 3.1.: Beispiele für Hüllkörper (links Sphere, mitte Axis Aligned Bounding Box, rechts Oriented Bounding Box)

3.2. Bounding Volume Hierarchie (BVH)

Eine BVH ist eine baumartige Struktur, deren Knoten BVs sind. Diese werden gruppiert und als innere Knoten von neuen BVs umschlossen. Jeder innere Knoten kann als eine eigene BVH angesehen werden. Durch Fortsetzen dieses Verfahrens entsteht ein BV, welches das gesamte Objekt umschließt. Dies ist der Wurzelknoten der BVH. Durch eine solche BVH kann die Szene wesentlich besser verarbeitet werden und die BVH bleibt bestehen, solange sich die Szene nicht verändert. Sollte sich die Szene verändern, könnte die BVH durch das Aktualisieren der Ausdehnung der betroffenen BVs weiter genutzt werden. Dabei verliert die BVH an Qualität, da die BVs nicht mehr optimal sind. Eine andere Möglichkeit ist, die BVH komplett neu zu erstellen.

In dieser Arbeit wird eine BVH mit einem top-down Verfahren erstellt. Bei der top-down Erstellung werden die Primitiven immer weiter aufgeteilt und dem rechten oder linken Teil des Binärbaums zugewiesen. Die Aufteilung erfolgt entweder am räumlichen Median oder wird mit der Surface Area Heuristic (SAH) entschieden [7]. Mit der SAH werden verschiedene Aufteilungen entlang der Achsen geprüft und die Aufteilung mit dem besten Ergebnis umgesetzt. BVHs die mit der SAH erstellt werden, haben eine bessere Qualität als die, die mit dem räumlichen Median erstellt wurden. Eine bessere Qualität reduziert die Anzahl der gesamt benötigten Schnittberechnungen des Algorithmus.

3.3. CUDA®

CUDA (Compute Unified Device Architecture) ist eine der Programmier-Techniken, die es ermöglicht Programme zu schreiben, welche dann von der GPU verarbeitet werden können. CUDA wird von der NVIDIA Corporation entwickelt. Die Rechenleistung von heutigen Grafikkarten, welche in Privat-Rechnern verbaut sind, ist groß und mit CUDA oder der Alternative OpenCL können auch Programme, die normalerweise nicht die GPU verwenden, auf diese Resource zurückgreifen. Die Stärke von CUDA ist die Möglichkeit der parallelen Berechnung. Die GPU ist darauf ausgelegt daten-parallele Berechnungen möglichst effizient zu verarbeiten. Mit den Hardware Generationen Tesla Kxx, GTX 8xx, and GTX 9xx wurde die Schwäche der floating-point Präzision bei NVIDIA Hardware auf den Standard IEEE 754 gebracht [3, 9]. Ein Nachteil der GPU gegenüber der CPU ist die Anbindung mittels PCIe, wodurch der Transfer von Daten von und zu der GPU zum Bottleneck wird. CUDA liefert neben einer Vielzahl an Beispielen ebenfalls einen Profiler und Monitor, welche ohne größere Schwierigkeiten verwendbar sind. Karimi et al. zeigen, dass unter der Verwendung von NVIDIA Hardware mit CUDA eine bessere Leistung erzielt werden kann als mit OpenCL [4]. In dieser Arbeit wurde CUDA und der C# Wrapper ManagedCUDA verwendet. Mit ManagedCUDA kann der Host-Code in C# geschrieben werden und den Kernel aufrufen.

3.3.1. Entwicklung von CUDA Programmen

Um ein Programm auf der Grafikkarte ausführen zu können, wird ein Kontext erzeugt, in dem dieses Programm aufgerufen werden kann. Ein Kontext hat ein Grid, in dem die Programme unterteilt in Blöcke liegen und auf GPU Zeit warten. Grid und Block besitzen drei Dimensionen. Ein Programm, das mit CUDA ausgeführt, wird bezeichnet man als Kernel und wird in CUDA-C geschrieben. CUDA-C ist eine CUDA spezifische Variante von C mit ausgewählten Features aus C++. NVIDIA stellt außerdem einen Compiler zur Verfügung um Kernel und Host Code zu kompilieren. Wenn ein Kernel aufgerufen wird, muss spezifiziert werden, wie groß das Grid und die Blöcke sind. Für jeden Platz in einem Block werden Threads erstellt, die den Programmcode des Kernel ausführen. Jeder Thread hat Zugriff auf den globalen Speicher (global memory), den Texturspeicher (texture memory) und geteilten Speicher (shared memory) (s. Abbildung 3.2). Kommunikation zwischen Threads kann implementiert werden, wird aber nicht empfohlen. Um beste Performance zu erreichen sollte jeder Thread unabhängig von anderen arbeiten. Threads teilen sich shared memory, sofern sie sich im selben Block befinden. Der Texturspeicher ist besonders schnell bei Leseoperationen, da er mit einem Cache versehen ist. Es ist auch möglich zur Laufzeit in den Texturspeicher zu schreiben, sollte dies erwünscht sein. Für das Lesen aus

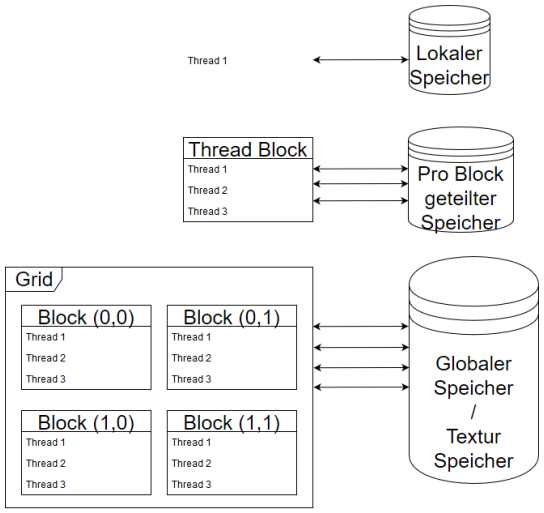

Abbildung 3.2.: Übersicht des GPU Speichers in CUDA

dem Texturspeicher liefert das CUDA Toolkit Texturobjekte, die dem Kernel nach Übertragen der Daten als Parameter übergeben werden. In den Texturspeicher schreiben und lesen kann nur mit einem Surfaceobjekt erzielt werden. Bevor ein Kernel aufgerufen wird, werden alle benötigten Daten in den GPU-Speicher übertragen. Nachdem CUDA alle Threads abgearbeitet hat, kann das Ergebnis aus dem GPU-Speicher zurück in den Host-Speicher transferiert werden.

