

Friedrich-Schiller-Universität Jena  
Fakultät für Mathematik und Informatik

# Bericht des externen Praktikums

*am Fraunhofer ITWM in der Abteilung CC HPC*

vorgelegt von Markus Pawellek

Matrikelnummer: 144645  
Fachrichtung: B.Sc. Mathematik

Praktikumszeitraum: 26.November 2013 – 30.Juni 2017

Jena, 27. Juni 2018





## Kurzprofil des Fraunhofer ITWM und der Abteilung CC HPC

Das Fraunhofer ITWM hat seinen Ursprung in dem Institut für Techno- und Wirtschaftsmathematik (ITWM) der Technischen Universität Kaiserslautern, welches am 9. November 1995 durch die Arbeitsgruppe Technomathematik (AGTM) ins Leben gerufen wurde. Das Ziel war die Verbindung von Mathematik und Praxis in Anbetracht von wissenschaftlichem Anspruch und unternehmerischen Denken. Am 11. April 2000 beschloss dann der Senat der Fraunhofer-Gesellschaft die Aufnahme des ITWM zum 1. Januar 2001.

Seitdem entwickelt das Fraunhofer ITWM hauptsächlich Simulationssoftware für die Analyse und die Lösung von Problemen im Industriebereich. Dabei steht vor allem die mathematische Modellierung verschiedenster realer Prozesse, als auch die Kombination der Hochschulmathematik mit ihrer praktischen Umsetzung im Mittelpunkt. Folglich bilden die klassischen Disziplinen der angewandten Mathematik, wie zum Beispiel Numerik, Stochastik oder Optimierung, wichtige Kernkompetenzen des Institutes. Beispiele für Geschäftsfelder, auf denen das Fraunhofer ITWM arbeitet, sind Virtuelles Material- und Produktdesign, Prozesssimulation und Diagnosesysteme. In jedem dieser Gebiete reichen die angebotenen Produkte von System- und Softwarelösungen bis hin zu Beratungs- und Supportangeboten.

Seit seiner Gründung ist der Betriebshaushalt des ITWM von anfangs 1.64 Mio. € auf 21.5 Mio. € im Jahre 2016 angestiegen. Inzwischen beschäftigt das

Fraunhofer ITWM über 400 Mitarbeiter, die sich von Auszubildenden und wissenschaftlichen Hilfskräften über Doktoranden bis hin zu Arbeitern in den zentralen, wissenschaftlichen und technischen Bereichen erstrecken. Die zentrale und moderne IT-Infrastruktur, bestehend aus circa 300 Servern und mehreren Hundert Terabyte Datenspeicher, stellt sowohl Linux- als auch Windows-Architekturen zur Verfügung und bietet den Mitarbeitern damit die Möglichkeit, effizient auf einer angepassten Umgebung zu forschen. Auch resourcenintensive Simulationsrechnungen lassen sich auf dem internen Linux-Hochleistungsrechner »Beehive« durchführen. Das Fraunhofer ITWM gehört damit zur Sparspitze der Mathematik in der Industrie und ist eines der größten und erfolgreichsten Institute der Techno- und Wirtschaftsmathematik.

Das Competence Center High Performance Computing (CC HPC) ist eine Abteilung des Fraunhofer ITWM, die seit dessen Gründung besteht und sich in enger Zusammenarbeit mit industriellen und akademischen Partnern mit der Frage, wie die immer komplexer werdenden Prozessoren und Parallelrechner effizient genutzt werden können, beschäftigt. Sie stellt neben Werkzeugen zum Umgang mit Supercomputern auch komplettete Softwarelösungen her und ist eine der tragenden Säulen des Fraunhofer ITWM. Insbesondere sind hier die Themenschwerpunkte seismische Datenverarbeitung, photorealistiche Visualisierung und skalierbare parallele Programmierung zu nennen.

### Quellen:

- <https://www.itwm.fraunhofer.de/de/ueber-fraunhofer-itwm>
- <http://www.visuallogisticsmanagement.de/wp-content/uploads/2015/10/ITWM.png>
- [https://www.itwm.fraunhofer.de/en/\\_jcr\\_content/stage/stageParsys/stage\\_slide/image.img.jpg/1498487288072\\_1440x448-ITWM-beiNacht.jpg](https://www.itwm.fraunhofer.de/en/_jcr_content/stage/stageParsys/stage_slide/image.img.jpg/1498487288072_1440x448-ITWM-beiNacht.jpg)



## Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
<b>2 Hilfsmittel und Entwicklungsumgebung</b>	<b>1</b>
2.1 Versionsverwaltung mit Git . . . . .	1
2.2 CMake . . . . .	2
2.3 C/C++-Compiler . . . . .	2
2.4 Weitere Hilfsmittel . . . . .	3
<b>3 Aufgabenstellung</b>	<b>4</b>
<b>4 Vorbereitung</b>	<b>4</b>
4.1 Einlesen der Szenendaten . . . . .	4
4.2 Grafische Ausgabe . . . . .	6
4.3 Performance-Analyse . . . . .	6
<b>5 Grundlagen und der naive Algorithmus</b>	<b>7</b>
5.1 Aufbau der Kamera . . . . .	7
5.2 Strahl-Dreieck-Schnittpunkttest . . . . .	8
5.3 Raytracing und Rendering . . . . .	10
5.4 Ergebnisse . . . . .	12
<b>6 Beschleunigungsstrukturen, Optimierung und weitere Features</b>	<b>12</b>
6.1 Linear Bounding Volume Hierarchies . . . . .	12
6.2 Path Tracing . . . . .	15
6.3 Optimierungen auf der CPU . . . . .	15
<b>7 Selbsteinschätzung und Bezug zum Studium</b>	<b>17</b>



## 1 Einleitung

In der Zeit zwischen dem 26. November 2013 und dem 30. Juni 2017 arbeitete ich am Fraunhofer ITWM im Competence Center High Performance Computing (CC HPC) als Hilfswissenschaftler. Im Verlauf dieser Beschäftigung wurde mir die Möglichkeit zu Teil, mich mit diversen Bereichen der Informatik und der Mathematik, wie zum Beispiel der Programmoptimierung, dem Parallel Computing, der Computergrafik und den Monte-Carlo-Methoden, sowie auch einigen Standardwerkzeugen der Softwareentwicklung, wie zum Beispiel *Git*, *Make*, *CMake*, *OpenGL*, *Qt*, *CUDA* und *Blender*, genauer auseinanderzusetzen. Vor allem für meine eigentliche Tätigkeit, der Entwicklung von Softwarelösungen für vorgegebene Problemstellungen, zeigten sich die erlangten Fachkenntnisse und Erfahrungen in diesen Gebieten von hohem Nutzen und ermöglichen mir eine professionelle Arbeitsweise innerhalb der verschiedenen Projektgruppen. Die komplexesten Aufgabenstellungen bestanden dabei aus der Implementierung eines echtzeitfähigen Raytracers auf der GPU und CPU, der Konstruktion eines statistisch-basierten graphischen Analysewerkzeugs für seismische Daten und der Entwicklung einer Klasse, die das Einlesen des Wavefront OBJ Dateiformates ermöglicht. Für die Ausführung der genannten Aufgaben verwendete ich im Wesentlichen die Programmiersprachen *C* und *C++*.

## 2 Hilfsmittel und Entwicklungsumgebung

Ein wesentlicher Bestandteil für die Bearbeitung der Aufgabenstellungen war die Einrichtung einer passenden Entwicklungsumgebung. Diese sollte einen effizienten Arbeitsfluss ermöglichen und die Zusammenarbeit mit anderen Teams vereinfachen. Hierfür verwendete ich sowohl Methoden, die ich durch meine Tätigkeit am Fraunhofer ITWM erlangte, als auch Herangehensweisen, welche mir aus Vorlesungen bekannt waren.

### 2.1 Versionsverwaltung mit Git

Ein typisches Werkzeug für eine Entwicklungsumgebung ist ein Versionskontrollsystem (VCS, engl.: *version control system*). Es protokolliert die Änderungen gegebener Dateien im Verlauf der Zeit. Auf jeden protokollierten Zeitpunkt (engl.: *commit*) kann zugegriffen werden, so dass auch nach dem Löschen von Informationen diese immer noch abrufbar sind. Ein VCS stellt damit eine effiziente Alternative des herkömmlichen Backups dar. Die Aufgabe des VCS wurde in meinem Projekt durch das Programm *Git* übernommen. Es ist ein verteiltes Open-Source-VCS, welches durch seine Arbeitsweise verschiedene Vorteile gegenüber anderen Systemen aufweist.

