UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

COMPUTERVISUALISTIK

Global Illumination mittels GPU Path Tracing und der Line Space Datenstruktur

Bachelorarbeit

zur Erlangung des Grades Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Felix-León Schröder

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

Koblenz, im März 2017

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Abbildungen habe ich, soweit nicht anders angegeben, selbst erstellt.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐

Koblenz, 23. März 2017

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Diese Arbeit zeigt eine neue Technik der Computergrafik zur Simulation von globaler Beleuchtung durch Path Tracing in Echtzeit. Das Path Tracing wird dafür mit Compute Shadern auf der Grafikkarte (*GPU*) realisiert, um das Rendering hoch parallelisiert auszuführen. Zur Beschleunigung der Strahlverfolgung wird dabei der Line Space in verschiedenen Varianten als Datenstruktur verwendet, um leere Bereiche in der Szene schneller zu überspringen. Der Line Space speichert Szeneninformationen basierend auf einer Voxelisierung in richtungsabhängige *Shafts* und wird sowohl auf der GPU generiert, als auch traversiert. Mit diesem Verfahren kann eine Szene physikalisch korrekt indirekt beleuchtet und mit weichen Schatten schattiert werden. Außerdem kann das Path Tracing damit weitgehend unabhängig von der Polygonanzahl mit über 100 Bildern pro Sekunde klar in Echtzeit durchgeführt werden und ist somit deutlich schneller als mit einem vergleichbaren Voxel-Gitter. Die Bildqualität wird davon nicht negativ beeinflusst und die Schattenqualität ist in den meisten Fällen deutlich besser als bei der Verwendung von Shadow-Mapping.

Abstract

This thesis presents a novel technique in computer graphics to simulate real-time global illumination using path tracing. Path tracing is done with compute shaders on the graphics card (*GPU*) to perform rendering in a highly parallelized manner. To improve the overall performance of tracing rays, the Line Space is used as an acceleration data structure in different variations, resulting in better empty space skipping. The Line Space saves scene information based on a previous voxelization in direction-dependent shafts and is generated and traversed on the GPU. With this procedure, indirect lighting and soft shadows can be computed in a physically correct way. Furthermore, using the Line Space, path tracing can be performed mostly independent of the complexity of the scene geometry with over 100 frames per second, which is truly real-time and much faster than using a comparable voxel grid. The image quality is not affected negatively by this technique and the shadow quality is in most cases much better compared to shadow-mapping.

Inhaltsverzeichnis

1 Einleitung

In keinem Bereich der Informatik hat sich die Hardware so rasant entwickelt wie im Bereich der Grafikprozessoren. So ist es heute möglich, Ray bzw. Path Tracing in Echtzeit auf der Grafikkarte (*GPU*) durchzuführen. Weiterhin gibt es verschiedene Datenstrukturen zur Beschleunigung, wie beispielsweise das Voxel-Gitter oder den Line Space. Die große Herausforderung dabei ist es, die einzelnen Verfahren so zu kombinieren bzw. zu implementieren, dass die Bildberechnung auch in Kombination mit globaler Beleuchtung echtzeitfähig bleibt und eine gute visuelle Qualität aufweist.

Ziel dieser Bachelorarbeit ist es, sich in die Programmierung der GPU und in das Path Tracing Verfahren einzuarbeiten und auf Basis dessen einen GPU-basierten Path Tracer zu entwickeln. Dafür wird der bereits funktionsfähige Ray Tracer der Computervisualistik Koblenz erweitert. Zur Beschleunigung des Path Tracers soll anschließend der Line Space als Datenstruktur verwendet bzw. weiterentwickelt werden. Damit soll die globale Beleuchtung in Echtzeit berechnet werden können.

In dieser Ausarbeitung werden zuerst die theoretischen Grundlagen der globalen Beleuchtung, des Path Tracings und des Line Spaces gelegt. Darauf aufbauend wird im zweiten Teil die Implementierung des Path Tracers und des Line Spaces inklusive einiger Variationen erläutert. Abschließend werden die Ergebnisse des auf dem Line Space basierenden GPU Path Tracers sowohl auf ihre Performance, als auch auf ihre visuelle Qualität untersucht und damit die Frage beantwortet, ob Path Tracing und somit die globale Lichtsimulation mit dem Line Space in Echtzeit möglich ist.

Teil I
Theoretische Grundlagen

Dieser Teil der Arbeit legt die theoretischen Grundlagen der in der Implementierung angewandten Verfahren. Dafür wurden diverse Quellen (siehe Literaturverzeichnis) als Referenz verwendet.

2 Globale Beleuchtung

Globale Beleuchtung beschreibt in der Computergrafik die physikalisch korrekte Simulation von Licht in einer virtuellen Szene, insbesondere wie sich dieses in der Szene verteilt [Kaj86]. Der Fokus liegt dabei neben der einfachen direkten Beleuchtung auch auf spezielleren Lichteffekten. Dazu zählt neben Reflexionen und Lichtbrechungen auch besonders die indirekte Beleuchtung. Letztere beschreibt das Phänomen, dass nahezu alle Materialien nur einen Teil des eintreffenden Lichtes absorbieren und den Rest reflektieren. Dieses reflektierte Licht beleuchtet dann

wiederum andere Objekte, welche wieder nur einen Teil des Lichts absorbieren, und so weiter, was in einer rekursiven Lichtstreuung resultiert. Damit ist es beispielsweise möglich, ein Zimmer nur mit dem durch einen Türspalt von außerhalb einfallenden Licht von einer anderen (nicht sichtbaren) Lichtquelle zu beleuchten.

2.1 Die Rendering-Gleichung

Die globale Beleuchtung wurde erstmals 1986 von Kajiya [Kaj86] und Immel *et al.* [ICG86] formalisiert. Die Autoren vereinten darin die meisten Beleuchtungseffekte in einer Formel, welche seitdem als die „Rendering-Gleichung" (1) bekannt und von großer Bedeutung ist. Sie beschreibt das Licht L, das von einem Punkt x in Richtung $\vec{\omega}$ abgestrahlt wird ($L(x, \vec{\omega})$) und summiert dazu das gesamte eintreffende Licht auf.

Das Material von x ist dabei durch eine BRDF definiert. Die BRDF (*engl:* ***B**idirectional **R**eflectance **D**istribution **F**unction*) beschreibt abhängig von der Eintritts- und Austrittsrichtung des Lichts ($\vec{\omega}'$ und $\vec{\omega}$, daher „bidirektional") die Reflexionseigenschaften des Materials.

$$L(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_{\Omega} f_r(x, \vec{\omega}', \vec{\omega})\, L(x, \vec{\omega}')\, (\vec{\omega}' * \vec{n})\, d\vec{\omega}' \tag{1}$$

Die einzelnen Komponenten der Formel beschreiben die folgenden Eigenschaften des Punktes x:

- L_e: Das vom Punkt selbst emittierte Licht in Richtung $\vec{\omega}$ (falls dieser wie z.B. eine Lampe selbst strahlt)
- $\int_{\Omega}$: Das Integral über alle Winkel der Hemisphäre über x um $\vec{n}$
- $f_r(x, \vec{\omega}', \vec{\omega})$: Die BRDF am Punkt x
- $L(x, \vec{\omega}')$: Das einfallende Licht
- $\vec{\omega}$: Die Richtung des Betrachters bzw. der Kamera
- $\vec{\omega}'$: Die Richtung des einfallenden Lichts
- $\vec{n}$: Die Normale von x

Da in der Praxis nicht über alle möglichen Richtungen integriert werden kann, wird das Integral in der Regel durch Aufsummieren diskreter Samples angenähert. Das einfallende Licht ($L(x, \vec{\omega}')$) kann dabei wiederum durch die Rendering-Gleichung beschrieben werden, wodurch man die oben beschriebene rekursive Lichtstreuung erhält.

2.2 Phong-Modell

Das Phong-Beleuchtungsmodell wurde bereits 1975 in [Pho75] vorgestellt. Auch wenn es vor der Rendering-Gleichung erfunden wurde, stellt es eine vereinfachte Annäherung der Gleichung bzw. der darin enthaltenen BRDF dar. Das Modell besteht grundsätzlich aus zwei Teilen: Dem diffusen und dem spiegelnden

(auch *spekularen*) Beleuchtungsterm (siehe Abbildung 1). Beide Teile sind zum einen von einer Lichtquelle abhängig, weshalb die Beleuchtungswerte von mehreren Lichtquellen addiert werden. Zum anderen spielt auch das Oberflächenmaterial eine Rolle, welches unter Anderem durch einen diffusen (k_d) und einen spekularen (k_s) Beleuchtungsanteil mit $k_s + k_d \leq 1$ definiert ist. Oftmals wird additiv noch ein ambienter Beleuchtungsterm hinzugefügt, welcher eine überall gleich wirkende Grundbeleuchtung darstellt. Gleichung (2) beschreibt diese Zusammensetzung. Dieses Modell wurde in der vorliegenden Arbeit zur Beleuchtung der Szenen verwendet.

$$L_{Phong}(x) = L_{ambient} + \sum_{L_i \in Lights} L_{diffuse}(x, L_i) + L_{specular}(x, L_i) \quad (2)$$

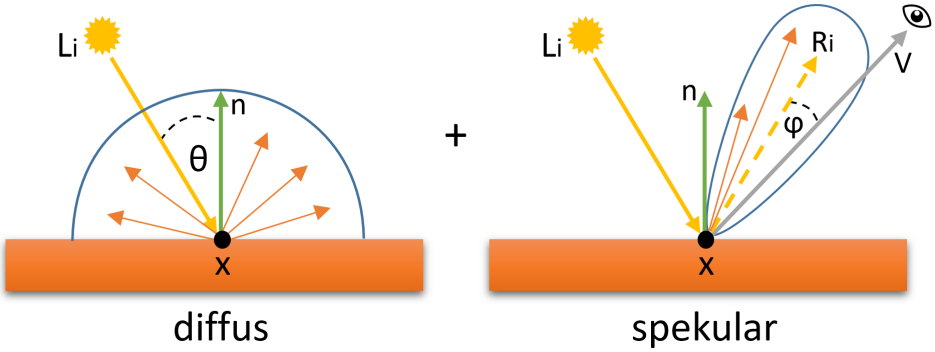

Abbildung 1: Diffuse und spekulare Beleuchtung im Phong-Modell. Die orangen Pfeile und blauen Formen symbolisieren die Lichtstreuung

2.2.1 Diffuse Beleuchtung

Die diffuse Beleuchtung (auch *Lambert-Term*) ist vom Einfallswinkel des Lichts auf eine Oberfläche abhängig. Strahlt die Lichtquelle senkrecht auf eine Fläche (Lichtvektor und Normale sind nahezu parallel), so ist die Beleuchtung am stärksten. Ist der Winkel θ zwischen Lichtvektor ($\overrightarrow{pos(x)pos(L_i)}$) und Normale ($\vec{n}$) jedoch groß oder sind die Vektoren sogar voneinander abgewandt, so kommt wenig bzw. kein Licht bei der Fläche an. Sind die beiden relevanten Vektoren normiert, so kann die Beleuchtungsintensität anhand des Skalarprodukts berechnet werden. Die Berechnung ist in Formel (3) dargestellt, wobei $col(L_i)$ die Farbe der Lichtquelle und $k_d \in [0, 1]$ der diffuse Beleuchtungsanteil des Oberflächenmaterials ist.

$$L_{diffuse}(x, L_i) = col(L_i) \cdot k_d \cdot \cos(\theta) = col(L_i) \cdot k_d \cdot (\overrightarrow{pos(x)pos(L_i)} * \vec{n}) \quad (3)$$

2.2.2 Spekulare Beleuchtung

Der spekulare Term ist anders als der diffuse sowohl von der Position der Lichtquelle, als auch von der Betrachter- oder Kameraposition abhängig und beschreibt eine nicht-ideale, keulenförmige Reflexion des Lichtvektors an der Oberflächennormale. Somit ist der Beleuchtungswert bei der spekularen Beleuchtung abhängig vom Winkel φ zwischen dem reflektierten Lichtvektor $\vec{R}_i$ und dem Kameravektor $\vec{V}$. Auch dieser Winkel (bzw. dessen Cosinus) kann bei normierten Vektoren mit dem Skalarprodukt berechnet werden. Dieser Wert wird dann mit der Glanzzahl s potenziert, was den Eindruck einer glänzenden Oberfläche (z.B. Plastik) erzeugt. Je höher die Glanzzahl ist, desto „perfekter" ist die Reflexion und desto glatter wirkt das Material; eine vollkommen perfekte Reflexion ist damit allerdings nicht möglich, da dafür $s = \infty$ sein müsste. Formel (4) zeigt die Berechnung dieses Beleuchtungsterms, wobei $k_s \in [0, 1]$ der spekulare Beleuchtungsanteil des Oberflächenmaterials ist.

$$L_{specular}(x, L_i) = col(L_i) \cdot k_s \cdot \cos(\varphi)^s = col(L_i) \cdot k_s \cdot (\vec{R}_i * \vec{V})^s \qquad (4)$$

2.2.3 Alternative Beleuchtungsmodelle

Der große Vorteil des Phong-Modells ist, dass es sehr einfach zu berechnen ist. Allerdings hat es auch einige Schwachstellen, da es viele Materialeigenschaften nicht beschreiben kann. Alternativen zu diesem Modell sind etwa das *Cook-Torrance-Modell* oder das *Schlick-Modell*, welche deutlich mehr Wert auf die physikalische Korrektheit und diverse Materialeigenschaften wie Mikrofacetten und Fresnel-Terme legen, dadurch aber auch wesentlich komplexer zu berechnen sind [PH04, Kap. 9].