3.3.2. Raytracing mit CUDA

Jeder Pixel wird vom Raytracing Algorithmus unabhängig von den anderen berechnet. Diese Unabhängigkeit der Pixel kann als Ansatz zur parallelisierung des Algorithmus verwendet werden. Der Raytracing Kernel wird in einem Grid ausgeführt, welches das zu erstellende Bild repräsentiert. Im Rahmen dieser Arbeit haben Experimente ergeben, dass ein Block mit einer Größe von $16 \times 16 \times 1$ optimal ist. Somit hat das Grid eine Größe von $\frac{width}{16} \times \frac{height}{16} \times 1$. Jeder Thread kann zur Laufzeit bestimmen, wo er im Grid positioniert ist.

```
1    x_coord = blockIdx.x * blockDim.x + threadIdx.x;
2    y_coord = blockIdx.y * blockDim.y + threadIdx.y;
```

Die Variablen blockIdx, blockDim und threadIdx werden jedem Thread zur Laufzeit bereitgestellt. Die Variable blockIdx ist ein Vektor mit dem X-, Y- und Z-Index des Blocks, in dem sich der Thread befindet. Die Variable blockDim hat die Werte, die bei der Definition der Blockgröße angegeben wurden. In threadIdx stehen die Indices X, Y und Z des Threads innerhalb des Blocks. Da jeder Thread einen Pixel im zu berechnenden Bild repräsentiert, sind diese

Koordinaten auch die Koordinaten des Pixels. Ein Bild hat nur zwei Dimensionen somit kann die Z-Koordinate ignoriert werden. Mit den Koordinaten kann der Thread dann einen ersten Strahl erstellen und sein Ergebnis in den Speicher an der richtigen Position ablegen.

4. Umsetzung

Hier wird der im Verlauf dieser Bachelorarbeit implementierte Raytracer vorgestellt. Das Bild 4.1 zeigt ein Diagramm der implementierten Module.

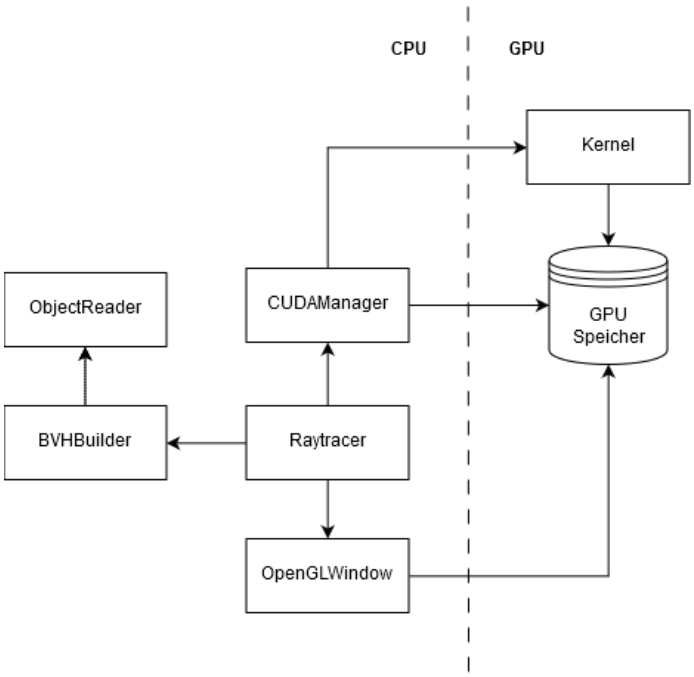

Abbildung 4.1.: Modulübersicht des Raytracers

4.1. Raytracer

4.1.1. Kamera und Strahl

Jeder Primärstrahl hat seinen Ursprung im Augpunkt bzw. Linsenpunkt der Kamera. Im folgenden Pseudocode wird das Erstellen eines Strahls erläutert. Die Kamera hat eine Position, eine Ausrichtung und einen Öffnungswinkel. Veränderungen an der Kamera werden auf der CPU berechnet und dann auf die GPU kopiert. Für das Erstellen eines Primärstrahls werden einige

Parameter benötigt. Diese Parameter werden beim Initialisieren oder Verändern der Kamera auf der CPU berechnet und dann in den Speicher der GPU transferiert. Der Pseudocode in 4.1 zeigt das Erstellen auf der GPU.

```
function createRay(camera, x, y, width, height) {
    α = x/width
    β = y/height

    xDirection = camera.xDirection · α · camera.xScale
    yDirection = camera.yDirection · β · camera.yScale

    direction = camera.direction + xDirection + yDirection
    origin = camera.position

    return new Ray(direction, origin)
}
```

Listing 4.1: Erstellung von Strahlen

4.1.2. Dreieck und Strahl

Für die Berechnung eines Schnittpunkts zwischen Strahl und Dreieck verwendet der hier vorgestellte Raytracer eine auf Laufzeit optimierte Methode vorgestellt von Tomas Möller und Ben Trumbore [1]. Die Methode ist dahingehend optimiert, dass sie nicht treffende Strahlen früh ablehnt und somit einige teure Operationen vermeidet. Die Methode bricht die Schnittpunktberechnung ab, sobald bestimmt werden kann, dass der Strahl parallel zum Dreieck ist. Sollte Parameter u bzw v größer 1 oder kleiner 0 sein, dann liegt der Schnittpunkt außerhalb der Fläche des Dreiecks (s. Pseudocode 4.2).

```
function intersectTriangle(ray, triangle) {
    e1 = p1 − p0
    e2 = p2 − p0
    q = d × e2
    α = e1 · q
    if (α > −ϵ and α < ϵ) return false
    f = 1/α
    s = o − v0
    u = f(s · q)
    if (u < 0.0) return false
    r = s × e1
    v = f(d · r)
    if (v < 0.0 or u+v > 1.0) return false
    t = f(e2 · q)
    return true, t
}
```