Mit *Git* erstellte ich für jedes meiner Projekte ein sogenanntes Archiv (engl.: *repository*). Diese Archive beinhalteten ein derzeitiges Arbeitsverzeichnis, sowie alle protokollierten Zeitpunkte mit deren zugehörigem Inhalt. Um diese auf mehreren Rechnersystemen bearbeiten zu können, verwendete ich die bekannten öffentlichen *Git*-Server *GitHub* und *GitLab*.

Um anderen Entwicklern das Verstehen meiner geleisteten Arbeit zu vereinfachen, strukturierte ich mein Projekt nach festen Regeln. *Commit*-Nachrichten, die jedem protokollierten Zeitpunkt eine Bedeutung gaben, mussten ein spezielles Format aufweisen<sup>1</sup>. Abbildung 1 zeigt die Verwendung dieses Formats anhand eines realen Beispiels.

---

<sup>1</sup><https://chris.beams.io/posts/git-commit/>

Implement first recursive version of Morton LBVH 

lyrahgames authored 6 months ago

This commit introduces the building and traversing of a BVH by using the Morton codes of primitives. At the current time there exist certain round-off errors. Therefore we have problems to render too small or too big scenes. The algorithm itself and the structuring of the whole ray tracer is still not finished.

See also: #12

Abbildung 1: Die Abbildung zeigt das Format von *Commit*-Nachrichten anhand eines realen Beispiels. Es existiert eine Überschrift im Imperativ und optional eine Erläuterung und Referenzen auf bereits dokumentierte Probleme.

*Commits* in *Git* werden zudem noch auf unterschiedliche Verzweigungen (engl.: *branches*) aufgeteilt. Dies ermöglicht die Bearbeitung des Quelltextes an mehreren Stellen gleichzeitig. Durch das Verschmelzen (engl.: *merge*) zweier Verzweigungen lassen sich Änderungen von einer Verzweigung in eine andere überführen. Eine unsystematische Verwendung von *Branches* führt bei einem *Merge* im Normalfall jedoch zu fehlerbehafteten Artefakten im Quelltext. Aus diesem Grund wurden in den hier gezeigten Projekten *Branches* nur auf spezielle Weise verwendet. Ich richtete mich vor allem nach dem bekannten *git-flow*-Modell<sup>2</sup>, bei welchem die eigentliche Entwicklung auf einer *Develop-Branch* und mehreren *Feature-Branches* vollzogen wird. Bei Veröffentlichungen von neuen Versionen werden *Master-* und *Release-Branches* verwendet. Abbildung 2 zeigt dieses Modell wieder anhand eines realen Beispiels.

## 2.2 CMake

Die Verwendung unterschiedlicher Hardware und Betriebssysteme stellt häufig ein Hindernis bei der Entwicklung robusten Quellcodes dar. Aus diesem Grund verwendete ich innerhalb meiner Projekte CMake. Es ist ein Build-System-Generator. Durch die Ausführung von CMake wird ein entsprechendes Build-System konfiguriert, welches sich dem zugrundeliegenden Rechnersystem anpasst. Das erzeugte Build-System ist in der Lage den Quellcode mit seinen Abhängigkeiten zu kompilieren und auch zu testen. Mittlerweile stellt es eines der Standardwerkzeuge für jeden C- und C++-Programmierer dar.

Um auch hier für andere Entwickler eine feste Struktur beizubehalten, stützte ich mich für das Schreiben von CMake-Dateien auf die zugehörige Dokumentation<sup>3</sup>. Somit konnte ich moderne Standards lernen und anwenden. Die CMake-Dateien beschrieben dadurch immer die benötigten Anforderungen, um den Quelltext fehlerfrei zu kompilieren.

## 2.3 C/C++-Compiler

Natürlich wird für die Kompilierung von Quellcode auch ein entsprechender Compiler benötigt. Das Ziel war es, den geschriebenen Code mit verschiedenen Compilern, wie zum Beispiel *GCC*, *Clang* und dem *Intel C++ Compiler*, übersetzen zu können. Dies sollte sicherstellen, dass innerhalb des Quelltextes keine Compiler-internen Konstrukte verwendet wurden, die auf anderen Systemen vielleicht nicht existierten.

---

<sup>2</sup><http://nvie.com/posts/a-successful-git-branching-model/>

<sup>3</sup><https://cmake.org/documentation>

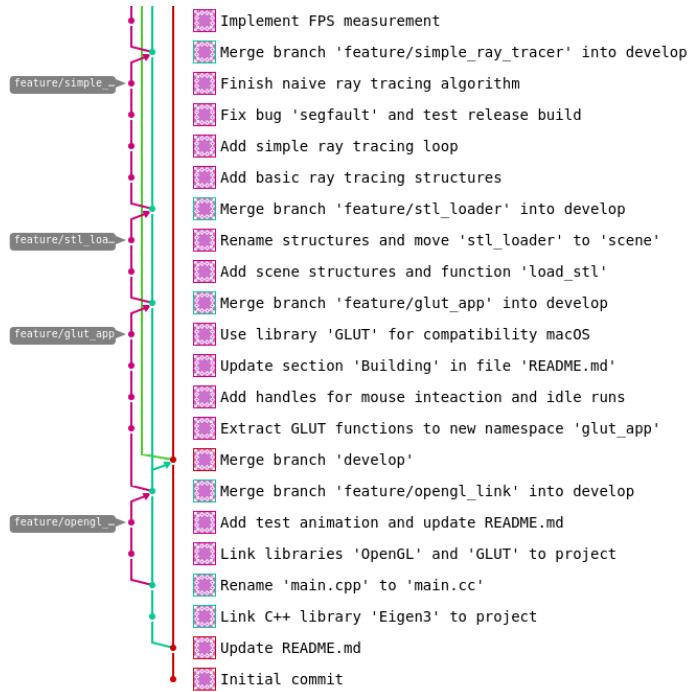


Abbildung 2: Die Abbildung zeigt ein Beispiel für die Anwendung des git-flow-Modells anhand eines durch *GitLab* erstellten Graphen. Es handelt sich um die reale Entwicklungsgeschichte (engl.: *history*) des hier behandelten Projekts. Jeder mit Text beschriftete Eintrag stellt einen protokollierten Zeitpunkt dar. Der Graph auf der linken Seite zeigt den zugehörigen Verzweigungsbaum (engl.: *commit tree*).

## 2.4 Weitere Hilfsmittel

Für das Kreieren von Quellcode ist die Verwendung eines Texteditors oder sogar einer integrierten Entwicklungsumgebung (IDE, engl.: *integrated development environment*) unumgänglich. Die Wahl eines Editors basiert auf den persönlichen Erfahrungen und Vorlieben des Entwicklers. Ich entschied mich für den Editor *Sublime Text*<sup>4</sup>, der als eine Mischung zwischen Texteditor und IDE angesehen werden kann. *Sublime Text* kann unentgeltlich benutzt werden. Er ist zudem Tastatur fixiert, extrem konfigurierbar und durch seine eigene Paketverwaltung beliebig erweiterbar. Zuvor konnte ich *Sublime Text* bereits in anderen Projekten verwenden, wobei dieser einen positiven Eindruck hinterließ. In Abbildung 3 ist ein Screenshot von *Sublime Text* im *Distraction-Free-Mode* gezeigt.

Eines der größten Probleme in der Softwareentwicklung besteht darin, fremden Quelltext zu lesen und zu verstehen. Durch *Style Guides* werden dem Programmierer Orientierungshilfen für die Formatierung des Codes und die Beantwortung von Stilfragen geboten. Die Verwendung von *Style Guides* ermöglicht ein konsistentes Code-Design innerhalb einer Projektgruppe und resultiert außerdem in lesbarem und strukturiertem Quelltext. Ich hielt mich in fast allen Punkten an den *Google C++ Style Guide*<sup>5</sup>. Durch das Programm *ClangFormat*<sup>6</sup> war es mir möglich den Quelltext nicht manuell zu bearbeiten, sondern automatisch durch den Computer formatieren zu lassen. Des Weiteren existierte ein Paket für *Sublime Text*, wel-

<sup>4</sup><https://www.sublimetext.com>

<sup>5</sup><https://google.github.io/styleguide/cppguide.html>

<sup>6</sup><https://clang.llvm.org/docs/ClangFormat.html>

### 3. AUFGABENSTELLUNG

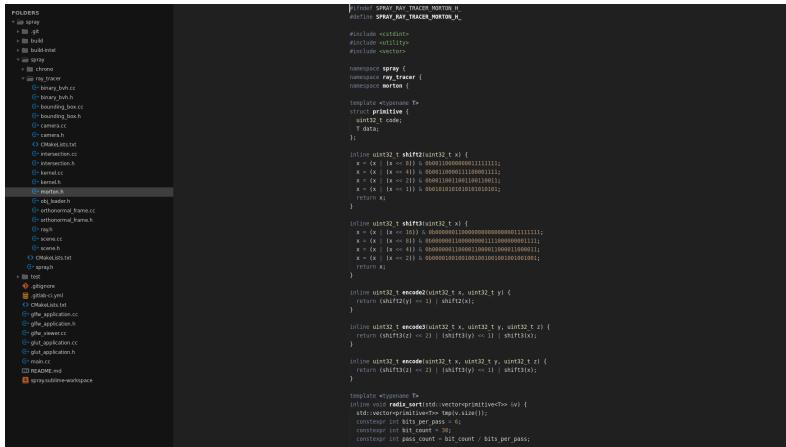


Abbildung 3: Die Abbildung zeigt den Editor *Sublime Text* mit dem hier bearbeiteten Projekt. Die linke Seite zeigt die sogenannte *Sidebar* mit dem Verzeichnisbaum des Projekts.

ches sicher stellte, dass *ClangFormat* nach dem Speichern einer Quelldatei auch auf diese angewendet wurde.

