Karlsruher Institut für Technologie

Konzeption und Entwicklung einer parallelen Ray- & Pathtracing-Anwendung

Projektpraktikum

Gruppe Nostrum
Manuel Karl, Dominik Kleiser, Marc Leinweber, Nico Mürdter

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Erstgutachter: Philip Pfaffe, Martin Tillmann

28. November 2016 – 05. Februar 2017

1 Einleitung

Ray- und Pathtracing sind Techniken mit hoher Bedeutung. Durch Computer erzeugte Grafiken, die nicht in Echtzeit berechnet werden, werden mit Hilfe von Ray- und Pathtracing-Technologien gerendert. Die Algorithmen verfolgen dabei Lichtstrahlen, um zu berechnen, ob ein Primitiv, das heißt ein Objekt der Szenerie, beleuchtet und nicht verdeckt ist.
Diese Arbeit entstand im Rahmen der Veranstaltung „Praxis der Multikern-Programmierung: Werkzeuge, Modelle, Sprachen“ der Fakultät für Informatik am Karlsruher Institut für Technologie. Ziel ist es, zunächst einen hochparallelen und effizienten Raytracer zu implementieren. In einem zweiten Teil wurde ein Pathtracer implementiert.
Beim Raytracing werden für jeden Pixel des zu rendernden Bildes zu allen Punktlichtquellen der Szene ein Strahl versendet. Trifft der Strahl die Lichtquelle und wird er nicht durch ein anderes Primitiv blockiert, wird der Punkt beleuchtet. Im hier implementierten Modell wurde ein Whitted-Style-Raytracer[1] ohne Reflektion mit Lambert-Schattierung implementiert. Hierbei handelt es sich um eine sehr einfache Variante, da Parallelisierung und Effizienz im Vordergrund standen.
Das Pathtracing ist eine stochastische Variante des Raytracings, die mit Hilfe von Monte-Carlo-Integration die globale Beleuchtung einer Bildstelle simuliert. Es werden statt Punktlichtquellen Flächenlichtquellen benutzt. Das bedeutet, dass die Primitive der Szene Licht emittieren können. Für jeden Strahl, der ein Szenerieobjekt trifft, werden n zufällige Strahlen versendet, die wiederum bei einem Treffer neue Kindstrahlen erzeugen (*Diffuses Raytracing*). Die Beleuchtung der Stelle ergibt sich aus der Beteiligung all dieser Strahlen.
In Kapitel 2 werden zunächst die Problemstellungen diskutiert und Ansätze zur Parallelisierung erörtert. Im anschließenden Kaptitel 3 wird die Umsetzung der Ansätze beschrieben. Die Ergebnisse werden in Kapitel 4 evaluiert. Mit Kapitel 5 wird diese Arbeit abgeschlossen.

[1] `https://de.wikipedia.org/wiki/Raytracing`

2 Analyse und Entwurf

Das folgende Kapitel beschreibt die Analyse der Problemstellung und die betrachteten Ansätze zur Parallelisierung. Zu Beginn wurde eine naive und nicht parallelisierte Implementierung erstellt. Es wurde schnell klar, dass die reine Parallelisierung naiver Algorithmen nicht zielführend sind. So lag das Augenmerk bei der Suche nach effizienten und schnellen Algorithmen für Ray- bzw. Pathtracing. Des Weiteren war eine schnelle Datenstruktur von Nöten, um eine effiziente Traversierung der Primitive (Dreiecke) zu ermöglichen.