2.3 Spezielle Lichteffekte

Neben der diffusen Reflexion, welche durch das Phong-Modell beschrieben wird, können Materialien auch noch viele andere Eigenschaften haben. Diese müssen durch andere Verfahren berechnet werden, da sie nicht im Phong-Modell enthalten sind. Eine perfekte Reflexion kann durch eine BRDF beschrieben werden. Transparente Materialien besitzen einen Brechungsindex und können analog zur BRDF durch eine BTDF (**B**idirectional **T**ransmittance **D**istribution **F**unction) beschrieben werden [PH04, Kap. 9]. Für Effekte wie Volumenstreuung (*Subsurface Scattering*) kann eine BSSRDF (**B**idirectional **S**cattering **S**urface **R**eflectance **D**istribution **F**unction) verwendet werden.

3 Path Tracing

Path Tracing ist eine spezielle Art des Ray Tracings. Im Folgenden werden zunächst die grundlegenden Algorithmen und Bausteine für Ray und Path Tracing erklärt und diese anschließend zum Path Tracing Algorithmus vereint.

3.1 Ray Tracing

Ray Tracing beschreibt zunächst lediglich die Verfolgung von Strahlen durch eine virtuelle Szene. Dabei kann entweder von der Lichtquelle (*forward Ray Tracing*) oder der Kamera (*backward Ray Tracing*) aus gestartet werden. In den meisten Fällen wird das backward Ray Tracing verwendet, da es deutlich schneller zum gewünschten Ergebnis führt; die Wahrscheinlichkeit, dass ein Lichtpfad die Kamera trifft ist beim forward Ray Tracing sehr gering. Abbildung 2 zeigt einen Vergleich der beiden Pfadverfolgungsstrategien. In beiden Fällen wird ein Strahl generiert und dessen vorderster Schnittpunkt mit der Szenengeometrie ermittelt. Anschließend kann der gefundene Schnittpunkt beleuchtet werden und es können weitere Strahlen ausgehend vom Schnittpunkt generiert und verfolgt werden. Zusätzlich können beim forward Ray Tracing an jedem Schnittpunkt Schattenfühler (bzw. Schattenstahlen) in Richtung der Lichtquellen verfolgt werden, um zu berechnen ob der Schnittpunkt verschattet ist oder von der entsprechenden Lichtquelle beleuchtet werden muss [PH04, Kap. 1].

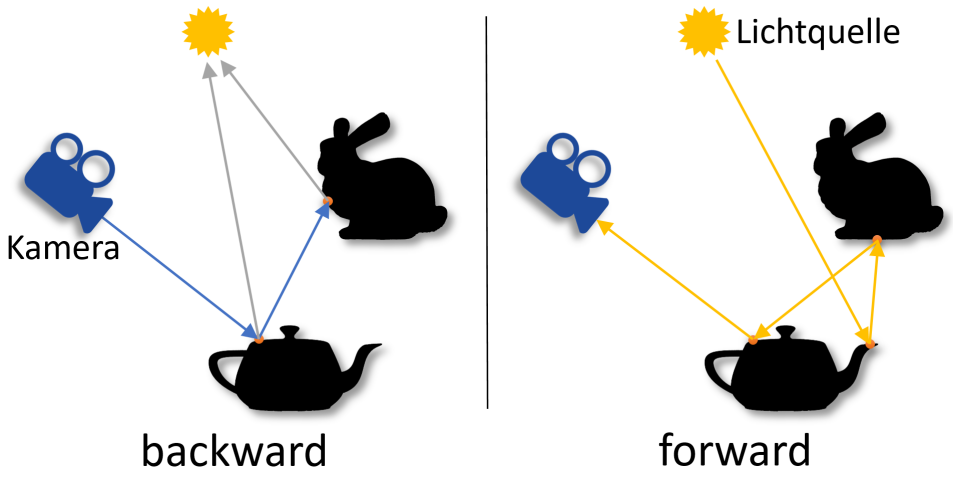

Abbildung 2: Vergleich von forward und backward Ray Tracing.
Links: Kamerapfad der Länge 2 (blau) mit zwei Schattenstrahlen (grau)
Rechts: Lichtpfad der Länge 4 (gelb)

Der Unterschied zwischen Pfadverfolgungsstrahlen und Schattenstrahlen ist, dass bei Pfadverfolgungsstrahlen (egal ob Licht- oder Kamerapfad) immer der *vorderste* Schnittpunkt eines Strahls mit der Szenengeometrie gesucht wird. Es werden also oft viele Schnittpunkte gefunden, unter denen der vorderste ermittelt werden muss. Für Schattenstrahlen ist hingegen nur zu entscheiden, ob es zwischen dem Strahlanfang (ein Schnittpunkt eines Kamera- oder Lichtpfades) und der Lichtquelle *irgendeinen* Schnittpunkt gibt (dieser muss nicht unbedingt der vorderste sein). Daraus folgt, dass Schattenstrahlen in der Regel etwas schneller berechnet bzw. verfolgt werden können als Pfadverfolgungsstrahlen.

3.1.1 Whitted Ray Tracing

Rekursives oder Whitted Ray Tracing (benannt nach Turner Whitted) führt eine rekursive Strahlverfolgung durch. Dafür werden an jedem Schnittpunkt zusätzlich zum Schattenfühler ein an der Oberfläche reflektierter und ein gebrochener Strahl weiterverfolgt (Abbildung 3 zeigt dies für den ersten Schnittpunkt). Dies wird rekursiv bis zu n mal wiederholt, wodurch ein Strahlenbaum der Tiefe n entsteht [PH04, Kap. 1]. Dieses Verfahren kann neben Schatten und einfachen diffusen Materialien auch spiegelnde und transparente Materialien darstellen. Allerdings ist es nicht in der Lage indirekte Beleuchtung bzw. diffuse Reflexionen zu berechnen.

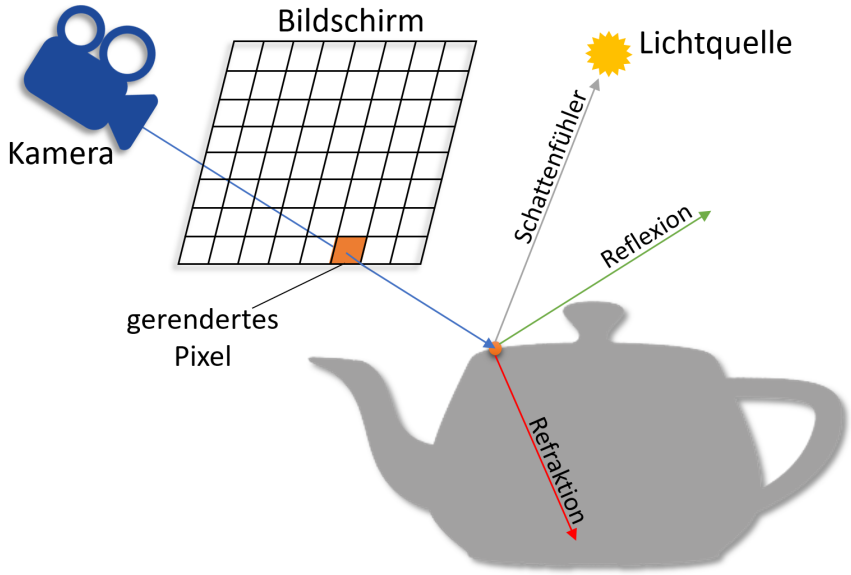

Abbildung 3: Kamera-Bildschirm-Modell und Strahlaufteilung beim Whitted Ray Tracing

3.2 Monte-Carlo-Methoden

Bei Monte-Carlo-Methoden spielt der Zufall eine sehr große Rolle. Das Verfahren ist daher auch nach dem gleichnamigen vom Glücksspiel geprägten Stadtteil „Monte Carlo" in Monaco benannt.

In der Mathematik und besonders auch in der Computergrafik gibt es häufig Gleichungen oder Funktionen, die Integrale beinhalten, welche nicht analytisch (z.B. durch Bilden der Stammfunktion) lösbar sind. Multidimensionale Integrale wie die Rendering-Gleichung bzw. die darin enthaltene BRDF eines Materials sind in der Regel solche nicht-trivialen Integralgleichungen. Solche Integrale können jedoch mit numerischen Methoden wie dem *Monte Carlo Estimator* gelöst werden. Beim Path Tracing wird dieses Verfahren für die Generierung der diffus reflektierten Strahlen verwendet.

3.2.1 Wahrscheinlichkeitsdichtefunktion (PDF)

Die Wahrscheinlichkeitsdichtefunktion (auch *PDF* von ***P**robability **D**ensity **F**unction*) beschreibt die Verteilung von Zufallsvariablen. Die Funktion gibt dabei die relative Wahrscheinlichkeit an, mit der eine Zufallszahl X_i einen bestimmten Wert a annimmt [PH04, Kap. 14]. Bei uniform verteilten Zufallszahlen ist die PDF konstant.

3.2.2 Monte-Carlo Estimator

Der Monte-Carlo Estimator ist das eigentliche mathematische Konstrukt zum numerischen Lösen eines Integrals. Die zugrundeliegende Idee ist es, das Integral an N zufällig gewählten Stellen „abzutasten". Anschließend wird der Mittelwert aus allen abgetasteten Werten gebildet, welcher den tatsächlichen Wert des Integrals schätzt bzw. annähert (5). Der Erwartungswert (E) des Monte-Carlo Estimators (F_N) entspricht somit dem Integral [PH04, Kap. 14].

$$\int_a^b f(x)dx = E[F_N] \quad mit \quad F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{5}$$

X_i ist dabei eine Zufallszahl aus dem Intervall $[a, b]$ mit der PDF $p(X_i)$. $f(X_i)$ ist die zu integrierende Funktion an der Stelle der Zufallszahl.

Das Ergebnis entspricht aufgrund des Einflusses von Zufallszahlen nicht unbedingt exakt dem eigentlichen Wert des Integrals. Bildet man jedoch den Mittelwert aus sehr vielen „Abtastungen" bzw. *Samples* (also wählt N sehr groß), so nähert man sich immer präziser an das tatsächliche Ergebnis an. Dieses Prinzip wird auch das „Gesetz der großen Zahlen" genannt: Je größer die Zahl N an Einzelergebnissen ist, desto genauer ist das Gesamt- bzw. Endergebnis. Wie schnell sich der Monte-Carlo Estimator an das Integral annähert, wird auch als *Konvergenz* bezeichnet.

3.2.3 Importance Sampling

Importance Sampling ist eine Technik, die verwendet wird um die Varianz des Monte Carlo Estimators zu verringern bzw. dessen Konvergenz zu verbessern. Um das zu erreichen wird versucht, die PDF möglichst exakt an die zu integrierende Funktion anzupassen, sodass beide Funktionen eine möglichst ähnliche Form haben [PH04, Kap. 15]. Folglich muss nun der abgetastete Wert nicht mehr mit der PDF gewichtet werden, da dies schon durch die Verteilung der Zufallszahlen geschehen ist. Ist die zu integrierende Funktion beispielsweise eine BRDF, so wird versucht, die PDF ebenfalls entsprechend dieser BRDF zu wählen. Insgesamt werden also die relevanteren Stellen in der zu integrierenden Funktion mit einer höheren Wahrscheinlichkeit abgetastet als unwichtige, wodurch eine deutlich schnellere Konvergenz und ein genaueres Ergebnis erreicht werden.