Listing 4.2: Möller-Trumbore ray triangle intersection

4.2. BVH

4.2.1. BVH erstellen

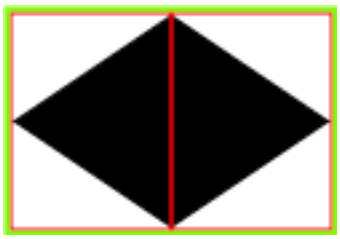

Abbildung 4.2.: BVH

Das Erstellen der BVH wird auf der CPU ausgeführt. Danach wird sie in den Texturspeicher der GPU geladen, um dort für die Berechnung des Bildes zur Verfügung zu stehen. Der hier vorgestellte Algorithmus zum Erstellen der BVH erzeugt nicht die bestmögliche BVH. Es wird auch nicht sichergestellt das die BVH balanciert ist. Auf die Suche nach der besten BVH wird verzichtet, da dies eine hohe Laufzeit beanspruchen würde. Die Arbeiten von Wald et al. [13, 5, 14] haben gezeigt, dass eine Surface Area Heuristic (SAH) basierte BVH mit geringerer Qualität immer noch eine gute Performance erzielen kann. Das Aufbauen der BVH wurde als rekursive Funktion implementiert, die die Liste der Dreiecke immer weiter unterteilt (s. Pseudocode in 4.3). Um eine guten Aufteilung der Dreiecke zu finden, werden alle Achsen des BVs untersucht, welches unterteilt werden soll (s. Pseudocode in 4.4). Die SAH wurde für diesen Algorithmus

vereinfacht. Mit $x_{dim} = x_{max} - x_{min}$, $y_{dim} = y_{max} - y_{min}$, $z_{dim} = z_{max} - z_{min}$ und t als Anzahl an Dreiecken innerhalb des AABB kann die SAH berechnet werden.

$$SAH(AABB) = (x_{dim} \cdot y_{dim} + y_{dim} \cdot z_{dim} + z_{dim} \cdot x_{dim}) \cdot t \tag{4.1}$$

```
function RecursiveBuild(List BoundingBoxList) {
    v^min = min(BoundingBoxList)
    v^max = max(BoundingBoxList)

    if(count(BoundingBoxes) < 4) return Leaf

    {LeftList, RightList} = FindBestSplit(BoundingBoxList)

    if(no better split found) return Leaf

    LeftChild = RecursiveBuild(LeftList)
    RightChild = RecursiveBuild(RightList)

    return Inner
}
```

Listing 4.3: Rekursive BVH Erstellung (Pseudocode)

```
function FindBestSplit(List BoundingBoxes) {
    minimumCost = SAH(BoundingBoxes)
    foreach(Axis ∈ {X, Y, Z}) {
        {LeftList, RightList} = split(BoundingBoxes,distance)
        if(SAH(LeftList)+SAH(RightList) < minimumCost) {
            remember split
        }else {
            increase distance
        }
    }
    if(no improvement found) return no better split

    return {LeftList, RightList}
}
```

Listing 4.4: Links Rechts Split der BoundingBox Liste

4.2.2. BVH Darstellung im GPU Speicher

Der Texturspeicher von CUDA unterstützt 1D, 2D und 4D Datentypen. Um die BVH im GPU Speicher darzustellen werden drei Texturen verwendet (s. Abbildung 4.3). In der ersten Textur werden Vektoren gespeichert, die die Position und Ausdehnung eines BVs beschreiben. Die zweite Textur besteht aus ganzzahligen Vektoren. Diese Vektoren speichern die Information, ob es sich um ein Blattknoten handelt und Referenzen auf Kindknoten bzw. enthaltene Dreiecke. In der dritten Textur werden die Indizes der in einem BVH Blattknoten enthaltenen Dreiecke gespeichert. Die IDs sind so sortiert, dass alle in einem BV enthaltenen IDs gruppiert sind.

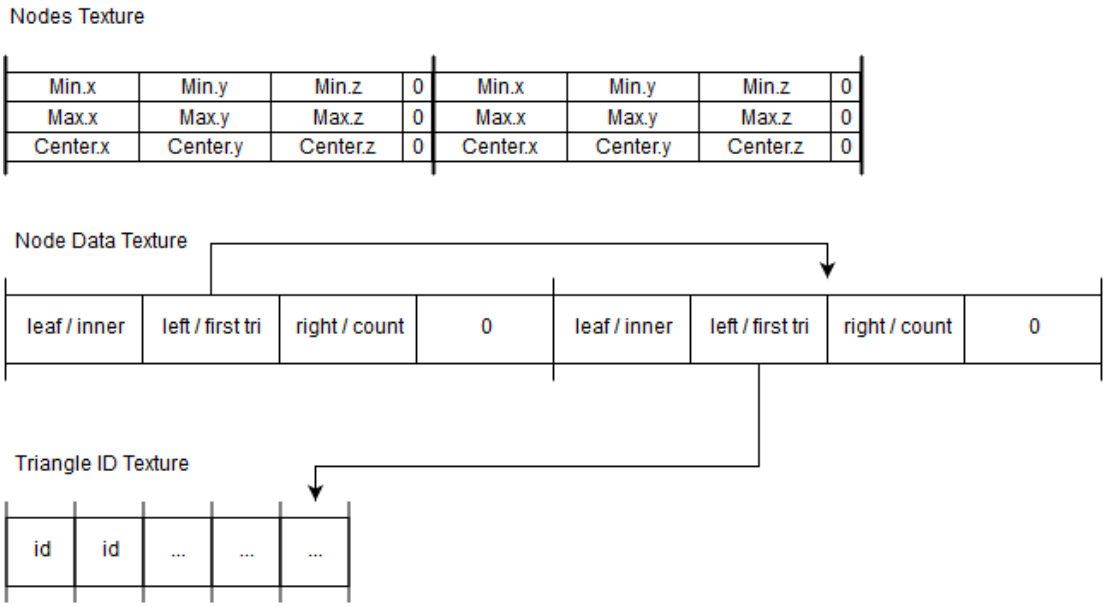

Abbildung 4.3.: Speicher auf der GPU

4.2.3. BVH und Strahl

Möller et al beschreiben eine Funktion, die den Schnitt zwischen Strahl und AABB prüft und dies mit möglichst geringem Aufwand [1]. Diese Funktion kann allerdings nicht die Distanz des Schnittes bestimmen. Im Pseudocode 4.5 wird die Funktion für den Schnitt Strahl und OBB auf den Sonderfall der AABB optimiert in Pseudocode beschrieben.