2.1 Analyse der Problemstellung

2.1.1 Raytracing

Der erste Raytracing-Ansatz bestand aus dem naiven Versuch, jeden Pixel Strahl zu verfolgen und eine einfache Schattenstrahlberechnung anzustellen. Dieser Ansatz lässt sich trivial parallelisieren. So konnte der naive Algorithmus mit OpenMP über alle Pixel paralellisiert werden. Hierbei wurde Datenparallelität benutzt. Als ein weiterführender Schritt wurde die Benutzung von Strahlenbündel eingeführt, welche mit Hilfe von Intel SSE-Instruktionen[1] realisiert wurden. Da eine der beiden Testmaschinen auch die verbesserte Version AVX unterstützt, wurde noch eine dritte Implementierung in AVX hinzugefügt.
Das Konzept der Strahlenbündel basiert auf der Annahme, dass die benachbarten Strahlen eines betrachteten Strahls mit hoher Wahrscheinlichkeit das gleiche Dreieck treffen und so auch die gleiche Traversierung der Datenstruktur vollziehen. Es lassen sich Bündel von Strahlen zusammenfassen und mit Hilfe von SSE-Instruktionen gleichzeitig betrachten. Da es sich bei SSE um interne Vektorinstruktionen handelt, welche pro Prozessorfaden getrennt voneinander operieren, ließ sich die Technik mit OpenMP kombinieren.

2.1.2 Pathtracing

Anders als beim Rayrtracing kann man beim Pathtracing nicht auf Strahlenbündel setzen, da die „Verfolgung“ eines Pfades auf zufälligen Richtungsänderungen basiert. Nur die von der Kamera wegführenden Strahlen besitzen die Eigenschaft, mit hoher Wahrscheinlichkeit das gleiche Dreieck zu treffen. Um dennoch eine gute Performanz beim Pathtracing zu erreichen, wurde hier auf die hochparallelisierte GPU gesetzt. Die Umsetzung ist hierbei ebenfalls ein datenparaller Ansatz über alle Pixel. Jeder GPU-Faden traversiert einen Pfad bis dieser terminiert.

[1] `https://software.intel.com/sites/landingpage/IntrinsicsGuide`

2.2 Datenstruktur

2.2.1 Räumliche Datenstruktur

Die Suche nach geschnittenen Dreiecken ist einer der aufwendigsten Schritte bei Ray- bzw. Pathtracing. Der naive Ansatz war eine vollständige Suche auf allen Dreiecken. Dabei wurden die Dreiecke unabhängig ihrer räumlichen Position betrachtet. Diese Vorgehensweise ist sehr ineffizient und skaliert nicht. Eine effiziente Lösung bietet hierfür eine Bounding Volume Hirachy in Form eines k-d-Baums.

Die Dreiecke werden anhand ihrer räumlichen Position in Bereiche unterteilt. Dazu wird der Raum rekursiv entlang einer Achsendimension in zwei kleinere Teilbereiche unterteilt, bis jeder Bereich nur noch eine kleine Anzahl an Dreiecken enthält. Die Unterteilungsmetrik kann verschieden gewählt werden. In einem ersten Ansatz wurde die naive Variante gewählt, bei der die Bereiche anhand des räumlichen Medians entlang der Achsendimension der größten Ausdehnung aufgeteilt wurden. Hierbei erhält man einen gut balancierten k-d-Baum.

Der zweite Ansatz bestand darin den k-d-Baum mit Hilfe der so genannten Surface Area Heuristic (SAH) zu teilen. Dabei werden die Kosten einer Bereichstraversierung gegenüber den Kosten für die Traversierung aller beinhalteten Dreicke abgeschätzt. Um hierbei ein gutes Minimum zu finden, werden alle Trennebenen miteinander verglichen. Die Trennebene mit den geringsten zukünftigen Bereichskosten gewinnt. Kann keine Trennebene gefunden werden, sodass die Traversierungskosten für alle Dreiecken größer als die einer Aufteilung sind, bricht die Rekursion ab.

Damit bei den beschriebenen Abläufen der Kopieraufwand für die Dreicke die Laufzeit nicht beeinträchtigt, wurden die Dreicke nur über ihren Index im ursprünglichen Vektor identifiziert. Dieses Konzept der Indirektion über Verktorindizes zieht sich durch sämtliche Teile des Projekts.

2.2.2 Axis Aligned Bounding Boxes und Dreiecksschnitt

Die Bereiche mit denen im oben beschriebenen k-d-Baum die Dreiecke räumlich unterteilt werden, können verschieden realisiert werden. Da in der hier verwendeten Implementierung nur an Achsen getrennt wird, sind die Bereiche an den Achsendimensionen orientiert. Man spricht von Axis Aligned Bounding Boxes. Hieraus ergeben sich noch weitere Vorteile. So kann man die ganze Bounding Box (im folgenden „BB“) allein durch zwei Punkte darstellen, welche die minimale bzw. maximale Ausdehnung der Box darstellen. Des Weiteren werden Schnitte einfacher, da man sich an den Achsendimensionen orientieren kann.