3.2.4 Russisches Roulette

Russisches Roulette ist ein sehr einfaches Verfahren, welches in vielen auf Monte-Carlo-Methoden basierenden Algorithmen zu Einsatz kommt. Es beschreibt lediglich die zufällige Auswahl einer von mehreren Möglichkeiten [PH04, Kap. 15]. Im Allgemeinen können das verschiedene Algorithmen, Strategien oder „Handlungsweisen" zur Ausführung eines Programms sein. Für das Path Tracing bedeutet das konkret die zufällige Auswahl des weiter zu verfolgenden Strahls.

3.3 Path Tracing

Beim Path Tracing wird anders als beim Ray Tracing nur *ein* Kamera- bzw. Lichtpfad (daher die Bezeichnung) pro Durchgang und Pixel berechnet. Ein Strahl wird also nie in mehrere Strahlen aufgeteilt und es entsteht kein verzweigter Strahlenbaum. Stattdessen wird an jedem Schnittpunkt abhängig vom Material durch russisches Roulette entschieden, welcher Pfad weiterverfolgt wird. Der Strahl wird an der Oberfläche also *entweder* diffus reflektiert (zB. mit Phong) *oder* gespiegelt *oder* gebrochen [PH04, Kap. 16].

Im gespiegelten oder gebrochenen Fall kann die Richtung des nächsten Strahls direkt eindeutig bestimmt werden. Im diffusen Fall wird hingegen eine zufällige Richtung gewählt. Die neue Richtung orientiert sich dabei entsprechend dem oben beschriebenen *Importance Sampling* an der BRDF des Materials. Je näher die Verteilung der generierten Strahlen an der BRDF des Materials liegt, desto schneller konvergiert das Bild. Schattenstrahlen werden nach wie vor im diffusen Fall für den Schattentest verwendet.

Der Algorithmus terminiert, sobald für den Strahl kein Schnittpunkt gefunden wurde (bzw. dieser die Szene verlassen hat) oder eine maximale Anzahl an Schnittpunkten gefunden wurde. Alternativ kann auch hier russisches Roulette verwendet werden, um zu entscheiden, ob ein weiterer Strahl generiert werden soll oder nicht.

Durch den großen Einfluss von Zufallswerten und aufgrund der Eigenschaften von Monte-Carlo-Methoden ist die Varianz des Ergebnisbildes nach dem ersten Durchgang (*Sample*) noch sehr hoch, was sich in starkem Bildrauschen bemerkbar macht. Wie bei Monte-Carlo-Methoden beschrieben, ist also eine mehrfache Ausführung des Path Tracing Algorithmus notwendig, um ein rauschfreies Bild zu erhalten, da erst dann das Bild gegen das erwartete Bild konvergiert [VL11]. Je nach Szene können dafür schon etwa 100 Samples ausreichen, in anderen Fällen sind mehrere tausend Samples nötig. Ein Sample wird im Ergebnisbild mit $\frac{1}{Gesamtanzahl\ Samples}$ gewichtet.

Dadurch, dass immer nur ein Pfad verfolgt wird, benötigt das Path Tracing pro Durchgang bzw. Sample deutlich weniger Rechenzeit als das Ray Tracing. Allerdings sind anders als beim Ray Tracing für eine gute Qualität des Ergebnisbildes viele Iterationen nötig. Ein sehr großer Vorteil des Path Tracings gegenüber dem standardmäßigen Ray Tracing ist, dass es in der Lage ist, durch die zufällige Verfolgung von diffus reflektierten Strahlen die globale Beleuchtung zu simulieren

und somit Effekte wie beispielsweise *Color Bleeding* darzustellen.

3.4 Abwandlungen von Path Tracing

Neben dem standardmäßigen Path Tracing Algorithmus existieren viele verwandte Verfahren, die auf Ray bzw. Path Tracing basieren, aber zusätzliche Techniken anwenden, um die Konvergenz zu beschleunigen oder die Bildqualität zu verbessern. Im Folgenden werden einige dieser Techniken kurz erläutert.

3.4.1 Stochastisches Ray Tracing

Stochastisches Ray Tracing vereint Whitted Ray Tracing und Path Tracing. Die Strahlverfolgung wird gemäß dem rekursiven Ray Tracing Algorithmus durchgeführt. Zusätzlich zum spiegelnden und gebrochenen Strahl werden aber an jedem Schnittpunkt noch mehrere diffus reflektierte Strahlen erzeugt, welche nach dem Prinzip des Path Tracings generiert und weiterverfolgt werden. Mit diesem Verfahren wird das Ray Tracing zwar aufgrund der stärkeren Verzweigung des Strahlenbaums noch aufwändiger, jedoch ist es damit möglich auch indirekte bzw. globale Beleuchtung zu berechnen.

3.4.2 Bidirektionales Path Tracing

Bidirektionales Path Tracing ist eine Kombination aus forward und backward Path Tracing und besonders für komplizierte Lichteffekte wie Lichtbündelungen (*Kaustiken*) von großem Nutzen. Die Idee hierbei ist, zuerst ein oder mehrere forward Path Tracing/s (also ausgehend von der/den Lichtquelle/n) durchzuführen und alle dabei gefundenen Schnittpunkte und deren Beleuchtungswerte bzw. -farben zwischenzuspeichern. Beim anschließenden backward Path Tracing (von der Kamera aus) wird dann jeder Schnittpunkt mit allen Lichtquellen *und* den gefundenen Schnittpunkten des Licht-Pfades beleuchtet [PH04, Kap. 16]. Abbildung 4 veranschaulicht das Verbinden der beiden Pfade.

Aufgrund der weitaus höheren Anzahl an zu testenden Schattenstrahlen benötigt das bidirektionale Path Tracing deutlich länger für die Berechnung eines Samples. Der große Vorteil liegt zum einen in der schnelleren Konvergenz des Bildes und zum anderen in der Möglichkeit, Kaustiken darstellen zu können. Beide Eigenschaften kommen zustande, da die tatsächliche Lichtstreuung wesentlich exakter berechnet wird als beim backward Path Tracing, ohne dabei derart drastische Performance-Einbußen wie beim normalen forward Path Tracing in Kauf nehmen zu müssen.

3.4.3 Photon Mapping

Photon Mapping basiert auf der Idee des *Particle Tracings* und verfolgt einen ähnlichen Ansatz wie das bidirektionale Path Tracing. Zuerst werden von einer Lichtquelle aus sehr viele (oftmals mehrere hunderttausend bis Millionen) Photonen in

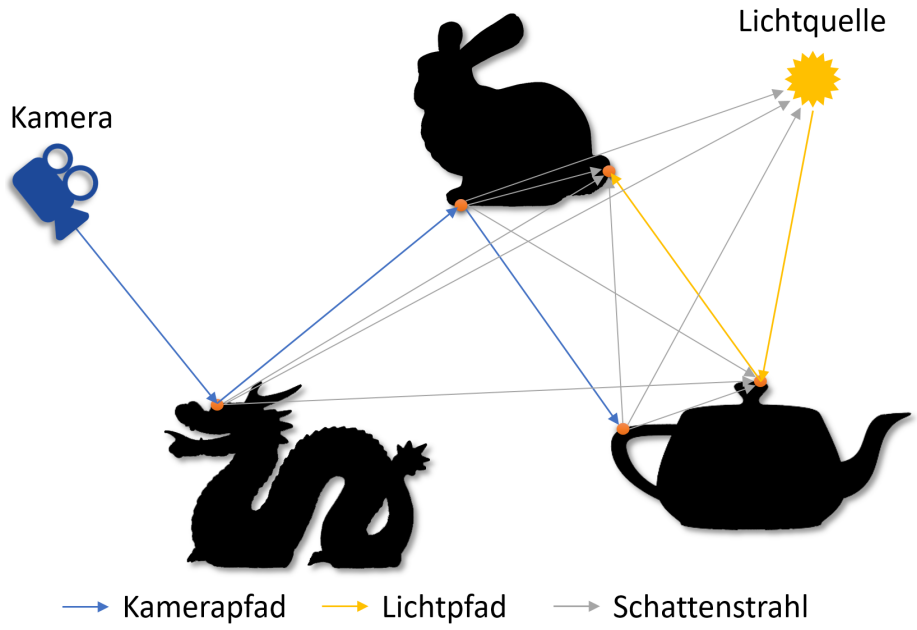

Abbildung 4: Verbindung des Kamera- und Licht-Pfades beim bidirektionalen Path Tracing

die virtuelle Szene geschickt. Jedes Photon wird dann als Partikel aufgefasst und verfolgt. Trifft ein Photon auf eine spiegelnde oder transparente Oberfläche, ändert es lediglich seine „Bewegungsrichtung". Trifft es hingegen auf eine diffuse Fläche, so wird das Photon (bzw. dessen Energie) ganz oder teilweise von der Oberfläche absorbiert. In diesem Fall wird das Photon in der Photon-Map gespeichert [PH04, Kap. 16]. Anschließend wird Ray Tracing oder Path Tracing zum Rendern der Szene durchgeführt, wobei jeder Schnittpunkt jetzt nur durch Interpolation der in einem bestimmten Umkreis liegenden Photonen aus der Photon-Map beleuchtet wird. Mit diesem Verfahren können nahezu alle optischen Effekte (u.a. auch Kaustiken) berechnet werden.

4 Line Space

4.1 Entstehung

Der Begriff „Line Space" wurde in der Computergrafik erstmals 1997 von Drettakis *et al.* in [DS97] verwendet, allerdings lediglich als Technik zum Aktualisieren von globaler Beleuchtung. K. Keul, S. Müller und P. Lemke präsentierten 2016 in [KML16] eine neue Definition des Line Spaces, welche eine allgemein anwendbare Beschleunigungsdatenstruktur für die Schnittpunktsuche zwischen Strahlen und 3D-Objekten beschreibt. Dieses Verfahren wurde seitdem sowohl von den Autoren, als auch von N. Klee in [Kle16], sowie in dieser Arbeit weiterentwickelt.

4.2 Aufbau

Der Line Space unterteilt einen beliebigen dreidimensionalen quaderförmigen Raum in sog. *Shafts*. Jede Seite des Quaders bzw. Voxels wird dafür in $N \times N$ *Patches* geteilt, wobei N die Auflösung des Line Spaces bezeichnet. Dies ergibt insgesamt $6 \times N^2$ Patches. Jedes Patch wird mit jedem anderen Patch des Voxels zu einem Shaft verbunden, was eine Gesamtanzahl von $36 \times N^4$ Shafts pro Voxel ergibt [KML16].

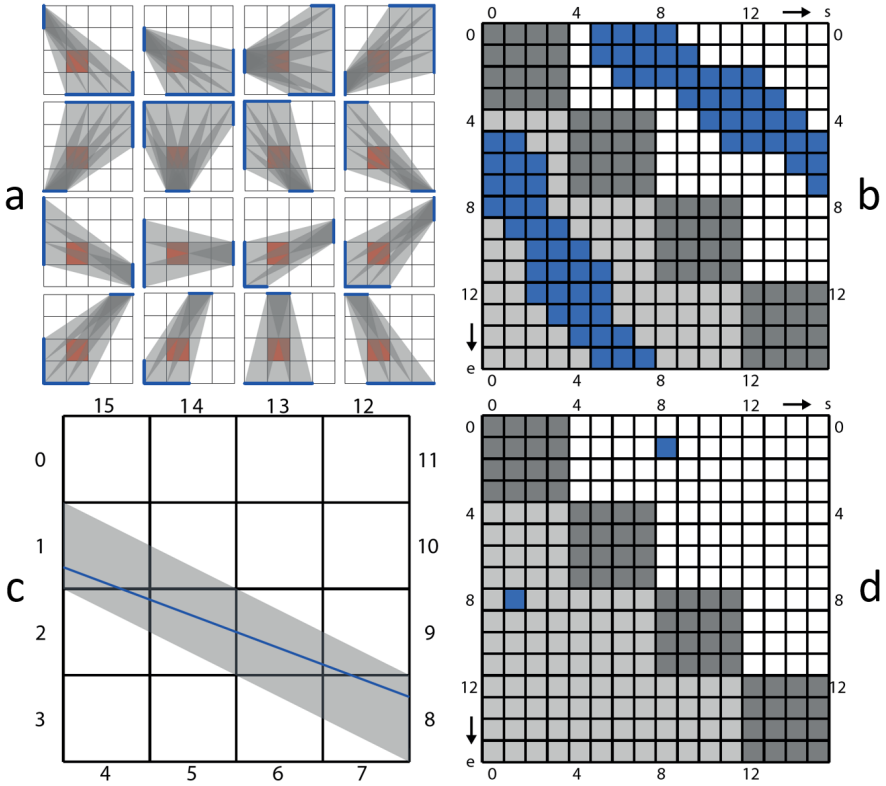

Abbildung 5: 2D-Line Space mit N = 4 aus [KML16].
a: Alle gefüllten Shafts je Start-Patch (rot = Objekt)
b: Der resultierende Line Space aus (a)
c: Ein Strahl durch das „2D-Voxel" und der zugehörige Shaft
d: Der Shaft aus (c) im Line Space.
Blau gefärbte Einträge in Line Space stehen für einen gefüllten Shaft; die grauen Flächen im Line Space weisen auf eine Redundanz hin

Abbildung 5 verdeutlicht das Prinzip für den zweidimensionalen Fall, der 3D-Fall funktioniert analog. Auffällig ist, dass zum einen eine Spiegelsymmetrie an der Diagonalen des Line Spaces existiert, da ein Shaft jeweils aus zwei Richtungen betrachtet wird. Das bedeutet, dass aufgrund dieser Redundanz nur eine Hälfte des Line Spaces gespeichert werden muss. Zum anderen liegen auf der Diagonalen immer $N \times N$ große „Blöcke", welche nie gefüllt sind, da sich die entsprechenden

Patches und somit die dazugehörigen Shafts alle auf der gleichen Seite (des Voxels) befinden und somit keine Objekte enthalten können. Die tatsächliche Anzahl an relevanten Shafts pro Voxel kann dadurch auf $15 \times N^4$ reduziert werden [KML16]. Der Faktor 15 entsteht dadurch, dass jede Seite des Voxels einmal mit jeder anderen verbunden wird (also $5 + 4 + 3 + 2 + 1 = 15$).