### 3 Aufgabenstellung

Meine Aufgabe bestand hauptsächlich darin, einen echtzeitfähigen Primärstrahlen-Raytracer beziehungsweise Path Tracer auf der CPU zu implementieren. Diese Implementierung sollte als Grundlage für weitere Forschung dienen und musste demnach gut strukturiert werden. Als Eingabedaten wurden triangulierte Szenenmodelle in den Dateiformaten STL und OBJ verwendet, die der Raytracer graphisch darstellen sollte. Des Weiteren wurde mir aufgetragen, mich mit einigen der bereits bekannten Optimierungen eines Raytracers vertraut zu machen und diese ebenfalls zu implementieren.

## 4 Vorbereitung

## 4.1 Einlesen der Szenendaten

**STL-Dateiformat** Das STL-Dateiformat (engl.: *stereolithography*) beschreibt die Oberfläche von 3D-Körpern mit Hilfe von Dreiecksfacetten. Jede Dreiecksfacette wird durch die drei Eckpunkte und die zugehörige Flächennormale des Dreieckes charakterisiert. Bei der hier verwendeten Variante handelt es sich um die binäre Form des Dateiformats, da so eine erhebliche Reduktion der Dateigröße erreicht wird. Zudem übersteigt die Einlesegeschwindigkeit des binären Formats die des Standardformats um mehrere Größenordnungen.

Das Einlesen einer STL-Datei wird im C++-Code durch die Klasse `std::fstream` realisiert. Diese Klasse ist in jedem C++-Compiler vorhanden, wie es im Sprachstandard festgelegt ist. Zunächst wird durch das Auslesen des Dateikopfes die Anzahl der Dreiecke bestimmt. Basierend auf dieser Zahl wird ein Array im Arbeitsspeicher alloziert. Mithilfe einer `for`-Schleife werden dann die Daten jedes einzelnen Dreiecks ausgelesen und im Array gespeichert. Im Folgenden ist dieses Verfahren durch ein Codebeispiel gezeigt.

Code: STL Loader

```

std::vector<Triangle> load(const std::string& file_path) {
    std::fstream file(file_path, std::ios::binary | std::ios::in);
    if (!file.is_open())
        throw std::runtime_error("File could not be opened! Does it
                                exist?");

    file.ignore(80);

    std::uint32_t triangle_count;
    file.read(reinterpret_cast<char*>(&triangle_count), 4);

    std::vector<Triangle> triangles(triangle_count);

    for (auto i = 0; i < triangle_count; ++i) {
        file.read(reinterpret_cast<char*>(&triangles[i].normal), 12);
        file.read(reinterpret_cast<char*>(&triangles[i].vertex), 36);
        file.ignore(2);
    }

    return triangles;
}

```

**OBJ-Dateiformat** OBJ (auch *Wavefront OBJ*) ist ein offenes Dateiformat zum Speichern von dreidimensionalen geometrischen Formen. Das Format wird von vielen Grafikprogrammen unterstützt und ist daher geeignet für die programm- und plattformübergreifende Weitergabe von 3D-Modellen. Es handelt sich um ein ASCII-basiertes Dateiformat, welches zeilenweise ausgelesen werden muss. Jede Zeile enthält einen Befehl mit den entsprechenden Argumenten. Für dieses Projekt wurden nur drei der vielen Befehle verwendet.

Code: OBJ Kommandos

```

# Eckpunkte
v <x> <y> <z>

# Normalen
vn <x> <y> <z>

# Dreiecke mit
f <v id>/<vn id> <v id>/<vn id> <v id>/<vn id>

```

Auch bei diesem Dateiformat wurde die Klasse `std::fstream` verwendet. Für die Eckpunkte, die Normalen und die Flächen wurde jeweils ein eigener Container generiert. Bei dem Container handelt es sich um das Template `std::vector` der C++-Standardbibliothek. Durch die Verwendung dieses Templates wird das Hinzufügen eingelesener Daten auch bei Überschreitung des reservierten Speichers effizient durchgeführt. Alle Koordinaten wurden mit einfacher Genauigkeit in dem Gleitkommatyp `float` und alle Referenzen auf Eckpunkte oder Normalen im Ganzzahltyp `int` gespeichert.

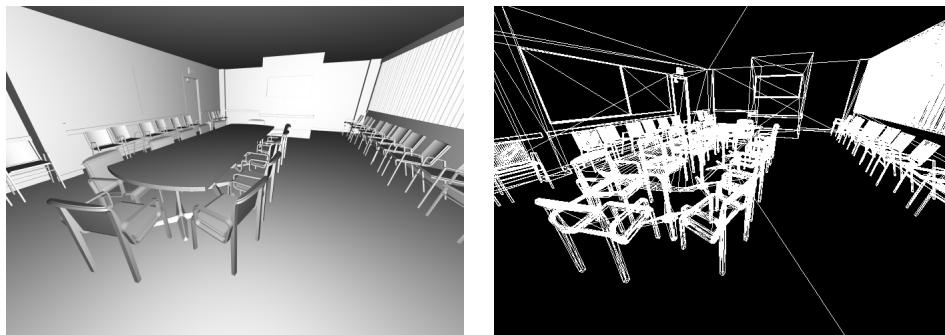


Abbildung 4: Die Abbildung zeigt die Szene *Conference Room*. Auf der linken Seite wurde das Bild mithilfe des Raytracers generiert. Die rechte Seite zeigt das *Wireframe* gerendert durch OpenGL.

## 4.2 Grafische Ausgabe

Für die graphische Ausgabe wurde die Bibliothek OpenGL zusammen mit dem Framework GLUT verwendet. GLUT erstellt einen OpenGL-Context und dient auch als Window-Manager. OpenGL wird benötigt um den Pixelpuffer eines Fensters (Array, indem der Status jedes Pixels eines Fensters gespeichert ist) auf dem Bildschirm auszugeben.

Um sowohl das Einlesen der Dateiformate als auch die Korrektheit des Raytracers zu testen, wurde die Graphics-Engine von OpenGL genutzt. Anhand einiger Standardmodelle, wie dem *Conference Room* in Abbildung 4, konnte die korrekte Funktion der Datei-Loader und der graphischen Ausgabe sichergestellt werden.

## 4.3 Performance-Analyse

Damit die Effizienz der Algorithmen objektiv miteinander verglichen werden konnte, wurde die mittlere Zeitspanne zum Erstellen eines Bildes (engl.: *Frame*) für jeden Algorithmus gemessen. Das Reziproke dieser Messgröße wird in der Literatur auch als *frames per second* (FPS) bezeichnet. Hierfür wurde innerhalb einer festgelegten Zeitspanne (in der Regel 5 Sekunden) gemessen, wie oft der Pixelpuffer neu berechnet und ausgegeben wurde. Der Quotient aus der gesamten Zeitspanne und der Anzahl der erzeugten Frames ist dann die mittlere Zeit, die zur Berechnung des Pixelpuffers benötigt wird.

Die hohe Komplexität der Szenen resultiert in starken Schwankungen der FPS je nach Position und Richtung der Kamera. Die verschiedenen Algorithmen arbeiten in unterschiedlichen Raumbereichen mit unterschiedlicher Effizienz. Für eine quantitative Analyse dieser Algorithmen wurden aus diesem Grund vorher festgelegte Kamerapfade verwendet. Der Beobachter bewegt sich auf diesen Pfaden und misst für jeden gegebenen Punkt die FPS. So können die Stärken und Schwächen der einzelnen Algorithmen ermittelt werden.