Die Dreiecksschnittberechnung ist eine der am meisten verwendeten Funktionen im gesamten Projekt. Deshalb wirkt sich ein effizienter Algorithmus hier sehr stark auf die Gesamtlaufzeit aus. Um dies zu realisieren, wurde der Dreiecksschnittalgorithmus von Tomas Möller und Ben Trumbore verwendet [4].

2.3 Parallelisierungsansätze

2.3.1 OpenMP

Aufgrund der oben beschriebenen Datenparallelität beim Raytracing, bietet sich eine Parallelisierung mittels OpenMP an. Hier können Schleifen durch Direktiven an den Übersetzer parallelisiert werden: `#pragma omp parallel for`. Die Schleife wird in gleich große Abschnitte unterteilt, die jeweils von einem Thread abgearbeitet werden. Dies ist besonders nützlich bei der Schnittberechnung, die für jeden Pixel durchgeführt werden muss. Zu beachten ist, dass die einzelnen Schleifeniterationen keine Abhängigkeiten voneinander haben dürfen. Durch die rekursive Erzeugung des k-d-Baums ergibt sich die Möglichkeit des Taskparallismus mittels `#pragma omp task`. Somit kann ein Thread „Arbeitspakete“ erstellen, die von weiteren Fäden bearbeitet werden. Hierbei ist eine Begrenzung der Task-Tiefe wichtig, da ab einer gewissen Tiefe der Mehraufwand zwischen Task-Erzeugung und Task-Bearbeitung zu groß wird und ein Verlust an Effizienz eintritt. OpenMP bietet sich im speziellen an, da die von OpenMP verwendeten Direktiven immer lauffähig sind und mittels einfacher Compiler-Flags aktiviert werden können. Es ist somit möglich einfach inkrementell zu parallelisieren.

2.3.2 Strahlschnittparallelisierung mit SIMD-Anweisungen

Durch die explizite Vektorisierung der Strahlen können mehrere Schnittberechnungen gleichzeitig durchgeführt werden. Dies wird erreicht, indem die Strahlen, repräsentiert durch mehrere 32 Bit Gleitkommazahlen, abhängig vom Prozessor durch AVX- bzw. SSE-Instruktionen bearbeitet werden. Bei AVX Instruktionen können 256 Bit große Register, bei SSE Instruktionen 128 Bit große Register benutzt werden. So ist es es möglich vier (SSE) oder acht (AVX) Strahlen in einem Register zu speichern. Anschließend können die Schnittberechnungen auf das Register angewandt werden.

2.3.3 Massive SIMD-Parallelisierung auf der GPU

Aufgrund der exponentiell steigenden Anzahl an Strahlen beim Pathtracing, bietet sich hier die Parallelisierung auf der GPU an. Diese bietet eine sehr hohe Anzahl an Fäden, sodass jeder Strahl von einem Faden bearbeitet werden kann. Abzuwägen ist hierbei der Mehraufwand zwischen Kopieraufwand von CPU auf GPU im Vergleich zur hoch parallelen Datenverarbeitung auf der GPU.
Ein Problem der GPU stellt die rekursive Berechnung der Schnitttests und der Datenstruktur dar. Die GPU ist nicht für rekursive Funktionalitäten ausgelegt. Des Weiteren sollte man aufgrund des Locksteps innerhalb eines GPU-Warps möglichst auf Kontrollstrukturen verzichten. Bei einer bedingten Verzweigung müssen beide möglichen Wege von allen 32 Fäden durchlaufen werden. Nicht verwendbare Ergebnisse werden anschließend verworfen. Man erzeugt somit im schlimmsten Fall den doppelten Rechenaufwand pro Warp.