4.2.1 Speichern der Daten

Für jeden Shaft kann anschließend durch *Clipping* ermittelt werden, ob bzw. welche Geometrie vom Shaft geschnitten wird. Abbildung 6 zeigt dies an einem beispielhaften Line Space mit $N = 4$. Ausgehend von den gefundenen Objekten können nun beliebige Informationen in den Line Space geschrieben werden. Das können sowohl einfache binäre Beleuchtungsinformationen für Schatten, als auch komplexere Daten wie beispielsweise Kandidatenlisten sein.

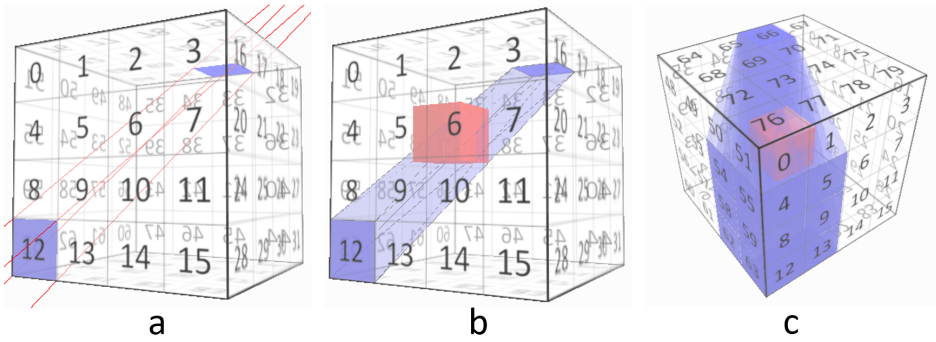

Abbildung 6: 3D-Line Space mit N = 4.
a: Shaft zwischen Patch 12 und 66
b: Der rote Würfel wird vom Shaft geschnitten
c: Alle von Patch 66 ausgehenden und gefüllten Shafts

4.3 Anwendung

Möchte man die zu einem Strahl gehörenden Informationen aus dem Line Space auslesen, so wird abhängig von der Strahlrichtung ein Start- und ein End-Patch im entsprechenden Line Space und somit auch der Shaft berechnet. Vorteilhaft ist, dass bei einem leeren Shaft direkt feststeht, dass der Strahl kein Objekt schneiden wird und somit keine unnötigen Schnittpunkttests nötig sind. Weiterhin müssen innerhalb eines Shafts weitaus weniger Objekte untersucht werden, als innerhalb eines ganzen Voxels oder sogar der gesamten Szene. Diese beiden Eigenschaften führen zu einer deutlichen Beschleunigung der Schnittpunktberechnung bzw. der Strahlverfolgung.

4.4 Alternative Datenstrukturen

Im Folgenden werden einige gängige alternative Datenstrukturen zum Line Space vorgestellt. Auch wenn diese Datenstrukturen alle eigenständig zur Beschleunigung von Schnittpunkttests dienen, könnten sie alle auf verschiedene Art und Weise durch den Line Space erweitert werden, um die Anzahl der letztendlich zu testenden Objekte weiter zu verringern.

4.4.1 Voxel Grid

Das Voxel Grid ist eine der einfachsten Beschleunigungsdatenstrukturen für Ray bzw. Path Tracing, weshalb es als „Basis"-Datenstruktur in dieser Arbeit verwendet wurde. Die Grundidee ist es, die gesamte Szene (bzw. deren Bounding Box) in $m \times n \times l$ gleich große Würfel oder Quader (*Voxel, von „volume element"*) zu unterteilen. Jedem Voxel werden dann die darin enthaltenen Objekte oder „Kandidaten" (z.B. Dreiecke) zugeordnet und gespeichert. [PH04, Kap. 4.3].

Die Traversierung der Voxel bei der Schnittpunktsuche für die Strahlverfolgung wird mit dem 3D-DDA-Algorithmus (*Digital Differential Analyzer*) [FI85] durchgeführt. Dafür wird für einen gegebenen Strahl zuerst das Start-Voxel bestimmt und dann schrittweise entlang des Strahls über die vom Strahl durchdrungenen Voxel iteriert [AW+87]. Für jedes dabei betrachtete Voxel wird dann ein Schnittpunkttest zwischen Strahl und allen enthaltenen Kandidaten durchgeführt. Es muss also lediglich ein kleiner Teil der gesamten Szenengeometrie mit dem Strahl geschnitten werden, anstatt alle in der Szene enthaltenen Objekte.

Diese Datenstruktur kann sehr schnell und einfach generiert werden, ist jedoch bei der Traversierung in vielen Fällen nicht ausreichend performant, was im Evaluationsteil dieser Arbeit gezeigt wird. Dieser Nachteil entsteht dadurch, dass bei kleinen Voxel-Gitter-Auflösungen einige Voxel sehr viele Objekte enthalten, was zu mehr Strahl-Objekt-Schnittpunkttests führt. Ist die Voxel-Gitter Auflösung jedoch hoch, so sind zwar innerhalb der einzelnen Voxel weniger Objekte enthalten, jedoch müssen weitaus mehr Voxel überprüft werden.

4.4.2 N-Tree und Octree

Der N-Tree (auch *Recursive Grid* oder *Adaptive Grid*) ist eine baumförmige Datenstruktur und basiert auf einer ähnlichen Grundidee wie das Voxel Grid. Dabei wird die Szene rekursiv in N^3 Voxel unterteilt. Es wird jedoch nicht unbedingt jedes Voxel weiter unterteilt, sondern nur solche, in denen eine hohe Geometriedichte vorliegt [KS97]. Die Kandidaten-Objekte werden nur in den Blatt-Knoten des entstehenden Voxel-Baumes gespeichert. Der Baum kann dabei *top-down* oder *bottom-up* erstellt werden.

Der Octree ist ein Spezialfall des N-Trees. Hier wird $N = 2$ gesetzt, wodurch ein Voxel (wenn nötig) entlang der drei Koordinatenachsen jeweils in der Mitte geteilt wird, was in acht Sub-Voxel resultiert. Die maximale Rekursionstiefe muss

hier in der Regel höher sein als bei N-Trees mit $N > 2$, da die Unterteilung pro Rekursionsschritt beim Octree kleiner ist.

Bei der Traversierung der Datenstruktur wird nun zuerst abhängig von der Strahlrichtung der zu untersuchende Blatt-Knoten gesucht (Tiefensuche) und erst dann ein Schnittpunkttest mit den Kandidaten durchgeführt. Insgesamt kann eine dynamischere Voxel-Auflösung als beim Voxel Grid erreicht werden und ein einzelnes Voxel enthält deutlich weniger Geometrie, was eine schnellere Traversierung ermöglicht. Allerdings sind die Generierung der Datenstruktur und der Abstieg in tiefere Rekursionsebenen des Baumes bei der Traversierung auch komplexer und dadurch aufwändiger [KS97].

4.4.3 Bounding Volume Hierarchie

Bounding Volume Hierarchien (kurz *BVHs*) basieren nicht wie die bisher beschriebenen Datenstrukturen auf der Unterteilung der Szene in Voxel, sondern auf der Generierung von hierarchischen Bounding Volumes (kurz *BVs*) um Teilmengen der Szenengeometrie und andere BVs herum [GS87] (siehe Abbildung 7). Bei der Traversierung wird dann unter Verwendung einer *Priority Queue* eine Tiefensuche auf der BVH durchgeführt, um dann Schnittpunkte zwischen Strahl und Szenengeometrie innerhalb eines BVs zu berechnen [PJH16, Kap. 4.3]. Die BVH als Beschleunigungsdatenstruktur ist beim Ray bzw. Path Tracing bisher die am weitesten verbreitete Technik, da sie sehr gute Ergebnisse bezüglich Generierung und Traversierung liefert.

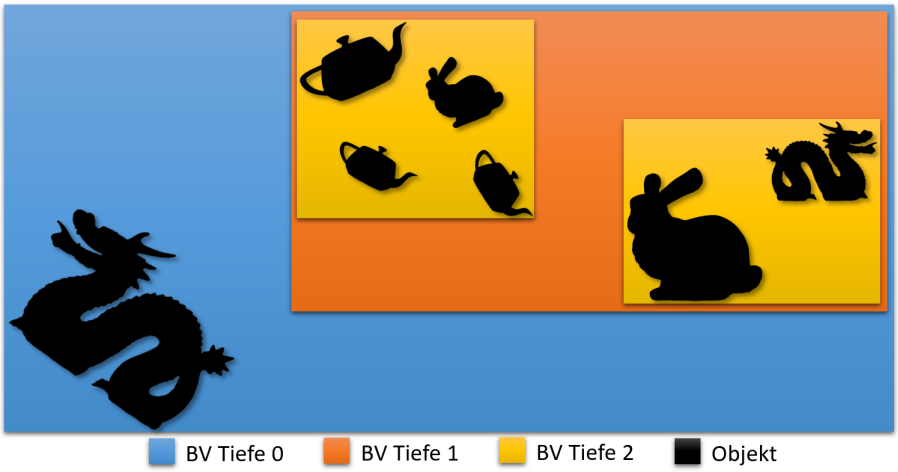

Abbildung 7: Beispielhafte Bounding Volume Hierarchie mit Tiefe 3

Teil II
Implementierung

In diesem Teil werden die im Rahmen dieser Arbeit implementierten Algorithmen erklärt. Alle beschriebenen Verfahren bzw. Programme sind auch als Code in dem zu dieser Arbeit gehörenden C++ Projekt enthalten[1].

5 GPU Programmierung

5.1 Pipeline

Die Standard-Pipeline, die in dieser Arbeit verwendet wird, besteht aus drei großen Bausteinen: Dem Erstellen der Datenstrukturen, dem Rendern der Szene bzw. dem Path Tracing und dem abschließenden Anzeigen des gerenderten Bildes. Je nach Verfahren kommen allerdings noch zusätzliche Render-Passes hinzu, wie zum Beispiel das Rendern eines G-Buffers. Abbildung 8 veranschaulicht diese Pipeline. Die Datenstrukturen werden demnach nur einmal erstellt und wiederverwendet. Nur bei einem Szenenwechsel müssen sie neu generiert werden.