Zum Festlegen der Kamerapfade muss eine Szene zunächst von einem Benutzer analysiert werden, um Bereiche mit hohen FPS-Schwankungen zu finden. Hierfür wurde die manuelle Steuerung der Kamera durch den Benutzer ermöglicht. Zudem existieren auch hochkomplexe Szenen, die nur durch die manuelle Analyse untersucht werden können.

## 5 Grundlagen und der naive Algorithmus

Abbildung 5 stellt eine Skizze des grundsätzlichen Algorithmus dar. Für jeden Pixel des Pixelbuffers wird ein Strahl von der Kameraposition durch den Pixel in die Richtung des Bildschirms ausgesendet. Jeder dieser Strahlen wird dann mit der Szene auf Schnittpunkte getestet. Existiert ein Schnittpunkt, so wird dieser durch weitere Algorithmen dargestellt. Existieren mehrere Schnittpunkte, so wird derjenige ausgewählt, der der Kamera am nächsten ist.

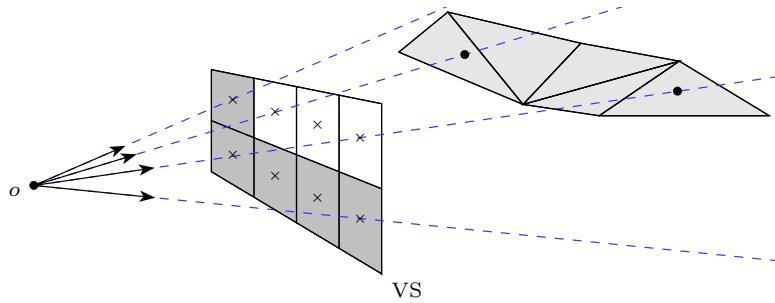


Abbildung 5: Die Skizze stellt das Raytracing-Verfahren dar. Bezüglich eines Beobachtungspunktes  $o$  wird durch jeden Pixel eines virtuellen Bildschirms VS ein Strahl geschossen. Die grau melierten Dreiecke stellen dabei die Szene dar. Jeder Pixel des virtuellen Bildschirms entspricht einem analogen Pixel des realen Bildschirms.

### 5.1 Aufbau der Kamera

Für die Implementierung wurde das Einlesen der Szene als Liste von Dreiecken wie in Abschnitt 4.1 realisiert. Die Kamera konnte mithilfe einer Klasse dargestellt werden. In der folgenden Abbildung 6 ist die Bedeutung der einzelnen Parameter skizziert. Die Position und

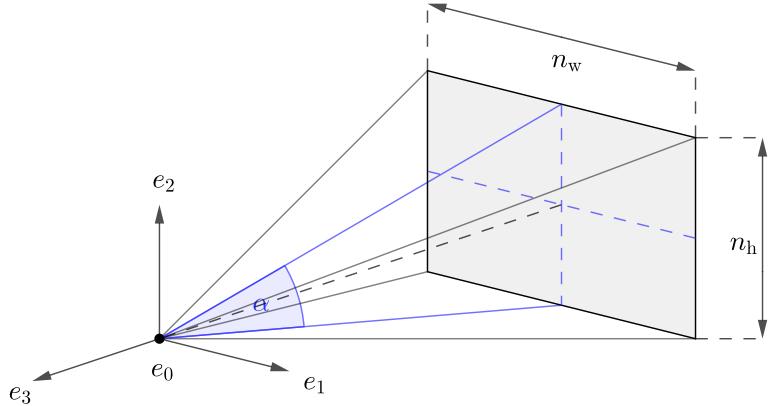


Abbildung 6: Die Abbildung veranschaulicht die Parameter der Klasse Camera anhand einer Skizze. Das Tupel  $(e_0, \{e_1, e_2, e_3\})$  beschreibt die affine Basis,  $n_w$  und  $n_h$  die Breite und die Höhe des virtuellen Bildschirms in Pixeln und  $\alpha$  den Öffnungswinkel (engl.: field of view).

Ausrichtung der Kamera wurde durch eine positiv orientierte affine Basis im  $\mathbb{R}^3$  beschrieben. Der erste Basisvektor zeigt in die Richtung, die für die Kamera als rechts definiert ist. Analog-

ges gilt für den zweiten und dritten Basisvektor bezüglich den Richtungen oben und hinten. Der zu einer Kamera gehörige virtuelle Bildschirm wurde durch die Anzahl seiner Pixel  $n_w$  und  $n_h$  in der Breite und Höhe charakterisiert. Die Entfernung zur Kamera wurde durch den Öffnungswinkel  $\alpha$  (engl.: *field of view*), unter dem die Höhe des Bildschirms von der Kamera aus zu sehen ist, angegeben. Weitere Größen wie die Kantenlänge  $p$  eines Pixels im Raum oder das Seitenverhältnis  $r$  des virtuellen Bildschirms lassen sich aus den bereits definierten Größen eindeutig berechnen.

$$p = \frac{2 \tan \frac{\alpha}{2}}{n_h} \quad r = \frac{n_w}{n_h}$$

## 5.2 Strahl-Dreieck-Schnittpunkttest

Die von der Kamera ausgesendeten Primärstrahlen mussten auf Schnittpunkte mit der Szene getestet werden. Für jeden Strahl wurden hierfür mithilfe des Möller-Trumbore-Verfahrens die Schnittpunkte mit allen Dreiecken der Szene berechnet. Um das genannte Verfahren genauer zu erklären, seien  $A, B, C \in \mathbb{R}^3$  die Eckpunkte eines Dreiecks, wobei gilt, dass die Menge  $\{B - A, C - A\}$  linear unabhängig ist. Wie sich leicht überprüfen lässt, lassen sich alle Punkte des Dreiecks durch die folgende Parametrisierung beschreiben.

$$\begin{aligned} M &:= \{(u, v) \in [0, 1]^2 \mid u + v \leq 1\} \\ \varphi: M &\rightarrow \mathbb{R}^3, \quad \varphi(u, v) := (1 - u - v)A + uB + vC \end{aligned}$$

Die Koordinaten  $u, v$  und  $1 - u - v$  werden auch als die baryzentrischen Koordinaten eines Dreiecks bezeichnet. Weiterhin nehmen wir an, dass ein Strahl durch die folgende Funktion  $r$ , den Ursprung  $o \in \mathbb{R}^3$  und die Richtung  $d \in \mathbb{R}^3 \setminus \{0\}$  beschrieben wird.

$$r: [0, \infty) \rightarrow \mathbb{R}^3, \quad r(t) := o + td$$

Für einen Schnittpunkt zwischen Strahl und Dreieck muss die folgende Gleichung für  $(u, v) \in M$  und  $t \in [0, \infty)$  erfüllt sein.

$$r(t) = \varphi(u, v)$$

Zur Lösung dieser Gleichung setzt man zunächst die Parametrisierungen von Strahl und Dreieck ein.

$$o + td = A + (B - A)u + (C - A)v$$

Durch Umbenennung der konstanten Vektoren lässt sich diese Gleichung in einer übersichtlicheren Form darstellen. Abbildung 7 skizziert die Bedeutung aller bisher verwendeten Variablen.

$$e_0 = ue_1 + ve_2 - td$$

$$e_0 := o - A \quad e_1 := B - A \quad e_2 := C - A$$

Fasst man nun die Koordinaten  $u, v$  und  $t$  zu einem Vektor zusammen, so lässt sich die Gleichung sehr elegant durch ein Matrix-Vektor-Produkt formulieren. Hierbei gehen wir

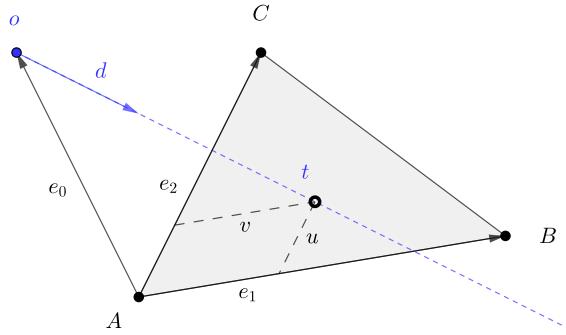


Abbildung 7: Die Abbildung zeigt die Bedeutung der verwendeten Variablen der Schnittpunktberechnung zwischen Strahl und Dreieck anhand einer Skizze.

davon aus, dass es sich bei den Vektoren  $e_0, e_1, e_2$  und  $d$  um Spaltenvektoren handelt.

$$e_0 = \begin{pmatrix} e_1 & e_2 & -d \end{pmatrix} \begin{pmatrix} u \\ v \\ t \end{pmatrix}$$