3 Umsetzung und Implementierung

Zur Umsetzung des Projekts wurde uns von Seiten des Lehrstuhls ein GitLab-Repository mit integriertem Continuous-Integration-Server (CI-Server) zur Verfügung gestellt. Programmiert wurde in portablem und modernem C++ (im aktuellen Standard *C++14*). Außerdem haben wir uns für das Buildsystem *cmake* und für die graphische Entwicklungsumgebung *CLion* entschieden. Die meiste Zeit wurde via *Teamviewer* mit der aus der agilen Softwareentwicklung bekannten Arbeitstechnik *Paarprogrammierung* gearbeitet.
Ziel war es, kompatible Software für zwei vorgegebene Test- und Benchmark-Systeme zu entwickeln (im Folgenden *i41pc189* und *i41pc205* genannt). Die Anwendung wurde aber für alle gängigen Systeme ausgelegt; Programmcode und CMake-Konfiguration wurden so eingerichtet, dass nur für die aktuelle Umgebung kompatible Maschinenbefehle kompiliert werden (z.B. Abschalten von AVX bei fehlender Hardwareunterstützung).
Die einzigen von der Aufgabenstellung zugelassenen Bibliotheken waren:

- `https://github.com/nothings/stb` zum Schreiben von Bitmaps,
- `https://nlohmann.github.io/json/` zum Lesen von JSON-Dateien,
- `https://github.com/syoyo/tinyobjload` zum Laden von OBJ-Dateien

sowie die NVIDIA Bibliotheken *CUDA* und *Thrust* zur Programmierung der Grafikkarte (GPGPU-Programmierung).

3.1 Software-Architektur

Beide Aufgabenteile (Raytracing und Pathtracing) mussten in einer gemeinsamen Anwendung implementiert werden. Welches der beiden Tracing-Verfahren zum Einsatz kommt wird dabei dynamisch zur Laufzeit aus der Szenen-Beschreibung (JSON-Datei) ausgelesen. Da anfangs noch mit Anpassungen und Erweiterungen der Aufgabenstellung zu rechnen war, entschieden wir uns für einen möglichst modularen und objektorientierten Software-Entwurf. Eine Auflistung und Beschreibung der entwickelten Module findet sich in Tabelle 3.1.
Der Programmablauf ließ sich in die folgenden Schritte untergliedern:

1. Dateieingabe: Lesen der Szenen-Beschreibung (JSON) und Obj-Datei

2. Vorverarbeitung der Szene: Vorberechnungen für die Dreiecke (z.B. Berechnung der Normalen), Konstruieren des k-d-Baums und Berechnung der Primärstrahlen; in diesem Schritt werden auch anhand der Szene Weichen für das weitere Vorgehen gestellt (Wahl des k-d-Baums, Entscheidung, ob GPU verwendet werden soll)

3. Rendering: Durchführung des Tracing-Verfahrens

4. Dateiausgabe: Schreiben des Ergebnisbildes in eine Bitmap-Datei

Modulname	**Beschreibung**
main.h	Einstiegspunkt mit Mainfunktion; verbindet die Komponenten miteinander und misst deren Ausführungszeiten.
options.h	Datei zum Setzen verschiedenerer statischer Programmparameter; zum Ein- und Ausschalten von AVX, SSE und CUDA.
io.h	Zuständig für das Parsen der JSON-Szenen-Beschreibung und der Wavefront OBJ-Datei.
scene_preprocessor.h	Vorverarbeitung der Szene (siehe oben).
pray.h	Mit dem Aufruf von `render(...)` wird das Tracing-Verfahren gestartet; die Entscheidung ob Ray- oder Pathtracing durchgeführt wird, wird anhand der Informationen aus dem Szene-Objekt (Parameter) entschieden.
geometry.h	Definiert die benötigten geometrischen Primitiven (*Vec3*, *Triangle*, *Bounding Box*, usw.) und die auf ihnen ausgeführten Algorithmen (z.B. Dreieck- und BB-Schnitt).
geometry_simd.h	Erweitert die Algorithmen aus *geometry.h* um deren SSE und AVX Pendants (Intrinsics).
spatial.h bf_search.h kd_tree.h kd_tree_sah.h kd_tree_hybrid.h	Schnittstelle für räumliche Datenstrukturen und die dazugehörigen Instanziierungen (siehe Abschnitt 3.2).
path_trace.cu.h kd_tree_gpu.h gpu_util.h	CUDA-Code für paralleles Pathtracing auf der GPU.