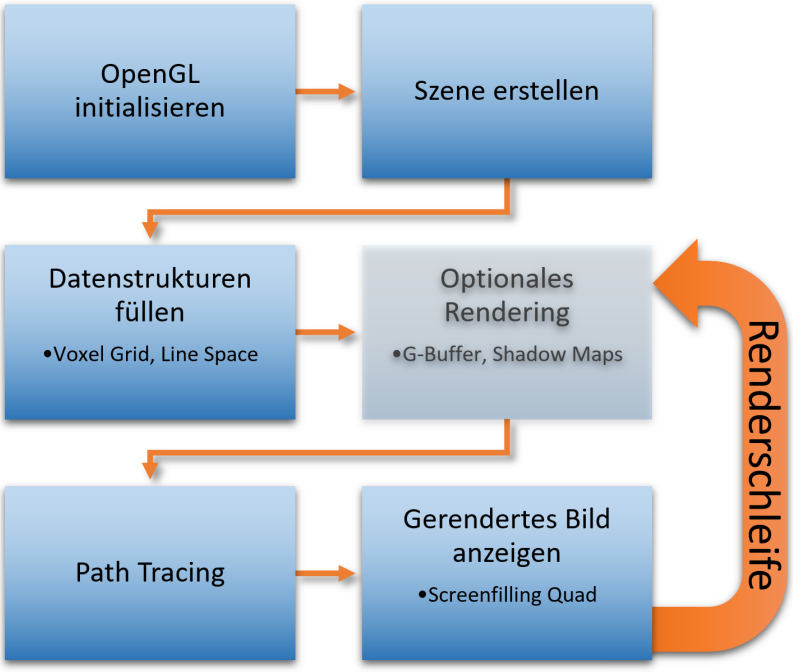

Abbildung 8: Path Tracing Pipeline

[1] Der Source-Code kann per E-Mail angefragt werden: fschroeder@uni-koblenz.de

5.2 Compute Shader

In dieser Arbeit wurden für fast alle Berechnungen *Compute Shader* verwendet. Diese sind bezüglich der verfügbaren Funktionen sehr ähnlich zu *Fragment Shadern*. Der größte Unterschied besteht darin, dass für einen Fragment Shader immer auch ein (wenngleich sehr schlichter) *Vertex Shader* vorhanden sein muss und die Berechnung dann durch die gesamte OpenGL-Rendering-Pipeline läuft. Da Compute Shader die Berechnungen direkt, also ohne den gesamten Rasterisierungs-Prozess starten können und nicht an die Viewport-Auflösung gebunden sind, eignen sie sich deutlich besser für *General-Purpose Computing on Graphics Processing Units (GPGPU)*. Das bedeutet zum einen, dass dreidimensionale Datenstrukturen (wie der Line Space) intuitiver aufgebaut werden können, da Compute-Shader-Blocks (anders als bei Fragment Shadern) ein-, zwei- oder dreidimensional definiert werden können. Zum anderen ergibt sich durch die bessere Aufteilbarkeit der Berechnungen auf die verfügbaren Prozessoren auf der GPU eine deutlich bessere Performance für das Path Tracing.

Diese Aufteilung geschieht durch das Definieren von lokalen und globalen „Arbeitsgruppen" (*local work groups* und *global work groups*). Während die globale Gruppengröße dynamisch zur Laufzeit durch den „Render"-Befehl

```
glDispatchCompute(num_groups_x, num_groups_y, num_groups_z)
```

bestimmt werden kann, muss die lokale Gruppengröße bereits zur Compile-Zeit des Shaders im Shader selbst durch den Layout-Qualifier

```
layout (local_size_x = 32, local_size_y = 1, local_size_z = 1) in
```

festgelegt werden.

Hierbei gilt: Je näher die Anzahl an lokalen Shader-Einheiten an der *Warp*-Größe (also der Anzahl an parallel abgearbeiteten Instruktionen, in der Regel 32) der GPU ist, umso performanter ist der Compute Shader. Für das Path Tracing bedeutet das konkret, dass sich insgesamt

$$\frac{Bildbreite}{local\_size\_x} \times \frac{Bildhöhe}{local\_size\_y}$$

globale Gruppen mit jeweils $local\_size\_x \times local\_size\_y$ lokalen Gruppen ergeben. Im Shader kann die ID (bzw. „Position") der Shader-Einheit dann eindeutig über die Built-in-Variablen $gl\_LocalInvocationID$ (lokal) und $gl\_WorkGroupID$ (global) bestimmt werden. Die $local\_size$ sollte je nach System und verfügbaren GPU-Prozessoren angepasst werden, um die schnellsten Berechnungszeiten zu erreichen.

5.3 Zufallszahlen auf der GPU

Wie bereits im Abschnitt 3 zu Path Tracing erklärt, ist ein essenzieller Bestandteil von Monte-Carlo-Methoden und somit auch von Path Tracing die Generierung von möglichst gut verteilten Zufallszahlen. Diese können aber aufgrund der

Funktionsweise einer GPU nicht direkt auf dieser erzeugt werden. Man verwendet daher sogenannte Pseudo-Zufallszahlen. Das sind Zufallszahlen, die eine gute, scheinbar zufällige [0,1]-Verteilung aufweisen, jedoch deterministisch berechnet werden. Dabei wird immer aus einer bereits generierten Zahl die nächste Zufallszahl berechnet.

Da diese Zufallszahlen auf reinen Berechnungen beruhen und nur eine begrenzte Anzahl an „Zufallszuständen" existiert, entsteht bei der Generierung früher oder später ein Zyklus (d.h. die generierten Zahlen wiederholen sich). Ist der Zyklus zu klein, werden sehr oft die gleichen Zufallszahlen erzeugt, was beim Path Tracing zur Bildung von Artefakten führen kann. Es ist also wichtig, ein Verfahren zu wählen, welches einen möglichst großen Zyklus *und* eine möglichst homogene Verteilung aufweist. In dieser Arbeit wurde dafür ein in [Ngu07, Kap. 37] vorgestellter *Pseudorandom Number Generator (PRNG)* verwendet, welcher auf Integer-Rechnungen und Bit-Operationen basiert und eine Kombination aus einem *Linear Congruential Generator (LCG)* und einem *Combined Tausworthe Generator (CTG)* ist.

5.3.1 Linear Congruential Generator

Der Linear Congruential Generator ist einer der einfachsten Zufallsgeneratoren. Er berechnet Zufallszahlen nach folgender Formel [Knu69] :

$$x_{n+1} = (a \cdot x + c) \bmod m \tag{6}$$

Dabei sind *a*, *c* und *m* Konstanten und *x* die vorherige Zufallszahl. Nach [PTVF92] kann aufgrund der 32 Bit Arithmetik mit unsigned Integer die Modulo-Rechnung ignoriert werden und man erhält:

Algorithmus 1: Linear Congruential Generator in GLSL

```
uint LCGStep(uint x, uint A, uint C) {
    return (A * x + C);
}
```

Vorteilhaft ist besonders die sehr einfache und somit schnelle Berechnung der Zahlen. Leider verfügt dieser Zufallsgenerator mit einem Zyklus von höchstens 2^{32} (aufgrund der Verwendung eines 32 Bit Integers) aber weder über viele mögliche Zahlen, noch über eine gute Verteilung der generierten Zahlen. Er sollte daher nicht als alleinstehender Generator verwendet werden, ist aber durchaus nützlich, um andere zu verbessern.

5.3.2 Combined Tausworthe Generator

Der Combined Tausworthe Generator baut auf einem ähnlichen Prinzip wie der weit verbreitete *Mersenne Twister* [MN98] auf, ist allerdings etwas vereinfacht und

somit performanter. Hierbei wird die vorherige Zufallszahl durch mehrfache Bit-Shifts («), XOR-Operationen (^) und eine AND-Operation (&) in eine neue Zahl überführt. [L'E96]

Algorithmus 2: Combined Tausworthe Generator in GLSL

```
uint CTGStep(uint x, int S1, int S2, int S3, uint M) {
    uint a = (((x << S1) ^ x) >> S2);
    return (((x & M) << S3) ^ a);
}
```

Dabei sind wieder *S1*, *S2*, *S3* und *M* Konstanten und *x* die vorherige Zufallszahl. Auch dieser Generator hat kurze Berechnungszeiten und liefert statistisch etwas bessere Verteilungen als der LCG. Zwar hat auch der CTG einen relativ kleinen Zyklus, jedoch kann dieser durch Kombination mehrerer CTGs stark erhöht werden.

5.3.3 Hybrider Zufallsgenerator

In dieser Arbeit wurde zur Generierung der Zufallszahlen eine in [Ngu07, Kap. 37] vorgeschlagene Kombination der beiden oben beschriebenen Generatoren verwendet. Dabei wird der aktuelle „Zufallszustand" in einer vier-kanaligen Textur gespeichert. Zur Generierung der nächsten Zufallszahl werden auf drei der Komponenten ein CTG und auf der vierten ein LCG angewendet. Die vier generierten Werte werden dann durch exklusives Oder (XOR) verknüpft und in einen Float-Wert zwischen null und eins umgewandelt, welcher als Zufallszahl zurückgegeben wird. Gleichzeitig bilden diese vier Werte den neuen Zufallszustand in der Textur.

Die Textur hat dabei die Auflösung des OpenGL-Viewports, sodass jeder Pixel seine „eigenen" Zufallszahlen hat. Um eine gute Anfangsverteilung der Zufallswerte zu gewährleisten, wird die Textur bei Programmstart einmalig mit Zufallszahlen der CPU gefüllt. Anschließend geschieht die gesamte Berechnung neuer Zufallszahlen auf der GPU.

Diese Kombination vereint die Vorteile der beiden Generatoren und beseitigt gleichzeitig ihre Nachteile. Somit können Zufallszahlen sehr schnell generiert werden, haben eine sehr gute [0,1]-Verteilung und wiederholen sich mit einem Zyklus von ca. 2^{121} nur sehr selten [Ngu07, Kap. 37].

6 GPU Path Tracer

Im Folgenden werden zuerst die für den im Rahmen dieser Arbeit implementierten GPU Path Tracer vorausgesetzten Konstrukte erklärt und damit anschließend dessen Funktionsweise erläutert.

6.1 Voxelisierung

In dieser Arbeit ist das Voxel-Gitter in einer 3D-Textur gespeichert. Jeder Eintrag in dieser Textur enthält dabei eine Referenz auf den für das entsprechende Voxel relevanten Teil in einer Kandidatenliste, welche wiederum als ein *Shader Storage Buffer Object (SSBO)* vorliegt.

Für die Voxelisierung wird eine konservative Rasterisierung [PF05, Kap. 42] der Szene durchgeführt. Die klassische Projektion eines 3D-Objekts auf den 2D-Viewport muss dafür leicht verändert werden. Dabei wird sichergestellt, dass für jedes Fragment, auf das ein Teil eines Polygons projiziert wurde, ein Fragment Shader aufgerufen wird. Dies ist sehr wichtig, da ansonsten Teile eines Polygons nicht rasterisiert werden würden, und es dadurch zu Löchern im resultierenden Voxel-Gitter kommen könnte. Die Konservativität der Rasterisierung wird hierbei durch einen *Geometry Shader* erzeugt, indem die Eckpunkte eines Dreiecks um einen berechneten Wert verschoben werden (*Dilatation*), damit alle relevanten Fragmente rasterisiert werden. Zusätzlich wird durch die Auswahl verschiedener Viewports und *Swizzling* (Vertauschen der Koordinaten) der Vertex-Positionen das Dreieck je nach dessen dominanter Normalenachse auf die XY-, XZ- oder YZ-Ebene projiziert, um die projizierte Fläche des Dreiecks zu maximieren. Abbildung 9 zeigt einen Vergleich zwischen konservativer und nicht konservativer Rasterisierung [ZCEP07].

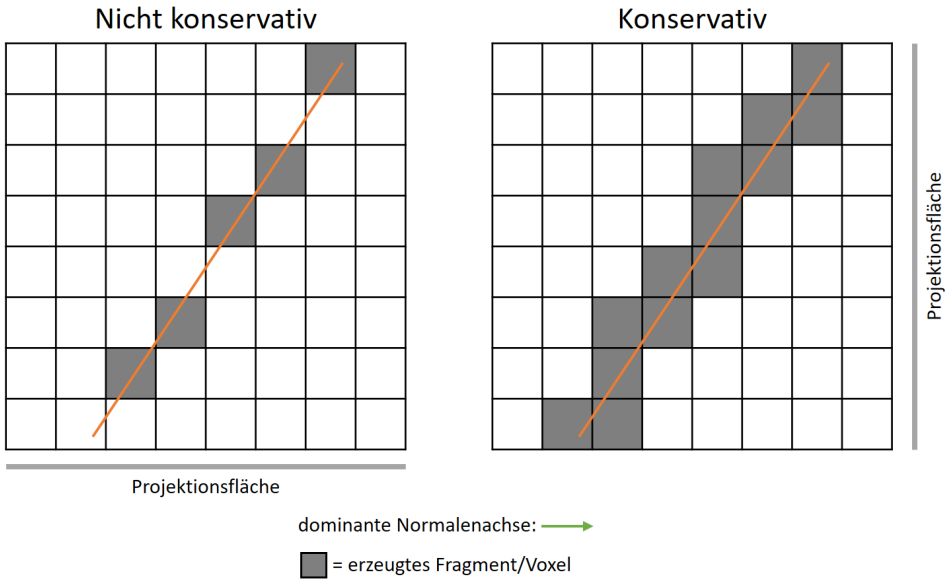

Abbildung 9: Konservative und nicht konservative Rasterisierung in 2D (3D analog)

Um die fertige Voxelisierung der Szene zu erhalten, sind zwei Durchgänge nötig. Im ersten Durchgang wird für jedes Voxel gezählt, wie viele Kandidaten es enthält. Mit diesen Informationen werden dann die Präfixsummen [Ngu07, Kap. 39] (siehe Abb. 10, 3D analog) für die Voxel berechnet und der nötige Speicherplatz

für die Kandidatenliste allokiert. Im zweiten Durchgang werden dann die Objekte (in Form ihrer Objekt-ID) an den entsprechenden Stellen in die Kandidatenliste eingefügt (Abbildung 11).