Der Schnittpunkt zwischen einem Strahl und einem Dreieck kann demnach aus der Lösung eines linearen Gleichungssystems, charakterisiert durch eine  $3 \times 3$ -Matrix, gewonnen werden. Offensichtlich wird die Determinante der angegebenen Matrix genau dann Null, wenn der Strahl parallel zur Ebene des Dreiecks verläuft. In diesem Falle existieren entweder keine oder unendlich viele Schnittpunkte, sodass keine Aussagen über  $u, v$  und  $t$  getroffen werden können. Nimmt man nun an, dass die Matrix nicht singulär ist, so lässt sich die Gleichung durch eine Invertierung lösen.

$$\begin{pmatrix} u \\ v \\ t \end{pmatrix} = \begin{pmatrix} e_1 & e_2 & -d \end{pmatrix}^{-1} e_0$$

Die Inverse der Matrix lässt sich analytisch leicht durch Skalar- und Kreuzprodukte beschreiben. Innerhalb des Computers ermöglicht diese Formulierung zudem eine effizientere Auswertung der Schnittpunkte.

$$\begin{pmatrix} u \\ v \\ t \end{pmatrix} = \frac{1}{\langle e_1, d \times e_2 \rangle} \begin{pmatrix} \langle d \times e_2, e_0 \rangle \\ \langle e_1 \times d, e_0 \rangle \\ \langle e_1 \times e_2, e_0 \rangle \end{pmatrix}$$

In einem letzten Schritt werden die Vektoren innerhalb der Skalar- und Kreuzprodukte zyklisch vertauscht, um die Lösung der Gleichung nur durch die Berechnung von zwei verschiedenen Kreuzprodukten zu erhalten.

$$\begin{pmatrix} u \\ v \\ t \end{pmatrix} = \frac{1}{\langle e_1, d \times e_2 \rangle} \begin{pmatrix} \langle d \times e_2, e_0 \rangle \\ \langle e_0 \times e_1, d \rangle \\ \langle e_0 \times e_1, e_2 \rangle \end{pmatrix}$$

Durch diese Gleichung lassen sich nun die Parameter  $u, v$  und  $t$  eindeutig bestimmen. Abschließend muss überprüft werden, ob diese auch die Bedingungen der Parametrisierungen

$(u, v) \in M$  und  $t \in [0, \infty)$  erfüllen. Die folgenden Ungleichungen stellen diese in vereinfachter Form dar.

$$\langle e_1, d \times e_2 \rangle \neq 0 \quad u, v, t \geq 0 \quad u + v \leq 1$$

Sind die Bedingungen erfüllt, so liegt ein Schnittpunkt vor. Eine beispielhafte Implementierung ist in dem folgenden Quelltext zu sehen.

Code: Möller-Trumbore-Algorithmus

```

bool intersect(const Ray& ray, const Triangle& triangle, Vector3f
    & uvt) {
    const Vector3f edge_1 = triangle.vertex[1] - triangle.vertex
        [0];
    const Vector3f edge_2 = triangle.vertex[2] - triangle.vertex
        [0];
    const Vector3f rhs = ray.origin() - triangle.vertex[0];

    const Vector3f cross_1 = ray.direction().cross(edge_2);
    const Vector3f cross_2 = rhs.cross(edge_1);

    const float determinant = edge_1.dot(cross_1);
    const float inverse_determinant = 1.0f / determinant;

    uvt[0] = rhs.dot(cross_1);
    uvt[1] = ray.direction().dot(cross_2);
    uvt[2] = edge_2.dot(cross_2);
    uvt *= inverse_determinant;

    return (0.0f != determinant) && (uvt[0] >= 0.0f) && (uvt[1] >=
        0.0f) &&
        (uvt[2] > 0.0f) && (uvt[0] + uvt[1] <= 1.0f);
}

```

### 5.3 Raytracing und Rendering

Durch eine Iteration über alle Dreiecke der Szene lassen sich für einen Strahl alle Schnittpunkte mit der Szene berechnen. Für das eigentliche Raytracing reicht es jedoch, den Schnittpunkt zu finden, der dem Ursprung des Strahls am nächsten ist. Aus diesem Grund wird bei der Iteration über alle Dreiecke mithilfe des Strahlparameters  $t$  getestet, welcher Schnittpunkt weiter entfernt ist. Der folgende Quelltext implementiert dieses grundlegende Verfahren.

Code: Naives Raytracing

```

Intersection trace(const Ray& ray, const Scene& scene) {
    // construct non-valid default intersection
    Intersection intersection;

    // compute intersections for every triangle
    for (const auto& triangle : scene) {

```

```

Vector3f uvt;
if (!intersect(ray, triangle, uvt)) continue;

// use closer intersection
if (uvt[2] < intersection.t()) {
    intersection.uvt() = uvt;
    intersection.triangle() = &triangle;
}
}

return intersection;
}

```

Um nun auch für den Benutzer ein Bild zu generieren, müssen durch Verwendung der bereits implementierten Kamera sogenannte Primärstrahlen für jeden Pixel des virtuellen Bildschirms erzeugt werden. Diese haben ihren Ursprung an der Position der Kamera. Die Richtungen können leicht durch die Verwendung der affinen Basis und der Pixelgröße aus Abschnitt 5.1 berechnet werden. Mit der zuvor beschriebenen Routine `trace` werden nun die Schnittpunkte, sofern sie vorhanden sind, bestimmt. Die erhaltenen Informationen werden dann in den Pixelpuffer hineingeschrieben und durch *OpenGL* auf dem Bildschirm ausgegeben. Auch für dieses Verfahren sei als Beispiel der folgende Code gegeben.

Code: Rendering

```

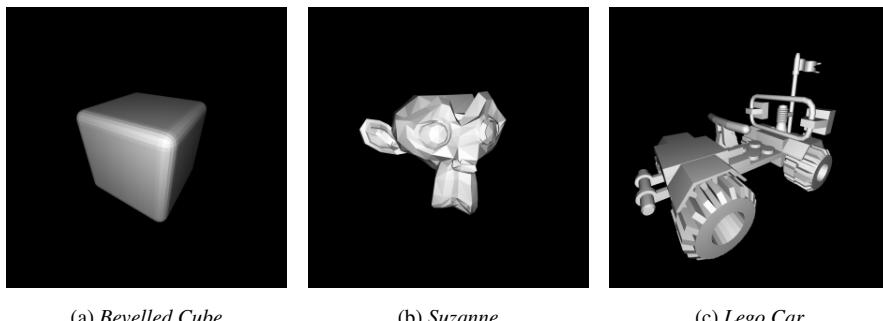
void render(const Scene& scene, const Camera& camera,
           Pixel_buffer& pixel_buffer) {
    // make sure pixel buffer has allocated memory needed
    pixel_buffer.resize(camera.screen_width() * camera.
                        screen_height());

    // trace a ray for every pixel
    for (auto i = 0; i < camera.screen_height(); ++i) {
        for (auto j = ; j < camera.screen_width(); ++j) {
            const Ray ray = primary_ray(camera, j, i);
            const auto intersection = trace(ray, scene);
            if (intersection) {
                // if there is an intersection use a dot product
                // to better visualize the part of the triangle
                float dot = -ray.direction().dot(intersection.triangle()
                    ->normal());
                if (dot < 0.0f) dot = 0.0f;
                pixel_buffer(j, i) = Vector3f(dot, dot, dot);
            } else {
                // if there is no intersection use color black
                pixel_buffer(j, i) = Vector3f(0, 0, 0);
            }
        }
    }
}

```

### 5.4 Ergebnisse

Die Ergebnisse der implementierten Methoden waren erfolgreich. In Abbildung 8 sind drei der hier verwendeten Modelle gezeigt. Jedes dieser Modelle besitzt im Vergleich zu anderen Szenen eine geringe Anzahl von Dreiecken. Trotz allem dauerte die Generierung der Bilder mehrere Sekunden bis hin zu einigen Minuten, je nachdem welches Rechnersystem verwendet wurde. Die Ergebnisse des naiven Algorithmus entsprachen demnach den Erwartungen. Es ging vor allem um die grundlegenden Strukturen des Projekts, die robust und verständlich implementiert werden sollten.



(a) Bevelled Cube

(b) Suzanne

(c) Lego Car

Abbildung 8: Die Abbildungen wurden mithilfe des hier entwickelten naiven Raytracers generiert. Zu beachten ist, dass jedes der verwendeten Modelle durch vergleichsweise wenig Dreiecke beschrieben wird.