Tabelle 3.1: Beschreibung der Module

3.2 Implementierung der räumlichen Datenstruktur

Es wurde schnell klar, dass die Schnittberechnung der Dreiecke mit den Strahlen der Flaschenhals der Laufzeit werden würde. Um hier möglichst einfach zwischen alternativen räumlichen Datenstrukturen wechseln zu können, entschieden wir uns für eine Vererbungshierarchie wie in Abbildung 3.1 dargestellt.

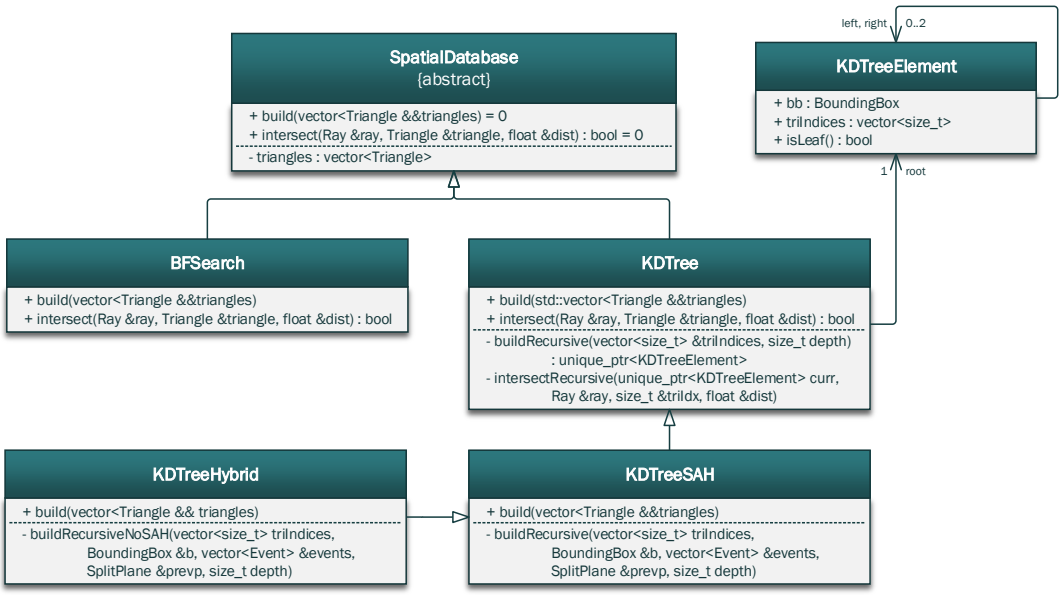

Abbildung 3.1: Klassendiagramm: Räumliche Datenstruktur

Die Klasse `SpatialDatabase` stellt zwei abstrakte Methoden `build` und `intersect` bereit. Diese werden dann von den Unterklassen `BFSearch` und `KDTree` implementiert. `BFSearch` führt bei der Schnittberechnung eine vollständige Suche über alle Dreiecke durch. Beim bauen wird lediglich ein *move* aller Dreiecke in die Instanzvariable *triangles* vorgenommen. In `KDTree` wird in `build` ein naiver k-d-Baum aufgebaut. `KDTreeSAH` erzeugt k-d-Bäume gemäß der Surface-Area-Heuristik. Die Schnittberechnung von `KDTreeSAH` wird von `KDTree` geerbt.

Der Aufbau der beiden k-d-Bäume erfolgt rekursiv und wurde wie in Unterabschnitt 2.3.1 beschrieben mit *OpenMP*-Tasks parallelisiert. Neue Tasks werden jedoch nur bis zu einer bestimmten Tiefe **TASK_DEPTH** erzeugt. Die maximale Tiefe des konstruierten Baums und die Mindestzahl benötigter Dreiecke für einen weitere Teilung sind durch die beiden Parameter **MAX_DEPTH** und **THRESHOLD** beschränkt. Für die Konstruktion des Baums mit SAH wurde zunächst der Algorithmen in $O(n \log^2 n)$ aus [6] implementiert. Später wurde dann zur Beschleunigung ein Wechsel auf den $O(n \log n)$-Algorithmus vorgenommen.