Jeder Strahl wird als erstes gegen den Wurzelknoten getestet (veranschaulicht in 4.4). Sollte der Strahl den Wurzelknoten nicht treffen, wird der Strahl mit keinem Objekt der Szene kollidieren. Bei einem Treffer werden beide Kindknoten getestet. Der nächstliegende Treffer wird

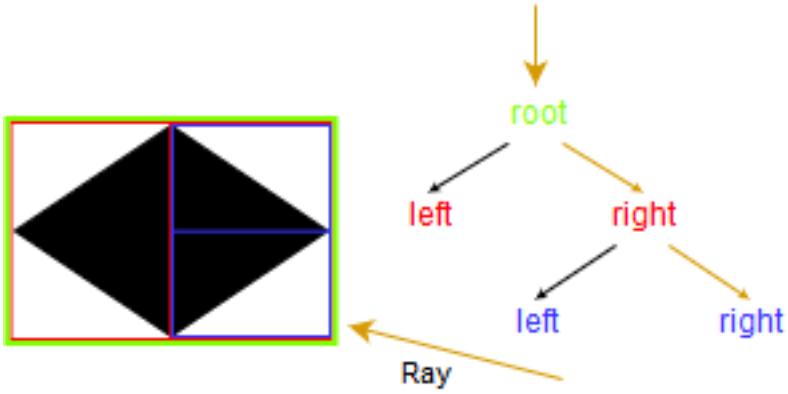

Abbildung 4.4.: BVH intersection

dann weiter untersucht, bis ein Blattknoten gefunden wird. Wenn ein Blattknoten gefunden wird, werden alle in diesem Blattknoten enthaltenen Dreiecke auf einen Treffer untersucht.

```
function intersectAABB(Ray, AABB) {
    t^min = −∞
    t^max = ∞
    p = a^c − o
    for each i ∈ {u, v, w}
        e = p^i
        f = d^i
        if(|f| > ε)
            t1 = (e + h_i)/f
            t2 = (e − h_i)/f
            if (t1 > t2) swap(t1, t2)
            if (t1 > t^min) t^min = t1
            if (t2 < t^max) t^max = t2
            if (t^min > t^max) return false
            if (t^max < 0) return false
        else if (−e − h_i > 0or − e + h_i < 0) return false
    if (t^min > 0) return true, t^min
    return true t^max
}
```

Listing 4.5: Ray AABB intersection

4.3. Anzeige

Das Ergebnis eines Pixels speichert der Raytracer in einem CUDA Surface-Objekt. Der Vorteil bei der Verwendung eines Surface-Objekts ist, das es im Textur Speicher der GPU liegt. Das Bild könnte nach einem Programm Durchlauf von der GPU zurück kopiert werden um es zu speichern oder anzuzeigen. Dieser Vorgang nimmt viel Zeit in Anspruch. Für eine Echtzeit-Anwendung, bei der das Bild immer aktuell angezeigt werden soll, wird das Bild zur Anzeige auf die GPU übertragen. Um diesen Vorgang zu beschleunigen, werden die Bilder im GPU Speicher belassen. Die Anzeige erfolgt in einem OpenGL Fenster, welches eine 3D Szene zeigt. Diese Szene besteht nur aus einem Objekt, einem Plane. Das Plane der OpenGL Szene hat eine Textur, welche im GPU Speicher bereit liegt. Die Textur ist das Ergebnis des Raytracers. Die Abbildung 4.5 zeigt eine Szene im Raytracer und die dazugehörige Darstellung in der OpenGL Szene und die Textur im Speicher.