## 6 Beschleunigungsstrukturen, Optimierung und weitere Features

Auf den in Abschnitt 5 beschriebenen Grundlagen bauten alle weiteren Optimierungen und Features auf. Ich habe mich mit diesem Projekt innerhalb meiner Arbeitsgruppe seit über vier Jahren beschäftigt und folglich einige Verbesserungen vorgenommen, sowie Messungen durchgeführt und Erfahrungen gesammelt. Die genaue Beschreibung der Lösungsmethoden und Ergebnisse aller Veränderungen würde über den Rahmen dieses Berichts hinausgehen. Demzufolge sind im Folgenden viele der Features und Optimierungen in einer kürzeren Form dargestellt. So ist es mir möglich, eine verständliche Zusammenfassung der Arbeit am Projekt zu vermitteln.

### 6.1 Linear Bounding Volume Hierarchies

Beschleunigungsstrukturen stellen eines der Kernelemente eines jeden Raytracers dar und funktionieren unabhängig von der Hardware. Ohne Algorithmen, die die Anzahl der Schnittpunkttests für jeden Strahl reduzieren, wäre die Zeit einen einzelnen Strahl durch die Szene zu schießen proportional zur Anzahl der Dreiecke. In den meisten Fällen handelt es sich hierbei jedoch um Zeitverschwendungen, da ein Strahl die überwiegende Mehrheit der Dreiecke gar nicht schneidet. Das Ziel von Beschleunigungsstrukturen besteht darin, Gruppen von nicht-schneidenden Dreiecken schnell zu verwerfen und die Übrigen zu sortieren, um nahegelegene Dreiecke zu bevorzugen.

*Bounding Volume Hierarchies* (BVH) sind Beschleunigungsstrukturen. Sie unterteilen eine gegebene Szene in eine Hierarchie von disjunkten Mengen von Dreiecken. Abbildung 9 zeigt diese Methode anhand zweier Skizzen. Dreiecke werden in den Blättern der Hierarchie

gespeichert, während sogenannte *Bounding Boxes* die Knoten zwischen den Blättern füllen. Ein Strahl traversiert den Baum, indem er in jedem Knoten, an welchem er vorbeikommt, einen Schnittpunkttest mit der *Bounding Box* durchführt. Fällt der Test negativ aus, so wird der Knoten mit seinen Kindern verworfen.

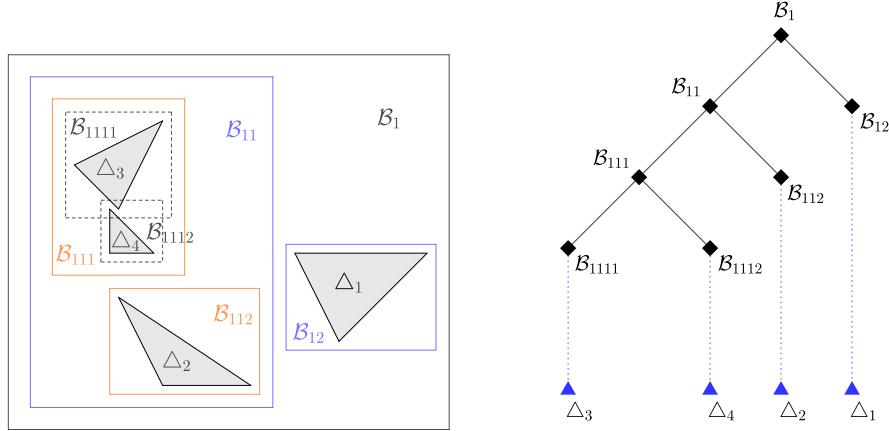


Abbildung 9: Die Abbildungen zeigen die skizzenhafte Darstellung einer möglichen BVH einer Szene bestehend aus den Dreiecken  $\Delta_1, \Delta_2, \Delta_3$  und  $\Delta_4$ . Im linken Bild sind die *Bounding Boxes* der BVH und im rechten Bild deren zugehörige Verknüpfungen im Sinne eines Baums gezeigt.

Um die Schnittpunktberechnung zwischen Strahl und *Bounding Box* so weit wie möglich zu vereinfachen, werden an den Koordinatenachsen ausgerichtete *Bounding Boxes* (AABB, engl.: *axis-aligned bounding box*) verwendet. Diese lassen sich eindeutig durch die Angabe zweier ihrer gegenüberliegenden Eckpunkte beschreiben. Eine AABB beschreibt damit einen Quader im Raum. Aus diesem Grund muss für jede der sechs Flächen ein Schnittpunkttest durchgeführt werden. Durch die Ausrichtung an den Koordinatenachsen vereinfachen sich diese sechs Tests jedoch zu einer Division, sechs Additionen und sechs Multiplikationen.

Um BVHs zu konstruieren lassen sich diverse Algorithmen verwenden. Im Projekt wählte ich eine Methode, die sowohl auf der CPU als auch GPU effizient funktionieren sollte. *Linear Bounding Volume Hierarchies* (LBVH) können in linearer Zeit bezüglich der Anzahl der Dreiecke konstruiert werden. Sie sind vergleichsweise einfach parallelisierbar und skalieren sowohl auf der CPU als auch auf der GPU.

Die Idee besteht darin, die Konstruktion einer LBVH auf ein Sortierverfahren zurückzuführen. Da es im dreidimensionalem Raum keine eindeutig ausgezeichnete Ordnung gibt, verwendet man *Morton Codes*. Diese bilden naheliegende dreidimensionale Punkte auf nahe liegende eindimensionale Punkte entlang einer Kurve ab. Aufgrund ihrer charakteristischen Form wird diese Kurve auch die Z-Kurve genannt. Abbildung 10 zeigt den Verlauf der Morton-Kurve für zwei unterschiedlich feine Gitter. Zudem zeigt sie, dass *Morton Codes* auf einer simplen Transformation basieren. Gegeben sei für ein  $n \in \mathbb{N}$  ein Tupel natürlicher Zahlen  $k \in \mathbb{N}_0^n$ . Die Binärdarstellung einer Koordinate  $k_i$  für  $i \in \mathbb{N}_0$  mit  $i < n$  sei gegeben durch

$$k_i = (\dots k_{i3} k_{i2} k_{i1} k_{i0})_2$$

In diesem Falle berechnet sich der *Morton Code*  $m(k)$  von  $k$  durch den folgenden Ausdruck.

$$m(k) := (\dots k_{22} k_{12} k_{02} \dots k_{21} k_{11} k_{01} \dots k_{20} k_{10} k_{00})_2$$

Durch die Verwendung von *Radix Sort* und der *Morton Codes* der Dreiecksmittelpunkte lässt sich nun durch ein *Bottom-Up*-Verfahren die BVH effizient und parallel konstruieren.

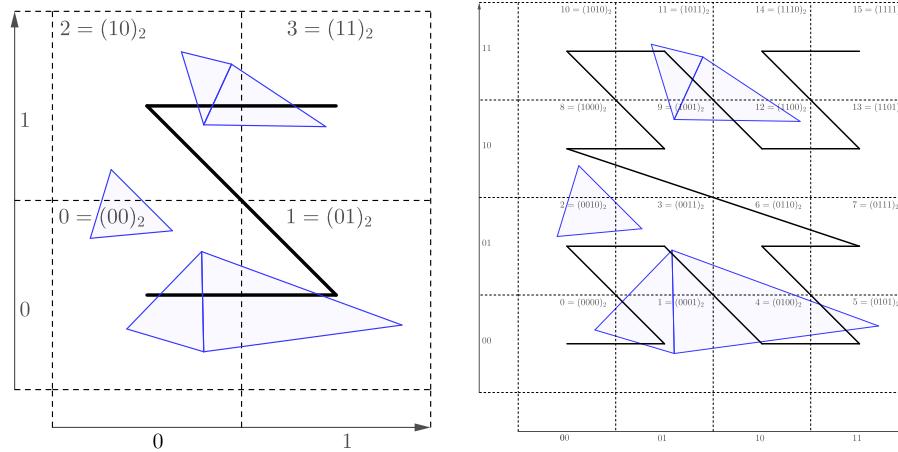


Abbildung 10: Die Abbildung zeigt den Verlauf der Morton-Kurve durch die Szene für zwei unterschiedliche Verfeinerungsgrade der Gitter.

Die Struktur einer BVH führt dazu, dass die Laufzeitkomplexität im Durchschnitt logarithmisch statt linear in der Anzahl der Dreiecke verläuft. Dies führte zu einer dramatischen Verbesserung der vorherigen Ergebnisse. Abbildung 11 zeigt vier der im Projekt verwendeten hochkomplexen Modelle. Jede Szene konnte durch User-Interaktion mit durchschnittlich 50 FPS rotiert werden. Damit übertraf die Verwendung der LBVH jede meiner Erwartungen. Auch die Konstruktion der LBVH war innerhalb von ein paar Millisekunden abgeschlossen. Doch auch dieser extrem performante Build-Prozess offenbarte nach genaueren Messungen einige Schwachstellen. Die entstehende BVH stellt keine optimale BVH der Szene dar. Sobald sich die Kamera innerhalb eines Objektes befand, verlangsamte sich das gesamte Rendering. Im Verlauf des Projektes wurden innerhalb meiner Arbeitsgruppe weitere Verfahren zur Konstruktion von BVHs eingebaut. Es scheint als würden die besten Bäume durch hybride Verwendung von mehreren Methoden entstehen.