3.3 Implementierung des Raytracers

Der Raytracer wurde zunächst in einer möglichst einfachen, sequentiellen Variante entworfen. Der Dreiecksschnitt wurde von Anfang an mit dem Möller–Trumbore-Schnittalgorithmus durchgeführt. Im nächsten Schritt wurde die vollständige Suche nach dem Dreieck mit geringstem Abstand durch eine wesentliche effizientere Suche in einem k-d-Baum ersetzt (siehe Unterabschnitt 2.2.1 und Abschnitt 3.2). Der erste Ansatz zur Parallelisierung war eine Verteilung der Bildpixel/Primärstrahlen auf mehrere Threads. Da hierbei keine Abhängigkeiten zwischen Pixeln existieren (*embarrassingly parallel*), konnte dies auf einfache Weise mit OpenMP vorgenommen werden. Eine weitere Beschleunigung konnte anschließend durch die Bündelung von Strahlen erreicht werden (siehe Unterabschnitt 2.1.1). Da die Testmaschine *i41pc205* keine AVX-Instruktionen unterstützt, entschieden wir uns dafür, abhängig von der Zielplattform, unser Programm sowohl für SSE (vier Strahlen) als auch für AVX (acht Strahlen) zu optimieren. Da jeder Prozessor seine eigenen SIMD-Register besitzt, skaliert dieser Ansatz auch äußerst gut mit der vorherigen Parallelisierung mit *OpenMP*. Um auf die hardwarenahen SIMD-Instruktionen zurückzugreifen, wurden intrinsische Funktionen (Intrinsics) verwendet. Hierzu musste der Möller-Trumbore-Algorithmus für SIMD umgeschrieben werden und es waren Anpassungen in der k-d-Baum Schnittberechnung erforderlich.

3.4 Implementierung des Pathtracers

Eine erste Version des Pathtracers konnte durch nur wenige Anpassungen am schon vorhandenen Raytracer implementiert werden. Es zeigte sich jedoch, dass aufgrund der mit der Rekursionstiefe exponentiell wachsenden Anzahl Strahlen eine reine Parallelisierung über die Pixel zu unzufriedenstellenden Laufzeiten führt. Wie in Unterabschnitt 2.1.2 beschrieben ist eine Parallelisierung des Pathtracings mit den SIMD-Registern der CPU nur eingeschränkt sinnvoll. Um das Tracing auf der GPU durchführen zu können mussten viele der schon vorhandenen Funktionen aus `geometry.h` in CUDA-Kernel-Code überführt werden. Außerdem muss der k-d-Baum in eine für die GPU besser geeignete Form umkopiert werden. Bei der Implementierung des Verfahrens in CUDA kam es zu mehreren Schwierigkeiten:

- Da das Programmiermodell von Grafikkarten nicht für rekursive Funktionsaufrufe ausgelegt ist, versuchten wir in unserem ersten Lösungsansatz diese aus dem Programmcode zu eliminieren. Dazu mussten in einem Array alle Strahlen (über alle Pixel) auf der momentanen Rekursionstiefe zwischengespeichert werden. Dieser Ansatz musste später aufgrund eines zu hohen Speicherverbrauches (sowohl auf der GPU als auch auf der CPU) verworfen werden.

- Bei der rekursiven Variante kam es ebenfalls zu Problemen mit dem GPU-Speicher. Hier kommt es zu einer zweifachen Rekursion: Einerseits die beim Pathtracing vorgebenene Rekursion beim Auftreffen auf ein Dreieck, andererseits der rekursive Abstieg auf dem k-d-Baum für die Schnittberechnung. Eine „Stackless“-Traversierung

(siehe [5]) des k-d-Baum gestaltete sich als deutlich schwieriger zu implementieren als zunächst erwartet.

- Eng verwandt mit dem vorherigen Punkt ist die Festellung, dass ein k-d-Baum der gemäß der SAH-Metrik aufgebaut wurde, nur eingeschränkt für die GPU geeignet ist. SAH neigt dazu deutlich tiefere Bäume zu erzeugen, was bei der Traversierung zu einer größeren Rekursionstiefe führt. Mit einem naiven k-d-Baum konnten teilweise deutlich bessere Laufzeiten erzielt werden. Um ein gutes Mittelmaß bei den Laufzeiten zu finden, wurde eine SAH-Abwandlung eingeführt, welche auf den ersten Ebenen die naive Unterteilungsmetrik verwendet und erst dann auf SAH wechselt. Siehe `KDTreeHybrid` in 3.1.