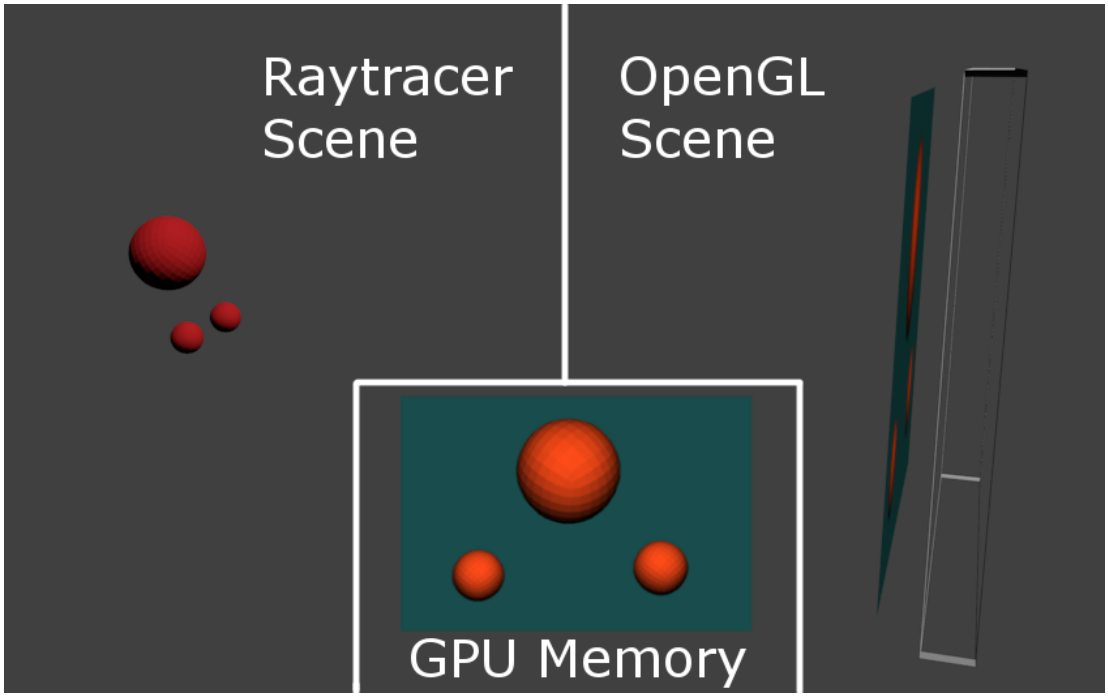

Abbildung 4.5.: Echtzeit-Anzeige des Raytracers

4.4. CUDA Manager

Da sich bei CUDA der Programmierer um die Speicherverwaltung zu kümmern hat, wurde für diesen Raytracer ein CUDAManager implementiert. Dieser CUDAManager initialisiert und verwaltet GPU-Variablen. Nach dem Beenden des Programms gibt der CUDAManager den belegten Speicher wieder frei.

4.5. Objekt Importer

Der Object Reader des Raytracers kann das Wavefront OBJ Format lesen und in die interne Darstellung umwandeln. Die interne Darstellung der Szene des Raytracers ist ähnlich zu dem OBJ Format. Es wird eine Liste an Vertices und eine Liste an Dreiecken gespeichert. Die Vertices sind Koordinaten der Punkte und die Dreiecke verweisen auf drei Vertices, die das Dreieck bilden.

5. Performanceanalyse

In diesem Teil wird die Performance des Raytracers analysiert. Zunächst werden einige Experimente ausgeführt, mit denen das Antwortzeitverhalten in Form von frames per second (FPS) untersucht werden soll. Als GPU für diese Experimente wurde eine NVIDIA GTX 1060 mit 6 GB Speicher verwendet. Die GPU wurde während des Tests ebenfalls als primäres Bildausgabegerät verwendet. Für alle Tests wurden die Test-Objekte in den Mittelpunkt der Szene platziert. Es gibt genau eine Lichtquelle, die von oben die Szene beleuchtet. Während der Tests wurden nur harte Schatten und keine Reflexion oder Brechung berechnet, da die Performance bereits ohne diese Features nicht ausreichend war. Die Kamera wurde beim Auswerten der FPS nicht bewegt.

Nachfolgend werden mögliche Schwachstellen des Programms mittels Ergebnissen eines Profilers gesucht. Hierfür wurde der NVIDIA Nsight Profiler verwendet.

5.1. Performance-Messung mit zunehmender Anzahl von Dreiecken

Dieses Experiment wurde eine quadratische Fläche als Objekt in die Szene platziert. Die Fläche wurde in vier Teile unterteilt und besteht somit aus acht Dreiecken. Um die Auswirkung der Anzahl der Dreiecke auf die FPS zu zeigen, wurde die Fläche immer weiter unterteilt. Dies hat zur Folge, dass sich die Anzahl der Dreiecke mit jedem Schritt vervierfacht. Das Experiment wurde mit verschiedenen Auflösungen ausgeführt. Da ein deutlicher Trend der Kurvenverläufe zu erkennen ist, wurde in Abbildung 5.1 nur eine Auswahl dargestellt. Die vollständigen Messergebnisse sind in Appendix A.1 zu finden.

Der Abbildung 5.1 ist zu entnehmen, dass die BVH im Bereich von 8 bis 128 Dreiecken dem zusätzlichen Aufwand pro Dreieck kaum entgegen wirken kann. Erst bei 512 Dreiecken haben mehr Dreiecke nur eine geringe Auswirkung auf die FPS. Für die niedrigste Auflösung wurde die Anzahl der Dreiecke ausgereizt und selbst mit über 32k Dreiecken lieferte der Raytracer noch ~45FPS. Bei mehr als 100k Dreiecken dauerte das Erstellen der BVH etwa fünf Minuten und der Datentransfer auf die GPU wurde abgelehnt.

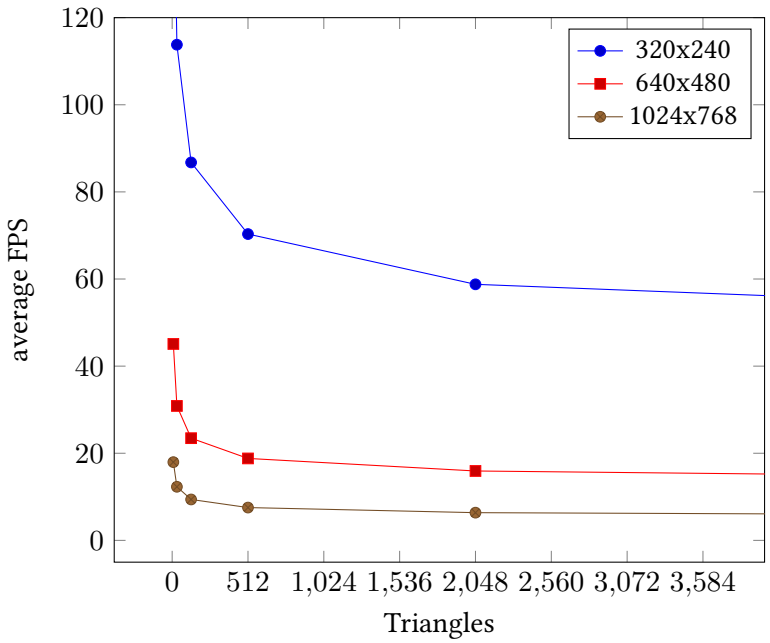

Abbildung 5.1.: Verhältnis von Anzahl an Dreiecken zu FPS

5.2. Performance-Messung mit verschiedenen Objekten

Für die folgenden Tests wurde neben der Ermittlung der FPS zusätzlich auch die Verwendbarkeit der interaktiven Kamera getestet. Die Test-Objekte sind eine Sphere, bestehend aus 720 Dreiecken, und eine Kuh, welche aus 5804 Dreiecken besteht. Eine Auswahl der Ergebnisse der Tests wurde in Tabelle 5.1 zusammengefasst. Die vollständigen Testdaten sind in Appendix A.2 aufgeführt und wurden in Abbildung 5.4 grafisch aufbereitet. Die Sphere und die lila Kuh können in jeweils drei verschiedenen Auflösungen in Abbildung 5.2 bzw. 5.3 betrachtet werden.