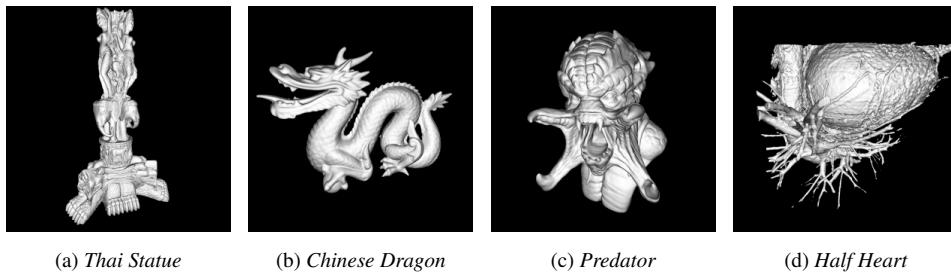


Abbildung 11: Die Abbildungen zeigen vier Modelle, welche durch den Raytracer gerendert wurden. Aufgrund der extrem hohen Dreiecksanzahl war es erst durch die Verwendung einer Beschleunigungsstruktur, wie der LBVH, möglich, diese Bilder in Echtzeit zu generieren.

## 6.2 Path Tracing

Da der Raytracer vor allem der Visualisierung dienen sollte, war es nötig, den Lichttransport innerhalb einer Szene zu simulieren. In der Theorie wird die Simulation von Lichtverteilungen durch die Rendergleichung (engl.: *rendering equation* oder *light transport equation*) beschrieben. Dabei handelt es sich um eine Fredholm-Integralgleichung 2. Art für die Strahldichte der gesamten Szene. Sie kann mithilfe der Energieerhaltung und Theoremen der geometrischen Optik und Radiometrie hergeleitet werden. Im Allgemeinen besitzt diese Gleichung keine geschlossene Lösung. Ihre Verwendung bedingt zudem die Implementierung von physikalischen Materialien, die Licht reflektieren, absorbieren und brechen. Die Materialien beschreiben dabei die genaue Interaktion von Lichtstrahlen mit den Oberflächen von Objekten durch bidirektionale Streuungsverteilungsfunktionen (BSDF, engl.: *bidirectional scattering distribution function*).

Eine BSDF gibt an, welcher Anteil des einfallenden Lichtes bezüglich der Einfallsrichtung in eine bestimmte Richtung gestreut wird. Sie stellt damit eine Verallgemeinerung der idealen Reflexion und Brechung an Oberflächen dar. Die Konstruktion komplexer Materialien erfolgte zumeist durch Bildung einer Linearkombination bereits bekannter und gut beschreibbarer BSDFs, wie der Lambertsche diffuse Strahler oder die *Glossy Reflection*, sodass diese effizient durch die Speicherung von Koeffizienten implementiert werden konnten.

Für die numerische Lösung der Rendergleichung transformiert man diese zunächst in eine äquivalente Formulierung basierend auf Pfadintegralen. Die Integrale lassen sich nun durch Monte-Carlo Methoden erwartungstreu schätzen. Trifft ein Primärstrahl auf die Oberfläche, so wird durch die randomisierte *Russian Roulette*-Methode entschieden, ob dieser Strahl reflektiert, absorbiert oder transmittiert wird. Auf der Basis dieser Entscheidung wird eine neuer zufälliger Strahl auf der entsprechenden Hemisphäre generiert, der wiederum den kompletten *Path Tracing*-Prozess durchläuft. Um die Farbe eines Pixels mit einem geringen Fehler zu schätzen, ist es notwendig eine große Anzahl dieser Pfade auszuwerten.

Abbildung 12 zeigt diverse durch *Path Tracing* generierte Bilder. Je nach Beleuchtung und Aufbau einer Szene konvergiert der Algorithmus unterschiedlich schnell. Vor allem im Inneren von Objekten muss eine enorm lange Zeit vergehen, bis ein akzeptables Ergebnis zu sehen ist. Es konnte leider nicht gemessen werden, ob die simulierten Ergebnisse auch der Wirklichkeit entsprachen. Dennoch erschienen die Bilder für das menschliche Auge durchaus realistisch und nicht Computer generiert.

## 6.3 Optimierungen auf der CPU

Neben Features und Beschleunigungsstrukturen gab es auch Hardware-basierte Verfahren zur Optimierung des Quellcodes. Innerhalb der Arbeitsgruppe habe ich mich mit *Cache Coherency* und der Parallelisierung auseinandergesetzt. Grundsätzlich gibt es mehrere Formen der Parallelisierung. Für das Projekt beschäftigte ich mich vor allem mit SIMD (engl.: *single input multiple data*) und MIMD (engl.: *multiple input multiple data*) beziehungsweise *Thread-Level-Parallelism*. Begründet war dies durch die Tatsache dass der Compiler bereits automatisch einen großen Teil des Quelltexts effizient optimierte.

Durch die Funktionsweise des Rendering-Verfahrens stellte die Implementierung von Thread-Parallelismus kein Problem dar. Für jeden Pixel des Pixelpuffers mussten mehrere hundert Strahlen durch *Path Tracing* unabhängig voneinander ausgewertet werden. Gerade die Unabhängigkeit der verschiedenen Pixel ermöglichte es mir, den Pixelpuffer in mehrere

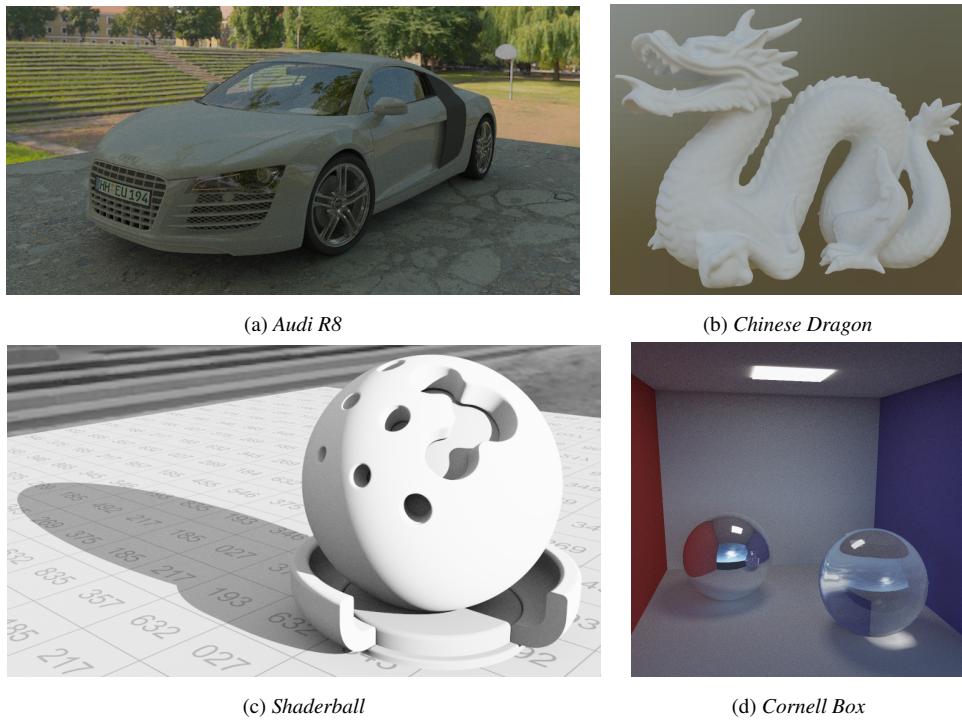


Abbildung 12: Die Abbildungen wurden mithilfe des hier implementierten *Path Tracing*-Verfahrens für verschiedene Modelle erzeugt. Die Materialien der Modelle wurden durch ideale Reflexionen, Brechungen und Lambertsche Strahler beschrieben. Für jeden Pixel wurden 1024 zufällige Pfade ausgewertet.

Pixelblöcke einzuteilen und diese dann den Threads des Computers zuzuweisen. Da unterschiedliche Pixelblöcke je nach Szene und Beleuchtung auch unterschiedlich lange berechnet wurden, verwendete ich eine dynamische Zuweisung der Pixelblöcke zu den Threads (siehe *load balancing*). Dieses Verfahren wurde sowohl durch *OpenMP* als auch durch eine vom ITWM entwickelten Thread-Bibliothek *MCTP* umgesetzt.