Da andere Gruppen beim Pathtracing deutlich geringere Laufzeiten erzielten als unserer Meinung nach mit dem vorgeschriebenem Algorithmus (auch als *diffuses Raytracing* bezeichnet) möglich ist, wurde ebenfalls noch eine etwas abgewandelte Variante des Verfahrens implementiert. Bei dem in [3] vorgestellten Algorithmus zum Pathtracing wird nicht bei jedem Treffer auf ein Objekt ein neuer Strahl erzeugt, stattdessen wird **NUM_SAMPLES** oft über jedes der Pixel iteriert und jeweils nur ein Pfad verfolgt. Am Ende wird der berechnete Pixelwert dann durch **NUM_SAMPLES** geteilt. Dies führt zu vergleichbaren Ergebnisbildern, erzeugt aber bei Rekursionstiefen größer eins deutlich weniger Strahlen; die Anzahl der Strahlen ist linear und nicht exponentiell zu **NUM_SAMPLES**.

4 Evaluation

In diesem Kapitel wird die Leisungsfähigkeit der Implementierung mit Hilfe verschiedener Benchmarks und Vergleiche untersucht.

Tabelle 4.1: Eigenschaften der Szenen

Szene	Auflösung	Dreiecke	Lichtquellen	Rekursiontiefe	Samples
Cube	8000x6000	12	2	-	-
Sponza	2048x2048	262267	1	-	-
Cornell (PT)	800x600	14	8	1	4000
Sponza (PT)	400x400	262267	12	1	32

4.1 Hauptspeicher

Für eine Szene mit 262267 Primitiven wird während des kompletten Renderprozesses ungefähr 1GB Hauptspeicher benutzt. Soll die Szene auf der GPU berechnet werden, wird geprüft, ob die zu benutzende GPU über genug Speicher verfügt.

Tabelle 4.2: Laufzeiten der lokalen Benchmarks

Szene	naiv	k-d-Baum	+ AVX	+ SAH	GPU	Heuristik
Cube	00:03:84	00:03:73	00:03:42	00:03:21	-	00:03:50
Sponza	∞	01:41:17	00:21:51	00:06:79	-	00:06:38
Cornell (PT)	∞	01:22:01	-	01:25:76	00:22:31	00:20:05
Sponza (PT)	∞	07:17:73	-	00:17:84	01:29:12	00:17:71

4.2 Laufzeit

Die Implementierung wurde auf den Praktikumssystemen sowie auf einer Enticklungsmaschine (im Folgenden „lokaler Benchmark“) in der Laufzeit evaluiert. Die lokalen Benchmarks wurden auf einem System mit Ubuntu 16.04, Intel Core i5-6600K CPU @ 3.50GHz, 32 GB Hauptspeicher und einer Nvidia Geforce GTX 1070 Grafikkarte mit 8 GB Grafikspeicher ausgeführt. In Tabelle 4.2 sind die Zeiten des lokalen Benchmarks aufgelistet. Getestet wurden vier verschiedene Szenen (Cube, Sponza, Cornell-Box mit Pathtracing und Sponza mit Pathtracing). In Tabelle 4.1 sind die Eigenschaften der Benchmarkszenen gelistet. Bei Pathtracing handelt es sich um Flächenlichtquellen. Das heißt es gibt Flächen die Licht emittieren und keine gesonderten Punktlichtquellen. Verglichen werden naive Implementierung (vollständige Suche), k-d-Baum, k-d-Baum mit AVX-Strahlbündelschnitt, selbiger mit SAH-Heuristik und für die Pathtracing-Szenen das Rendern auf der GPU (Nutzung des trivialen k-d-Baumes). Die Spalte Heuristik zeigt die erreichten Zeiten, wenn die Implementierung Datenstruktur und Ausführungsumgebung selbst wählt. Alle Benchmarkes wurden OpenMP-parallelisiert ausgeführt.
Nachfolgend sind die Ergebnisse der Benchmarks der Wettbewerbsmaschinen zu sehen. In Abbildung 4.1 sind die Ergebnisse mit dem Pathtracing Algorithmus aus [3] zu sehen und in 4.2 sind die Ergebnisse mit dem vorgegebenen Algorithmus zu sehen.