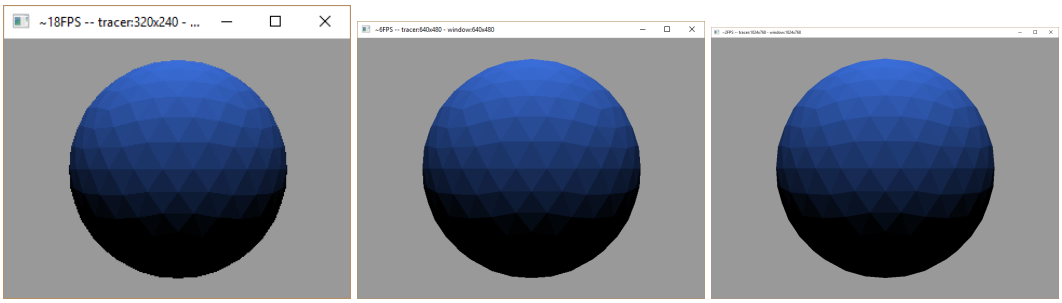

Abbildung 5.2.: Szene mit einer blauen Sphere in verschiedenen Auflösungen (von Links nach Rechts 320x240, 640x480 und 1024x768)

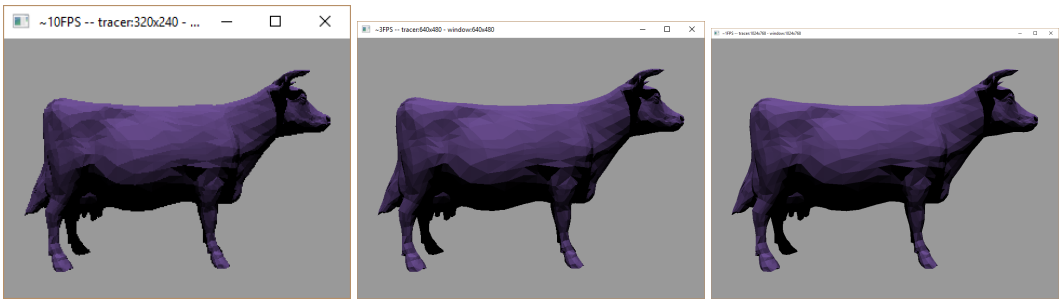

Abbildung 5.3.: Szene mit einer lila Kuh in verschiedenen Auflösungen (von Links nach Rechts 320x240, 640x480 und 1024x768)

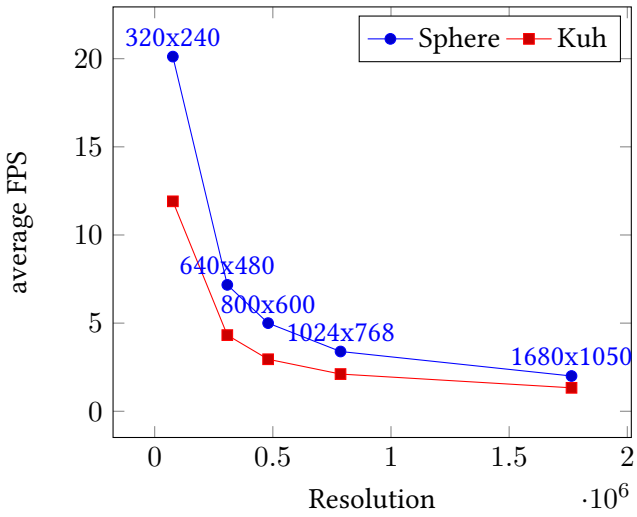

Abbildung 5.4.: Verhältnis von Auflösung(Breite x Höhe) zu FPS bei Sphere und Kuh

Tabelle 5.1.: Performance Objekte und Kamera

Auflösung	**FPS(Sphere)**	**Kamera(Sphere)**	**FPS(Kuh)**	**Kamera(Kuh)**
320x240	18	✓	12	✓
640x480	6	✗	3	✗
1024x768	2	✗	1	✗

Wenn die FPS unter 10 sinken, lässt sich eine interaktive Kamera kaum noch flüssig steuern. Es ist zu beobachten, dass der Verlauf des FPS-Einbruchs sowohl bei der einfacheren Sphere als auch bei der komplexeren Kuh ähnlich ist. In Abbildung 5.4 ist auch zu sehen, dass die Einbrüche, bei Auflösungen größer 1024x768, nicht mehr so drastisch sind.

5.3. Profiling

Mit dem NVIDIA Nsight Profiler wurden verschiedene Analysen über Performance in Bezug auf den Durchsatz, mögliche Ursachen für geringe Performance und Arithmetische Operationen durchgeführt. Die ausführliche Dokumentation des Profilers ermöglichte eine einfache Auswertung der Ergebnisse.

5.3.1. Durchsatzanalyse

Bei der Durchsatzanalyse wird gezeigt, welche Arten an und wie viele Floating Point Operationen (FLOP) der Raytracer durchführt. Des Weiteren wird gemessen, wie viele GFLOP (FLOP $\times 10^9$) der Raytracer pro Sekunde abarbeiten kann (s. Abbildung 5.5).