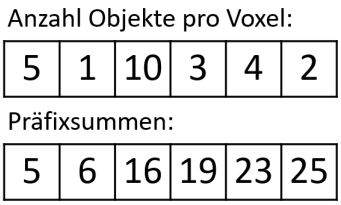

Abbildung 10: 1D-Beispiel für Präfixsummenbildung, das Verfahren ist analog auf größere Voxelmengen und den dreidimensionalen Fall übertragbar

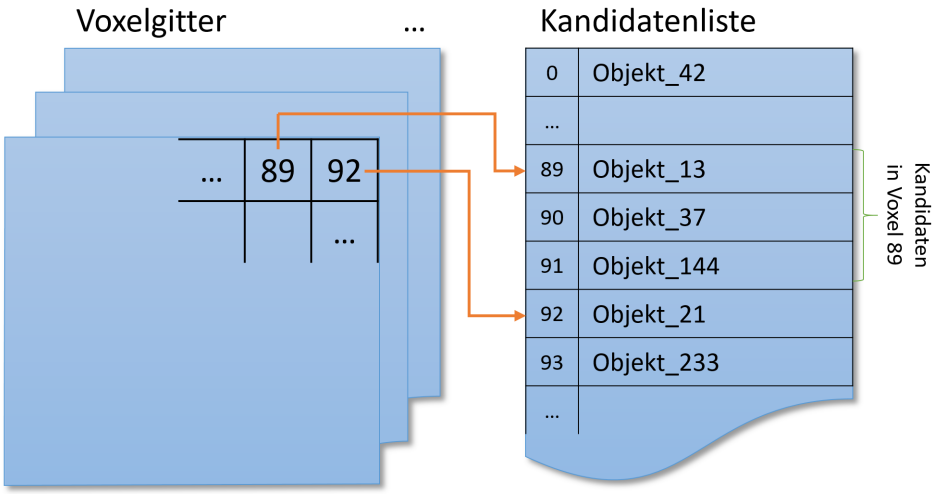

Abbildung 11: Aufbau der Voxel-Datenstruktur

6.2 Traversierung der Voxel

Die Traversierung wurde nach dem in 4.4.1 beschriebenen Algorithmus implementiert. Wird ein Schnittpunkt gefunden, müssen keine weiteren Voxel untersucht werden (es wird *true* zurückgegeben). Wird die Bounding Box der Szene beim Iterieren über die Voxel verlassen oder eine maximale Strahlverfolgungslänge erreicht, muss ebenfalls abgebrochen werden, da dann kein Schnittpunkt mehr gefunden werden kann bzw. soll (es wird *false* zurückgegeben).

6.3 Flächige Lichtquellen

Der Path Tracer arbeitet mit flächigen Lichtquellen, da diese visuell bessere Ergebnisse bringen und physikalisch plausibler sind. Gleichzeitig sind sie beim Path Tracing aber kaum aufwändiger zu berechnen als Punktlichtquellen oder Spot-Lights.

Eine Lichtquelle ist also neben ihrer Position (P), Farbe (C) und Intensität (I) auch durch eine geometrische Form repräsentiert. In dieser Arbeit werden nur planare Lichtquellen unterstützt, für die Geometrie ist also zusätzlich die Angabe der Größe und der Normale der Lichtquelle nötig.

Soll nun ein Schattenfühler von einem Punkt (S) aus in Richtung der Lichtquelle abgetastet werden, so wird zunächst eine zufällige Position (P_{rnd}) auf der Lichtquelle berechnet und als Ziel des Strahls gesetzt. Ist die Lichtquelle vom betrachteten Punkt aus nicht verdeckt (es gab also keine Überschneidung mit dem Schattenfühler), kann ein Beleuchtungswert (L) berechnet werden. Dieser ist abhängig vom Abstand (d) zwischen der Lichtquelle und dem zu beleuchtenden Punkt und durch Erweiterung des Phong-Beleuchtungsmodells [Pho75] folgendermaßen für flächige Lichtquellen definiert:

$$L = \frac{C * I * A * \cos(\theta_E) * \cos(\theta_R)}{\pi * d^2} \tag{7}$$

Dabei sind die Cosinus-Terme jeweils abhängig vom Winkel zwischen Strahl und Normale (n) der Lichtquelle (E) bzw. der zu beleuchtenden Fläche (R) und können mithilfe des Skalarprodukts berechnet werden. Abbildung 12 veranschaulicht diese Rechnung.

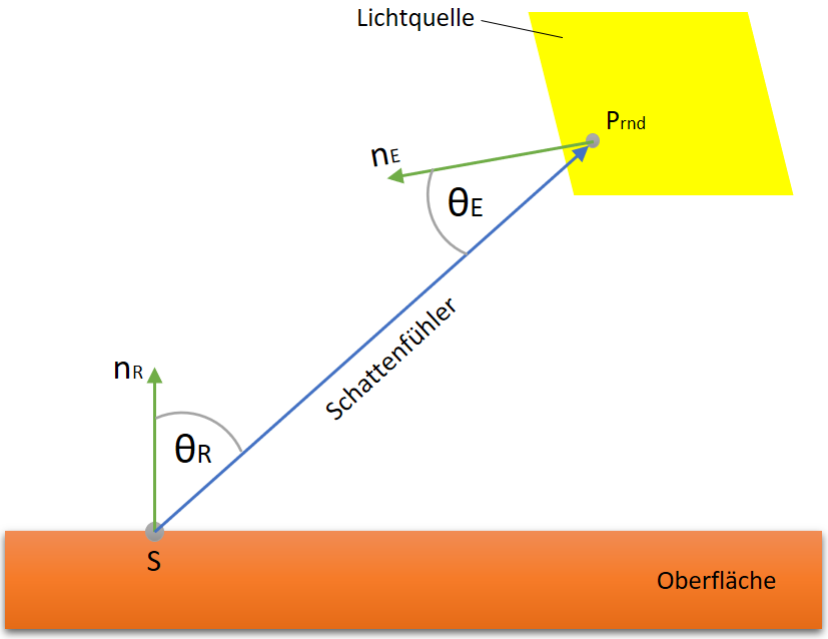

Abbildung 12: Winkel für die Beleuchtung

Der große Vorteil dieser Lichtquellen ist, dass mit ihnen weiche Schatten dargestellt werden können. Diese sind in der Realität auch nahezu überall zu beobachten, da reale Lichtquellen immer flächig sind. Ein weicher Schatten bedeutet, dass

zwischen der totalen Verschattung (*Umbra* oder Kernschatten) und den nicht verschatteten Stellen an bzw. um ein Objekt ein Übergangsbereich (*Penumbra* oder Halbschatten) existiert [Kle16] [BB84]. Dieser entsteht dadurch, dass in diesem Bereich die Lichtquelle nur zum Teil durch das Objekt verdeckt wird und somit ein Halbschatten zu sehen ist. Abbildung 13 verdeutlicht dieses Phänomen.

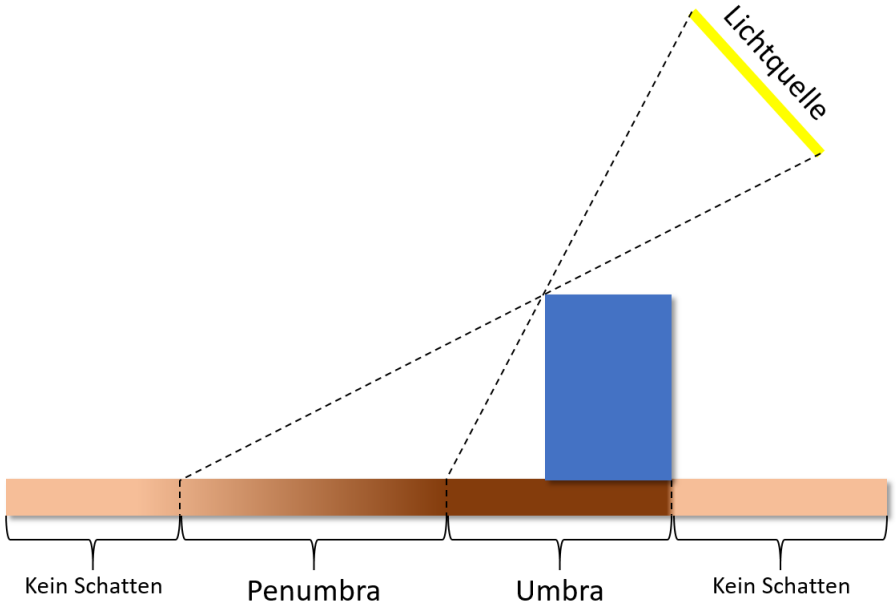

Abbildung 13: Entstehung von Halbschatten

6.4 Bausteine des Path Tracers

Der Kern des Path Tracers besteht neben den bisher vorgestellten Funktionen hauptsächlich aus den Methoden *Trace*, *Shade* und dem *Phong-Sampling*. Diese sind für das Verfolgen des (Kamera-) Pfades verantwortlich und bearbeiten die gefundenen Schnittpunkte der Strahlen mit der Szene.

6.4.1 Trace

Die Methode *Trace* startet zuerst die Berechnung des vordersten Schnittpunktes eines Strahls mit der Szene. Hierfür wird entweder das Voxel Grid oder der Line Space oder eine Kombination aus beiden verwendet. Wurde ein Schnittpunkt gefunden, so wird abhängig von den Materialeigenschaften des geschnittenen Objekts durch *russisches Roulette* entschieden, ob der reflektierte, gebrochene oder diffuse Strahl weiterverfolgt werden soll. Wird einer der ersten beiden Fälle ausgewählt, kann direkt ein eindeutiger Strahl für die nächste Iteration berechnet werden. Wenn jedoch der diffuse Fall eintritt, so muss erst der Schnittpunkt durch *Shade* beleuchtet werden, und anschließend ein neuer Strahl mithilfe des *Phong-Samplings*

generiert werden. Dies wird bis zu einer maximalen „Strahltiefe" (*Light-Bounces*) sequentiell wiederholt, wobei berechnete Farbwerte aufsummiert werden. Trifft ein Strahl direkt eine Lichtquelle oder gar kein Objekt (also den Hintergrund), kann die Verfolgung dieses Strahls direkt abgebrochen werden, da das Ergebnis dann unmittelbar vorliegt.

6.4.2 Shade

Die Methode *Shade* ist lediglich dafür verantwortlich, das Licht von allen Lichtquellen in der Szene „einzusammeln". Dafür wird eine Schleife über alle Lichtquellen ausgeführt, die entsprechenden Schattenfühler werden verfolgt und die berechneten Beleuchtungswerte werden addiert. Abschließend wird der Gesamt-Beleuchtungswert mit der Farbe des zu beleuchtenden Objekts multipliziert und man erhält den beleuchteten Farbwert für den entsprechenden Punkt.

6.4.3 Phong-Sampling

Die Methode *SampleDirection* führt abhängig von der Oberflächennormale, der Glanzzahl und der Richtung des eintreffenden Strahls das Phong-Sampling aus. Dabei wird ein neuer Strahl in eine zufällige Richtung generiert. Diese zufälligen Richtungen sind jedoch nur bei einer Glanzzahl von Null (perfekt diffuses Material) uniform über eine Halbkugel um die Normale verteilt. Je größer die Glanzzahl ist, desto stärker orientiert sich die Verteilung in Richtung des an der Normalen reflektierten eintreffenden Strahls. Dies entspricht dem im Phong-Modell beschriebenen Verhalten des Lichts [Pho75].