Die Ergebnisse dieser Implementierung entsprachen voll und ganz den Erwartungen. Die Geschwindigkeit der Bildgenerierung konnte um die Anzahl der Cores der CPU eines Computers vervielfacht werden.

Die manuelle Parallelisierung durch SIMD ist prinzipiell nur durch spezielle Bibliotheken, wie *OpenMP*, oder durch das Einbinden der sogenannten *Intrinsics*, wie den *Streaming SIMD Extensions* (SSE) oder den *Advanced Vector Extensions* (AVX), möglich. Sie erlauben es dem Entwickler im Quelltext auf die Vektorregister der CPU zuzugreifen. Je nach Baujahr eines Computers passen in diese Register vier bis sechzehn Gleitkommazahlen einfacher Genauigkeit hinein. Demnach bedeutet eine effiziente Nutzung dieser Register eine vier- bis sechzehnfache Steigerung der Performance. Die manuelle Verwendung dieser *Intrinsics* geht jedoch mit einer starken Hardwarebindung einher. Code der durch AVX optimiert wurde, lässt sich nur auf Computern, die ebenfalls AVX kennen, ausführen. Aus diesem Grund werden häufig mehrere Varianten eines Algorithmus implementiert, um eine maximale Portabilität zu erreichen.

Im Projekt konzentrierte ich mich auf den Kernel des Raytracers. Jeder Strahl musste die BVH traversieren um einen Schnittpunkt zu ermitteln. Anstatt innerhalb eines Threads

die Strahlen sequentiell zu testen, bin ich dazu übergegangen, mehrere Strahlen in einem Strahlenpaket durch mehrere Vektorregister zu speichern. Diese Neustrukturierung geht davon aus, dass alle Strahlen, die in diesem Paket vorkommen, kohärent sind, beziehungsweise einen ähnlichen Strahlenverlauf aufweisen. Diese Annahme ist zumindest für Primärstrahlen, die von der Kamera ausgesendet werden, bis zu einem gewissen Grade erfüllt. Bei der eigentlichen Traversierung wurde das Strahlenpaket wie ein einziger Strahl behandelt und auf Schnittpunkte mit den einzelnen AABB der Knoten getestet. Ein Knoten und dessen Kinder wurden jedoch nur verworfen, falls keiner der Strahlen im Paket die zugehörige AABB schneidet. Waren die Strahlen, wie zuvor angenommen, kohärent, so ließen sich die Schnittpunkte der Strahlen eines Pakets genauso so schnell wie der Schnittpunkt eines einzelnen Strahls berechnen. Wiesen die Strahlen allerdings eine starke Divergenz auf, so reduzierte sich die Effizienz des Verfahrens erheblich, da mehr Äste der BVH traversiert werden mussten.

Die Ergebnisse dieser Implementierung zeigten eine drei- bis fünfzehnfache Steigerung der Performance für die Berechnung von Primärstrahlen. Sekundärstrahlen, welche zum Beispiel durch die Reflexion an Oberflächen entstanden, ließen die *Frame Rate* einbrechen. In extremen Fällen konnte sogar festgestellt werden, dass die automatische Optimierung des Compilers bessere Ergebnisse lieferte.

## 7 Selbsteinschätzung und Bezug zum Studium

Im Rahmen des Projektes konnte ich tiefgreifende Kenntnisse in der Anwendung von den in Abschnitt 2 beschriebenen Hilfsmitteln erwerben. Diese habe ich mir selbst angeeignet und zudem in der Praxis getestet. Dadurch konnte ich das erlangte Wissen nicht nur auf einzelne Projekte sondern auch auf die vielfältigen Kontexte des Studiums anwenden. Als Beispiel seien hier diverse Computational Science Praktika genannt, bei denen ich durch die Anwendung von *Git*, *CMake* und *ClangFormat* eine wesentlich effizientere Arbeitsweise, sowie auch besser strukturierten beziehungsweise lesbaren Quelltext, entwickelte. Des Weiteren lernte ich, Entwicklungsumgebungen auf verschiedenen Computersystemen einzurichten und zu warten. Beim Einrichten dieser achtete ich darauf, dass die Verwendung des aktuellen Standards der verschiedenen Tools möglich war. Folglich besitze ich einen guten Überblick über die Standards von C++, *Git* und *CMake*. Diese Kompetenzen konnte ich im Studium vor allem dafür verwenden, anderen Studenten zu helfen und mich in Gruppen besser zu etablieren. Gerade durch die Zusammenarbeit innerhalb einer Gruppe erkannte ich, wie wichtig ein sehr gut lesbarer, strukturierter und konsistenter Programmierstil ist. Während des Projektes konnte ich mir diesen aneignen und in darauffolgenden Projekten verbessern. Somit diente der entstehende Code zu jeder Zeit als solides Fundament für die Entwicklung des Projekts. Erst dadurch war es möglich die Problemstellungen kreativ und effizient zu lösen und flexibel auf die Anforderungen anderer Teammitglieder einzugehen, ohne sich in der Unordnung des Quelltextes zu verlieren.

Dennoch habe ich während des Projekts festgestellt, dass ich mich noch mit einigen Aspekten meiner Arbeitsweise auseinandersetzen muss. In einigen Phasen des Projekts neigte ich dazu, von der wesentlichen Aufgabenstellung abzuweichen und mich nur um kleinere Details zu kümmern. Diese Details waren nicht essentiell für den Fortschritt der Arbeit und raubten mir einen beträchtlichen Anteil meiner Zeit. Es gab auch vereinzelte Codeausschnitte, bei denen ich im Nachhinein feststellte, dass ich sie nicht gebraucht hätte. Dementsprechend

werde ich in Zukunft verstkt darauf achten, meine Zeiteinteilung zu optimieren und mich auf das Wesentliche zu konzentrieren.

Innerhalb des Projektes gab es diverse Mglichkeiten, die aus dem Studium bekannte Mathematik anzuwenden. Es waren bereits grundlegende Kenntnisse der linearen Algebra ntig, um den naiven Algorithmus, der fr alle weiteren Optimierungen als Ausgangspunkt diente, zu konstruieren. Fr die grafische Ausgabe und die Schnittpunktberechnungen waren die Matrix- und Vektorrechnung unerlsslich. Zudem war es im Computer nicht mglich, echte reelle Zahlen darzustellen. Als Ersatz wurden die aus der Numerik bekannten Gleitkommazahlen verwendet, deren endliche Genauigkeit zu Fehlern und Artefakten fhren kann. Erst durch den Gebrauch numerischer Methoden konnten diese Fehler nicht nur verstanden, sondern auch behoben werden. Auch die Konstruktion von Beschleunigungsstrukturen, wie der BVH, konnte durch mathematische Optimierung erklrt und teilweise verbessert werden. Dies lsst sich dadurch erklren, dass Beschleunigungsstrukturen fr einen Raytracer immer auf ein Minimierungsproblem zurckzufhren sind. Die Berechnung der Strahldichte-Verteilung im Raum basiert auf physikalischen Theoremen der geometrischen Optik, wie zum Beispiel der Rendergleichung<sup>7</sup>. Die verschiedenen Materialien der Objekte lassen sich mithilfe der Elektrodynamik beschreiben und knnen durch ihre speziellen Eigenschaften die Auswertung der Strahldichte unter Umstnden vereinfachen. Die eigentliche Berechnung der Strahldichte kann sogar auf zwei Gebiete der Mathematik, der hheren Analysis und der Stochastik, zurckgefhrt werden. Die Simulation der Strahldichte ist quivalent zu dem Lsen einer Integralgleichung. Stze der hheren Analysis erlauben dem Entwickler das Problem in eine einfachere Form zu transformieren. Durch stochastische Methoden, wie der Monte-Carlo Methode, wird man dann in die Lage versetzt, die Lsung des Problems erwartungstreu zu schtzen.

Abschlieend lsst sich sagen, dass viele Aspekte der Mathematik und Physik im Projekt Anwendung fanden und dadurch gefestigt wurden. Umgekehrt ermglichte mir die Arbeit einen tiefen Einblick in die Verwendung professioneller Hilfsmittel und Arbeitsweisen. Mit dem entstandenen Quelltext konnte die Aufgabenstellung erllt werden. Das kompilierte Programm lief robust und effizient auf der CPU und GPU. Trotz allem htte ich gerne noch mehr Messungen und Tests durchgefhrt, um verschiedene Algorithmen miteinander zu vergleichen. Insgesamt motivierte mich die Arbeit, mich weiterhin mit dem Thema Raytracing vertiefend auseinanderzusetzen.

---

<sup>7</sup><https://de.wikipedia.org/wiki/Rendergleichung>