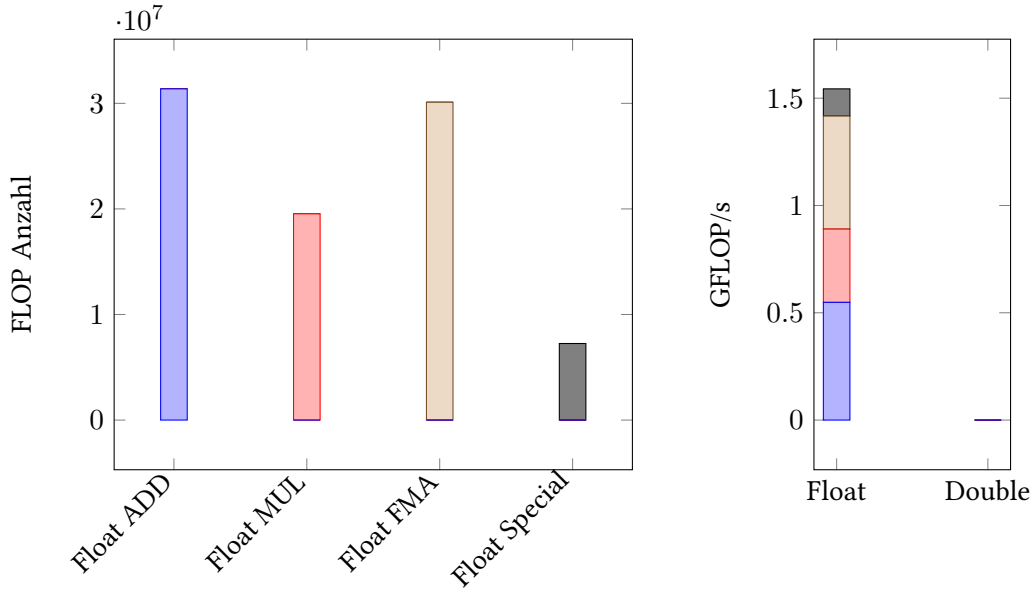

Abbildung 5.5.: CUDA Performance: Links Arten der FLOPs, Rechts GFLOP/s (Nachempfunden aus NVIDIA Nsight Profiler)

Auf der rechte Seite ist zu sehen, dass der Raytracer nur single-precision Operationen verwendet und, dass die Performance des Raytracers möglicherweise nicht optimal ist. Laut Angaben des Herstellers kann eine GTX 1060 theoretisch 4.375 GFLOP/s erzielen. Natürlich ist zu beachten, dass nicht alle Operationen des Raytraycers Floating Point Operationen sind und die 4.375 nicht erreicht werden können.

5.3.2. Problemanalyse

Die Problemanalyse zeigt mögliche Ursachen für geringe Performance (s. Abbildung 5.6). Die hier aufgeführten Probleme können dazu führen, dass Rechenzeit ungünstig verteilt wird.

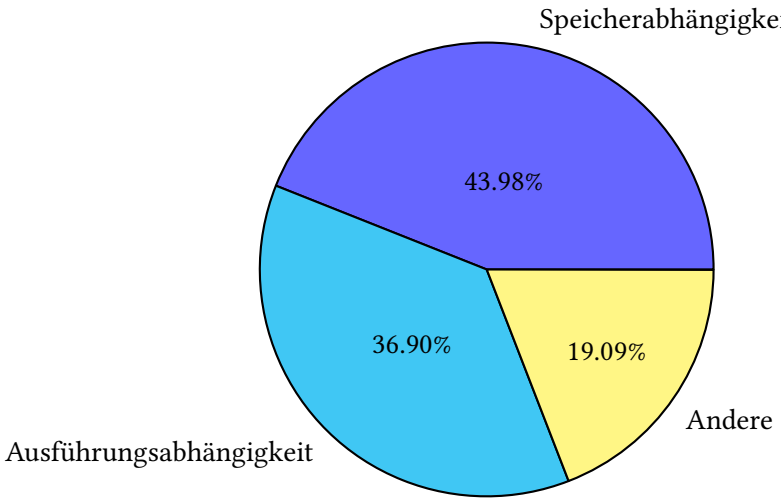

Abbildung 5.6.: CUDA Performance: Gründe für schlechte Performance (Nachempfunden aus NVIDIA Nsight Profiler)

Die häufigste Ursache, die der Profiler finden konnte, ist Speicherabhängigkeit. Dies bedeutet, dass die Threads häufig darauf warten müssen, dass Daten aus dem Speicher geladen werden. Ein Grund für ein solches Warten könnte ein anderer Thread sein, der die Resource gerade blockiert. Zu viele Zugriffe auf den Speicher können ebenfalls dieses Problem verursachen.

Die zweite häufig auftretende Ursache ist Ausführungsabhängigkeit. Wenn das Programm des Threads komplex ist, kann dies einen schlechten Einfluss auf die Performance haben. Ein zu komplexer Ablauf eines Programms kann dazu führen, dass der Scheduler der GPU nicht optimal die Rechenzeit verteilen kann.

5.3.3. Arithmetische Operationen

Diese Analyse zeigt, welche und in welchem Verhältnis arithmetische Operationen durchgeführt werden (s. Abbildung 5.7). Zu beobachten ist, dass das Verhältnis an Floating Point Operationen sehr gering ist, obwohl diese sehr häufig im Raytracing Algorithmus verwendet werden. Im Gegensatz dazu ist das Verhältnis der Integer Shift Operationen sehr hoch. Daran ist zu erkennen, dass der Compiler von NVIDIA viele Operationen optimiert und in Shift Operationen umgewandelt hat. Auch der Anteil an Integer Additionen ist sehr hoch, obwohl

viele Float Vektoren addiert werden, was ebenfalls dem Compiler zuzuschreiben ist. Die Anzahl der Integer Multiplikationen ist wie erwartet.