6.5 Inkrementelles Path Tracing

Beim Path Tracing ist ein Ergebnisbild aus einem einzigen Path Tracing-Durchgang meist zwar schnell berechnet, weist jedoch meistens eine sehr schlechte Qualität auf. Dies liegt vor allem am Rauschen, welches durch die vielen Zufallskomponenten verursacht wird. Wie bereits im theoretischen Teil zu Monte-Carlo-Methoden beschrieben, erhält man nach dem Gesetz der großen Zahlen erst durch mehrfache Ausführung ein gutes Ergebnis. Würde man jedoch einfach eine vorher festgelegte Anzahl an Renderdurchgängen ausführen und erst nach Beendigung *aller* Durchgänge das berechnete Bild anzeigen, wäre die Anwendung aufgrund der langen Berechnungszeit nicht mehr interaktiv bzw. echtzeitfähig.

Um dem entgegen zu wirken, wird immer nur *ein* Durchgang berechnet und dieser anschließend mit allen bisher berechneten Bildern nach folgender Formel vermischt:

$$C_{s+1} = \frac{s}{s+1} * C_s + \frac{1}{s+1} * C_{Trace} \tag{8}$$

Dabei ist s die Anzahl an bisher berechneten Bildern (*Samples*), C_s das gemischte Bild aller bisherigen Samples, C_{Trace} das im aktuellen Durchgang berechnete Bild

und C_{s+1} das neu gemischte Bild. Diese Technik wird auch *Progressive Refinement* genannt.

Der große Vorteil dieser Methode ist, dass nach jedem Durchgang ein Bild angezeigt werden kann. Das angezeigte Bild ist dadurch anfangs stark verrauscht, wird aber im Laufe der Zeit schnell immer weiter verfeinert, ohne dass der Nutzer lange auf ein Bild warten muss, wodurch die Anwendung interaktiv bzw. echtzeitfähig bleibt. Allerdings muss das inkrementelle Path Tracing bei jeder Änderung der Szene (Kamera, Lichtquelle oder Objekt) zurückgesetzt werden, da sich dann die Lichtsimulation ändert und neu berechnet werden muss.

7 Optimierung durch den Line Space

7.1 Generierung

Die Generierung des Line Spaces basiert auf der vorher durchgeführten Voxelisierung. Jedes Voxel hat also seinen eigenen Line Space. In der einfachsten Ausführung wird für jeden Shaft eine begrenzte Anzahl *k* der Kandidaten (also Dreiecke) gespeichert. Die verfügbaren Kandidaten werden dabei aufgeteilt, sodass ein „front"- und ein „back"-Buffer entsteht. In Algorithmus 3 ist die Generierung als Pseudo-Code dargestellt.

Algorithmus 3: Line Space Generierung in Pseudo-Code.

```
procedure GenerateLineSpace()
  for all objects o in Voxel v do
    for all shafts s in Line Space l do
      if ObjectShaftClipping(o, s) then
        Ray frontRay = midpoint ray of s
        Ray backRay = Invert(frontRay)
        if ObjectRayIntersection(o, frontRay) then
          AddObjectToFrontCandidates(o)
        end if
        if ObjectRayIntersection(o, backRay) then
          AddObjectToBackCandidates(o)
        end if
      end if
    end for
  end for
end procedure
```

Innerhalb eines Voxels werden alle seine Kandidaten an allen Shafts des Line Spaces für dieses Voxel geclippt. Befindet sich ein geclipptes Dreick innerhalb eines Shafts, so wird anschließend ein Schnittpunkttest zwischen dem Mittelpunktstrahl des Shafts und dem Dreieck durchgeführt. Wurde ein Schnittpunkt gefunden, so wird das Dreieck, sofern es eines der $\frac{k}{2}$ vordersten Dreiecke (bzgl. des Strahls)

ist, in die Kandidatenliste aufgenommen. Anschließend wird der Mittelpunktstrahl invertiert und das Vorgehen für den „back"-Buffer wiederholt.

Die Kandidatenliste pro Shaft enthält also nach Abschluss der Generierung die $\frac{k}{2}$ vordersten und $\frac{k}{2}$ hintersten Objekte innerhalb des Shafts. Diese Aufteilung in „front"- und „back"-Buffer ist für die spätere Traversierung des Line Spaces wichtig.

Alle dabei generierten Daten werden in Texturen gespeichert. Der Line Space ist durch eine zweikanalige 16 Bit unsigned Integer (*RG16UI*) 3D-Textur repräsentiert, welche die Koordinaten der jeweiligen Shafts in der Kandidaten-Textur enthält. Um die Line Spaces möglichst platzsparend und somit Speicher-effizient zu gestalten, wurden die entsprechenden Textur-Einträge wie in Abbildung 14 gepackt. Die Kandidaten-Textur ist ein einkanaliges 32 Bit unsigned Integer (*R32UI*) Textur-Array, welches aus k Layern besteht und die Kandidaten eines Shafts in Form ihrer Objekt-ID enthält. Im Folgenden wird diese Methode als der „normale" Line Space bezeichnet.

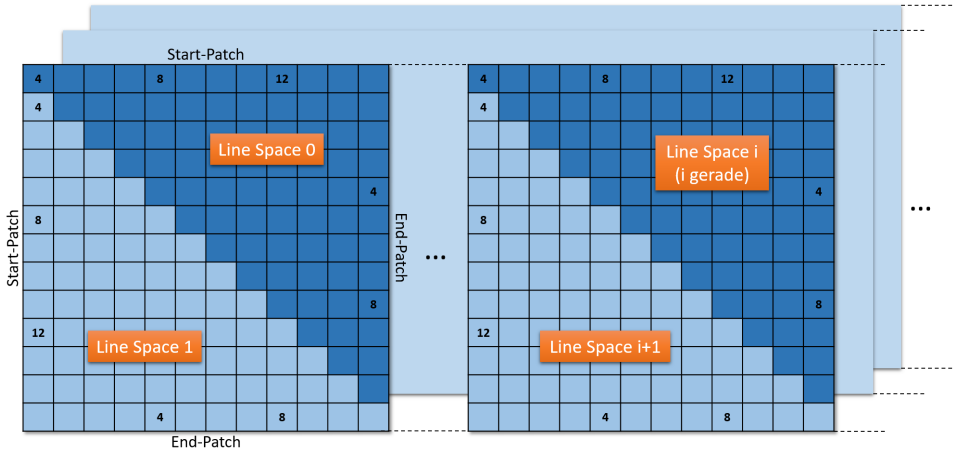

Abbildung 14: Aufbau der Line Space Textur (gepackt, N = 4)

7.1.1 Dynamische Speicherallokation

Das größte Problem an diesem Verfahren ist die statische Speicherallokation, welche dazu führt, dass in einem Shaft maximal k Kandidaten gespeichert werden können. Bei einer hohen Geometriedichte pro Voxel kann es dadurch zu „Löchern" im Line Space kommen, da zum Teil relevante Dreiecke nicht in die Kandidatenliste aufgenommen werden. Wird der Line Space lediglich für die indirekte Beleuchtung (also für diffuse Strahlen) verwendet, fällt dies nicht sehr auf, bei Spiegelungen oder transparenten Objekten führt es jedoch zu gut sichtbaren Artefakten.

Eine Lösung für dieses Problem wäre eine dynamische Speicherallokation, welche im Rahmen dieser Arbeit aber nicht implementiert wurde. Dabei würde ähnlich wie bei der Generierung des Voxel-Gitters in einem ersten Durchgang die

Größe der Kandidatenliste des Line Spaces bestimmt werden. Damit könnte diese dann dynamisch allokiert werden. In einem zweiten Durchgang würden dann basierend auf den Präfixsummen die Kandidaten für jeden Shaft in die Kandidatenliste eingefügt werden. Dadurch hätte jeder Shaft exakt genug Speicherplatz für alle darin enthaltenen Kandidaten und es würde nicht mehr zu Artefakten kommen. Ein kleiner Nachteil wäre jedoch die etwas längere Generierungszeit, die durch die aufwändigere Speicherallokation entstehen würde.

7.2 Traversierung

Die Traversierung des Line Spaces basiert ebenfalls auf der Voxelisierung. Hier wird zuerst eine normale Traversierung der Voxel mittels eines 3D-DDA-Algorithmus [FI85] durchgeführt. Anders als bei der reinen Verwendung des Voxel-Gitters wird innerhalb eines Voxels nun aber nicht ein Schnittpunkttest mit allen Kandidaten des Voxels durchgeführt. Stattdessen wird der zum Strahl gehörende Shaft im Line Space berechnet und außerdem entschieden, ob der Strahl von vorne oder von hinten durch den Shaft verläuft. Abhängig davon wird anschließend ein Schnittpunkttest des Strahls mit allen Kandidaten im „front"- oder „back"-Buffer der Kandidatenliste durchgeführt.

7.3 Pre-Illuminated Line Space

Der Pre-Illuminated Line Space speichert keine Kandidaten, sondern Farben in den Shafts. Hierfür werden ähnlich wie beim normalen Line Space der vorderste und hinterste Schnittpunkt in einem Shaft gesucht. Wurden die entsprechenden Objekte gefunden, werden diese anschließend mithilfe von Schattenfühlern zu allen Lichtquellen (wie beim Path Tracing) beleuchtet und eine „front"- und eine „back"-Farbe in ein vierkanaliges 16 Bit Float (*RGBA16F*) Textur-Array mit zwei Layern gespeichert.

Zwar dauert die Vorberechnung der beleuchteten Farben aufgrund der Schattenfühler deutlich länger als die Generierung des normalen Line Spaces, jedoch können die Farben im Path Tracing dann direkt ausgelesen werden. Da während des Path Tracings also bis auf das Herausfinden des Shafts keine weiteren Berechnungen nötig sind, bietet der Pre-Illuminated Line Space noch einmal eine deutliche Beschleunigung gegenüber dem normalen Line Space. Diese Methode eignet sich hauptsächlich für diffuse Strahlen und bedingt für Schattenstrahlen.

Für diese Beschleunigung müssen allerdings zwei Nachteile in Kauf genommen werden: Zum einen ist die gesamte Beleuchtung statisch. Das heißt, dass alle Lichtquellen in der Szene nach der Berechnung des Pre-Illuminated Line Spaces nicht mehr verändert (z.B. bewegt) werden können, da alle Beleuchtungsinformationen in den Line Space „gebacken" wurden. Zum anderen hat sich im Laufe der Implementierung herausgestellt, dass je nach Voxel-Auflösung Fehler bzw. Artefakte in der Schattenberechnung und/oder der indirekten Beleuchtung entstehen

können. Auf dieses Problem wird Evaluationsteil dieser Arbeit genauer eingegangen.

7.4 PNMT Line Space

Der PNMT Line Space speichert für jeden gefüllten Shaft jeweils die **P**osition, **N**ormale, **M**aterial und **T**exturkoordinaten (ähnlich wie bei einem G-Buffer) des vordersten und hintersten Schnittpunkts. Dabei wird für die Schnittpunktberechnung genauso wie beim normalen Line Space vorgegangen, jedoch wird nur der vorderste und hinterste Schnittpunkt in einem Shaft gesucht. Wurden die Schnittpunkte berechnet, werden anschließend die oben genannten Werte des jeweiligen Schnittpunktes in ein vierkanaliges 16 Bit Float (*RGBA16F*) Textur-Array mit vier Layern geschrieben, was eine Gesamtanzahl von 16 Werten pro Shaft ergibt. Abbildung 15 zeigt den Aufbau der PNMT-Textur.

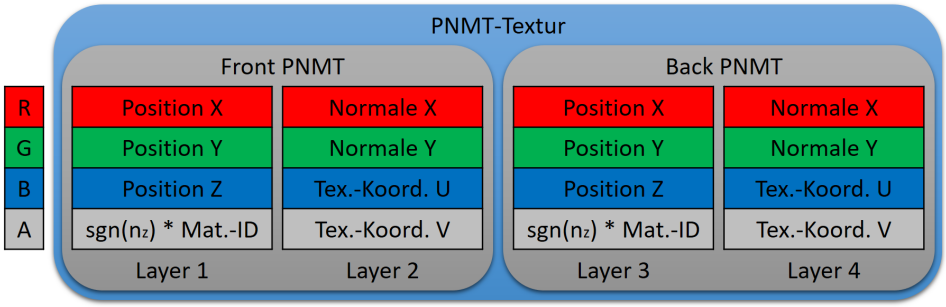

Abbildung 15: Aufbau der PNMT-Textur (alle Werte 16 Bit Float)

Für die Normale werden hier nur die x- und y-Koordinaten gespeichert. Zusätzlich wird das Vorzeichen der z-Koordinate in die Material-ID (welche immer positiv ist) kodiert. Somit kann die Normale effizient gespeichert werden und später im Path Tracing eindeutig rekonstruiert werden. Die folgende Formel wird zur Rekonstruktion der z-Koordinate verwendet:

$$n_z = Vorzeichen_z * \sqrt{1 - n_x^2 - n_y^2} \tag{9}$$

Beim Path Tracing werden dann die benötigten PNMT-Daten ausgelesen und mit diesen ein Beleuchtungswert berechnet. Zwar müssen dafür Schattenfühler verfolgt werden, jedoch fällt die gesamte Schnittpunktberechnung mit der Szenengeometrie weg. Bezüglich der Performance stellt diese Methode einen Mittelweg zwischen dem normalen Line Space und dem Pre-Illuminated Line Space dar, wobei die Beleuchtung komplett dynamisch bleibt. Lichtquellen können also während des Path Tracings verändert werden, ohne dass der Line Space neu berechnet werden muss. Auch diese Methode ist vor allem für diffuse Strahlen und je nach Voxel-Auflösung für Schattenstrahlen sehr gut nutzbar. Allerdings kann es auch hier zu den bereits erwähnten und später näher beschriebenen Schatten- und Beleuchtungsartefakten kommen.