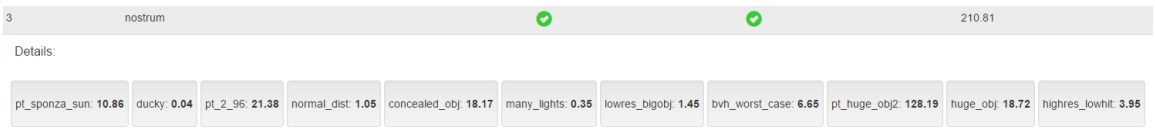

Abbildung 4.1: Wettbewerbsmaschine *i41pc189*: Pathtrace Algorithmus aus [3]

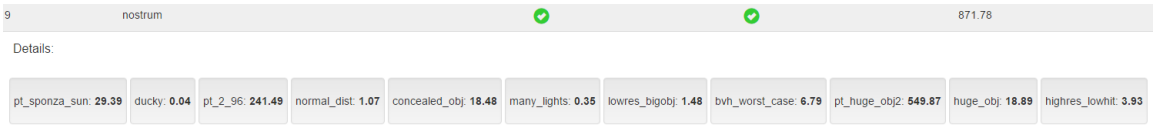

Abbildung 4.2: Wettbewerbsmaschine *i41pc189*: Vorgegebener Pathtrace Algorithmus

5 Fazit

In dieser Arbeit wurde ein massiv paralleler Raytracer sowie ein Pathtracer implementiert. Dazu wurden die Problemstellungen analysiert und vorhandene Arbeiten auf ihre Tauglichkeit untersucht. Aufgrund dieser Ergebnisse wurde ein CPU-Raytracer mit zwei verschiedenen Datenstrukturen (k-d-Baum mit und ohne SAH-Heuristik) sowie ein CPU- und GPU-Pathtracer implementiert.
Während der Arbeit zeigte sich, dass Wahl und Aufbau der Datenstruktur stark am Erfolg einer effizienten Lösung teilhaben. Trotz der zunächst trivial erscheinenden Parallelisierung, ergaben sich Probleme gerade für sehr große Szenen. Neben der Datenstruktur war auch das Pathtracing aufgrund des exponentiellen Wachstums schwer zu bewältigen. Es ergaben sich große Probleme bei der Wahl der Ausführungsumgebung (GPU vs CPU). Nichtsdestotrotz konnten auch für sehr große Szenen sehr gute Renderzeiten bei Full-HD-Auflösungen erreicht werden.
Als offene Punkte bleiben die stapelfreie Implementierung des GPGPU-Pathtracing und des k-d-Baum-Schnittes sowie eine Evaluierung des Raytracings auf der GPU.

Literatur

[1] Andrew Adams u. a. “Gaussian kd-trees for fast high-dimensional filtering”. In: *ACM Transactions on Graphics (ToG)*. Bd. 28. 3. ACM. 2009, S. 21.

[2] Byn Choi u. a. “Parallel SAH kD tree construction”. In: *Proceedings of the Conference on High Performance Graphics*. Eurographics Association. 2010, S. 77–86.

[3] James T. Kajiya. “The Rendering Equation”. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), S. 143–150. ISSN: 0097-8930. DOI: 10.1145/15886.15902. URL: http://doi.acm.org/10.1145/15886.15902.

[4] Tomas Möller und Ben Trumbore. “Fast, minimum storage ray/triangle intersection”. In: *ACM SIGGRAPH 2005 Courses*. ACM. 2005, S. 7.

[5] Stefan Popov u. a. “Stackless KD-Tree Traversal for High Performance GPU Ray Tracing”. In: *Computer Graphics Forum*. Bd. 26. 3. Wiley Online Library. 2007, S. 415–424.

[6] Ingo Wald und Vlastimil Havran. “On building fast kd-trees for ray tracing, and on doing that in O (N log N)”. In: *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE. 2006, S. 61–69.