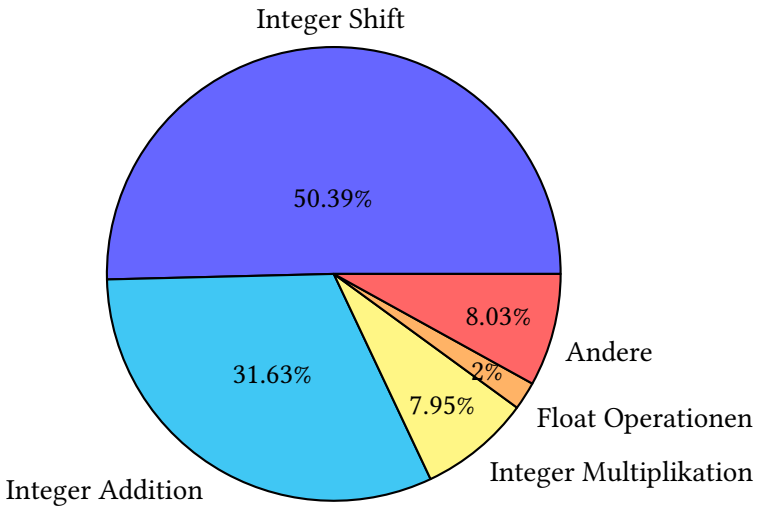

Abbildung 5.7.: Arithmetische Operationen und das Verhältnis in dem sie auftreten (Nachempfunden aus NVIDIA Nsigh Profiler)

6. Fazit und Ausblick

Die Analyse zeigt, dass der vorgestellte Raytracer keine besonders hohe Performance für hohe Auflösungen erreicht. Dennoch haben die Experimente gezeigt, dass die Anzahl der Dreiecke, mit einer Hilfsstruktur wie einer BVH, die Performance nur gering beeinflusst. Das Datenmanagement des Raytracers ist nicht optimal. Es werden einige Speicherbereiche belegt aber nicht verwendet. Dies hat zur Folge, dass bei jedem Lesen und Schreiben dieser Speicherbereich trotzdem gelesen oder geschrieben wird, da CUDA nur Texturen mit float4 unterstützt, aber nur float3 Vektoren gespeichert werden. Dies wurde mit dem Nsight Profiler ebenfalls gezeigt.

Wie wichtig eine Hilfsstruktur ist, zeigt die folgende Überlegung. Angenommen die einzigen Kosten des Raytracers sind die Kosten der Schnittberechnung, dann müsste der Raytracer bei einer Auflösung von 1080p $13.601.325.760$ Berechnungen pro Bild auswerten (s. A. Kapitel 2.1). Eine NVIDIA GTX 1060 erreicht laut Hersteller etwa 4.375 GFLOP/s, was $4.375.000.000.000$ FLOP/s entspricht. Angenommen die Schnittberechnung kann auf 20 FLOPs reduziert werden, dann folgt 20 FLOPs $\times 13.601.325.760 = 272.026.515.200$. Somit könnte die GPU $\frac{4.375.000.000.000}{272.026.515.200} = 16.08$ FPS erzielen. Für eine Echtzeit-Anwendung sollten mindestens 25 FPS erreicht werden. Dies lässt vermuten, dass ohne eine Hilfsstruktur ein Raytracer nach aktuellem Stand der Technik keine Verwendung in einer Echtzeit-Anwendung finden wird. Natürlich ist diese Überlegung ein Worst-Case-Szenario und die Realität zeigt, dass der Raytracer gar nicht so viele FLOPs braucht. Des Weiteren ist zu beachten, dass das Laden der Daten aus dem Speicher ebenfalls Zeit in Anspruch nimmt. Natürlich sind die Kosten der Schnittberechnung nicht nur FLOPs sondern auch Integer Operationen (IOP).

Viele Designentscheidungen, bei der Implementierung des Raytracers, haben sich als nicht optimal herausgestellt. Eine dieser Entscheidungen war das Implementieren des kompletten Algorithmus in einem Kernel. Andere Arbeiten zeigen, dass das Aufteilen des Algorithmus in kleinere Kernels und das Verketten ihrer Aufrufe eine bessere Performance liefern [10]. Eine weitere Möglichkeit der Verbesserung der Performance, wäre das Aufteilen des Bildes und das Verwenden von mehreren GPUs, welche dann nur einen Teil des Bildes berechnen. Da das Verwenden einer niedrigen Auflösung eine deutlich bessere Performance liefert, könnte

mehr Performance erzielt werden, wenn das zu berechnende Bild zunächst in einer geringen Auflösung erstellt wird und mit zusätzlichen Strahlen die Qualität verbessert wird.

A. Testdaten

Tabelle A.1.: Auswirkung von Anzahl an Dreiecken auf FPS - Daten

	FPS				
Triangles	**320x240**	**640x480**	**800x600**	**1024x768**	**1680x1050**
8	162.99	45.09	30.02	17.96	9.41
32	113.79	30.86	20.56	12.30	6.59
128	86.76	23.46	15.45	9.39	5.05
512	70.32	18.81	12.27	7.53	4.08
2048	58.78	15.93	10.33	6.36	3.42
8192	50.66	13.76	8.94	5.51	2.94
32768	45.07	–	–	–	–

Tabelle A.2.: Performance verschiedener Objekte - Daten

Auflösung		**Sphere-FPS**	**Kuh-FPS**
320x240	76800	20.12	11.91
640x480	307200	7.17	4.32
800x600	480000	5.00	2.95
1024x768	786432	3.39	2.11
1680x1050	1764000	2.00	1.33

Literaturverzeichnis

[1] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-time rendering*. CRC Press, 2008.

[2] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM, 1968.

[3] I. S. Committee et al. 754-2008 ieee standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008, 2008.

[4] K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[5] C. Lauterbach, S.-E. Yoon, D. Manocha, and D. Tuft. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 39–46. IEEE, 2006.

[6] K. Lüders and R. O. Pohl. *Pohls Einführung in die Physik, Band 2: Elektrizitätslehre und Optik*. Springer.

[7] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[8] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*, page 7. ACM, 2005.

[9] NVIDIA Corporation. *CUDA C Programming Guide*. 2018.

[10] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, et al. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics (TOG)*, 29(4):66, 2010.

[11] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[12] J. Schmittler, I. Wald, and P. Slusallek. Saarcor: a hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36. Eurographics Association, 2002.

[13] I. Wald. *On fast construction of SAH-based bounding volume hierarchies*. 2007.

[14] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26(1):6, 2007.

[15] T. Whitted. An improved illumination model for shaded display. In *ACM SIGGRAPH Computer Graphics*, volume 13, page 14. ACM, 1979.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 10. April 2018 Gerhard Wagner