8 Weitere untersuchte Verfahren

Um dem Path Tracer in der visuellen Qualität und/oder in der Berechnungszeit zu verbessern, wurde das standardmäßige Path Tracing durch verschiedene Verfahren erweitert. Von diesen benötigen einige dafür zusätzliche Renderpasses, andere integrieren sich direkt in die Standard-Pipeline.

8.1 G-Buffer

Anstatt den ersten Strahl von der Kamera aus in die Szene zu schießen, kann zur Berechnung der nötigen Informationen sehr gut ein klassischer G-Buffer [ST90] verwendet werden. Dieser hat den Vorteil, dass er durch die OpenGL-Pipeline deutlich schneller berechnet werden kann, als die Verfolgung des primären Kamerastrahls. Ein kleiner Nachteil ist jedoch, dass das beim klassischen Path Tracing implementierte *Jittering*[2] der Primärstrahlen nicht mehr möglich ist und es dadurch zu *Aliasing*[3] an Kanten im Bild kommt.

Der hier verwendete G-Buffer besteht dabei aus der Position, der Normalen, den UV-Koordinaten und dem Materialindex, welche in einem zusätzlichen Renderpass in drei Texturen gerendert werden. Beim Path Tracing können diese Informationen dann direkt ausgelesen und der entsprechende Punkt direkt beleuchtet werden ohne den Punkt erst durch Strahlenverfolgung berechnen zu müssen. Dies erhöht die Frame-Rate um ca. 10% gegenüber dem kompletten Path Tracing.

8.2 Rauschreduzierung durch Filterung

Um dem Problem der anfangs stark verrauschten Ergebnisbilder des Path Tracings entgegenzuwirken, wurden diverse lineare und nicht-lineare Rauschreduktionsfilter aus der Bildverarbeitung auf das Bild angewendet. Dabei wurde der Einfluss des Filters bei steigender Sample-Anzahl kontinuierlich verringert, da bei vielen Samples das Rauschen durch die Konvergenz des Monte-Carlo-Verfahrens von selbst verschwindet und die Filter dann nicht mehr nötig sind. Besonders für die ersten Frames können sie aber durchaus zu einer verbesserten Qualität des Bildes führen. Die verwendeten Filter orientieren sich alle an den von Priese in [Pri15] beschrieben Algorithmen. Abbildung 16 zeigt einen Vergleich der verschiedenen Filter für das erste Sample.

8.2.1 Gaußfilter

Das Gaußfilter [4] ist ein reines lineares Glättungsfilter. Jedes Pixel wird dabei entsprechend der Gauß'schen Normalverteilung mit den umliegenden Pixeln vermischt.

[2] Jittering: Aufaddieren eines zufälligen Offsets auf z.B. eine Position oder Koordinate.

[3] Aliasing: Entstehung treppenartiger Strukturen an Kanten im Bild.

[4] Das Filter in der Bildverarbeitung ist bezüglich des Genus neutral. Folglich ist hier die Verwendung des Artikels „das" grammatikalisch korrekt.

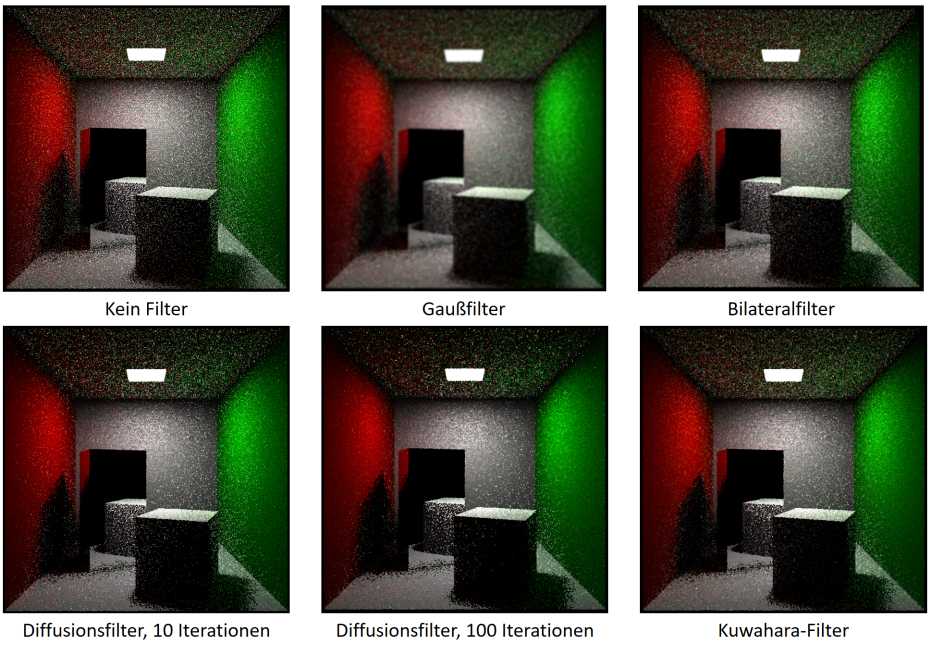

Abbildung 16: Rauschreduktionsfilter im Vergleich, angewendet auf das erste Sample

Dadurch erscheint das Bild etwas verwaschen, weil nicht nur das Rauschen beseitigt, sondern auch Kanten verwischt werden. Da Rauschen allerdings von den meisten Menschen als angenehmer empfunden wird als verschwommene Konturen, ist dieses Filter nicht sehr gut geeignet.

8.2.2 Bilateralfilter

Das Bilateralfilter ist eine Erweiterung des Gaußfilters. Hier wird nicht wie beim Gaußfilter nur ortsabhängig, sondern auch werteabhängig gefiltert. Wie stark ein umliegender Pixel mit dem aktuell zu bearbeitenden Pixel vermischt wird, hängt also von deren Abstand *und* deren Ähnlichkeit zueinander ab. Beide Abhängigkeiten werden wieder mit der Gauß'schen Normalverteilung gewichtet. Vorteilhaft ist, dass dieses Filter gleichzeitig Rauschen reduziert und Kanten erhält, weshalb es deutlich besser geeignet ist als das Gaußfilter.

8.2.3 Anisotrophisches Diffusionsfilter

Dieses Filter basiert auf dem physikalischen Phänomen der Diffusion. Dabei wird der Zustands- bzw. Farbausgleich zwischen benachbarten Pixeln simuliert. Es diffundiert dabei immer die Farbe der vier umliegenden Pixel in den aktuell betrachteten Pixel. Diese Diffusion hängt von der Diffusivität und der Änderungsgeschwindigkeit ab, welche für ein bestmögliches Ergebnis angepasst werden können. Da bei diesem Verfahren in einem Diffusionsschritt nur sehr kleine Änderungen statt-

finden, muss das Filter mehrfach (ca. 10 bis 100 mal) angewendet werden. Auch dieses Filter wirkt glättend und gleichzeitig kantenerhaltend, ist jedoch wegen der iterativen Ausführung etwas aufwändiger zu berechnen als das Bilateralfilter.

8.2.4 Kuwahara-Filter

Das Kuwahara-Filter wirkt ebenfalls kantenerhaltend und rauschreduzierend. Es basiert auf dem Vergleich von Varianzen der Nachbarpixel. Zuerst werden vier Pixelgruppen aus dem betrachteten Pixel und den acht umliegenden Pixeln gebildet und deren Varianzen berechnet. Anschließend wird der Pixelwert auf den Mittelwert der Gruppe mit der kleinsten Varianz gesetzt. Dieses Filter kann am schnellsten berechnet werden und bietet gute Ergebnisse, ist anders als die vorher beschriebenen Filter allerdings (außer durch *Blending*) schlecht mit der Sample-Anzahl skalierbar.

8.3 Bidirektionales Path Tracing

Wie schon im theoretischen Teil erklärt, wird beim bidirektionalen Path Tracing zusätzlich zum Kamerapfad auch ein Lichtpfad verfolgt. In dieser Arbeit wurde dafür in jeder *Shader-Invocation* (also für jedes Pixel im Ergebnisbild) zuerst ein Lichtpfad für jede Lichtquelle bis zu einer bestimmten Tiefe berechnet. Dabei wurde prinzipiell auf die gleiche Art und Weise vorgegangen, wie bei der Kamerapfadverfolgung. Die Strahlgenerierung geschieht hier jedoch von der Lichtquelle aus. Außerdem werden nicht nur die Farben aufsummiert, sondern bei jedem Schnittpunkt des Lichtpfades mit der Szene ein Licht-Punkt (*Light-Vertex*) für die spätere Beleuchtung in einem Array gespeichert. Sobald alle Lichtpfade berechnet wurden, wird der Kamerapfad wie gewohnt verfolgt. Wird dann ein Punkt beleuchtet, wird nicht nur über alle Lichtquellen, sondern auch über alle Light-Vertices iteriert und ein Beleuchtungswert berechnet.

Leider ist das bidirektionale Path Tracing aufgrund der zusätzlichen Pfadverfolgung und der deutlich höheren Anzahl an generierten Lichtquellen sehr aufwändig zu berechnen, weshalb nur eine extrem geringe Frame-Rate erreicht wird. Zwar kann die Berechnungszeit durch zufälliges Auswählen einer kleineren Anzahl an Lichtquellen aus der Menge aller Lichtquellen (inklusive Light-Vertices), welche dann zur Beleuchtung verwendet werden, etwas beschleunigt werden. Jedoch wird die Anwendung dadurch nicht wirklich interaktiv.

8.4 Hybrides Path Tracing

Das hybride Path Tracing ist prinzipiell ebenfalls eine Form des bidirektionalen Path Tracings. Es wird jedoch versucht, den größten Nachteil, also die langen Berechnungszeiten des bidirektionalen Path Tracings, zu umgehen. Dazu wird die Tatsache ausgenutzt, dass die CPU während des Path Tracings auf der GPU wenig bis gar nicht ausgelastet ist. Die Idee ist also, nur den Kamerapfad auf der GPU

zu berechnen und den Lichtpfad für jeden Lichtquelle durch ein getrenntes Path Tracing auf der CPU zu generieren.

Hierfür wurde der GPU Path Tracer auf die CPU (also in C++-Code) übertragen und in einem zusätzlichen *Thread* zur Renderschleife ausgeführt. Dabei wird genau wie beim standardmäßigen bidirektionalen Path Tracing für jede Lichtquelle ein Lichtpfad verfolgt und alle Schnittpunkte mit der Szene in Form von Light-Vertices in ein Array bzw. eine Liste gespeichert. Einmal pro Durchlauf der Renderschleife wird diese Liste abgerufen und als SSBO an den Path Tracing Shader übergeben, welcher dann die darin enthaltenen Lichtpunkte zur Beleuchtung verwenden kann. Durch diese gleichmäßigere Auslastung von GPU und CPU können deutlich höhere Frame-Raten erreicht werden, was zu einer weitaus besseren Interaktivität als beim klassischen bidirektionalen Vorgehen führt.

8.5 Checkerboard Rendering

Das Checkerboard Rendering [Man16] zielt im allgemeinen ebenfalls auf höhere Frame-Raten ab. Dieses Verfahren wurde in dieser Arbeit jedoch nicht so sehr für eine bessere Interaktivität, sondern vorrangig für eine schnellere Konvergenz des Bildes implementiert. Die Idee hierbei ist, in einem Render-Durchgang nur die eine Hälfte der Pixel zu rendern und erst im nächsten Durchgang die andere. Dabei wird für die gerenderten Pixel ein Schachbrettmuster (*engl. „Checkerboard"*) verwendet und abwechselnd die „schwarzen" oder „weißen" Pixel des „Schachbretts" gerendert. Abbildung 17 veranschaulicht dieses alternierende Vorgehen.

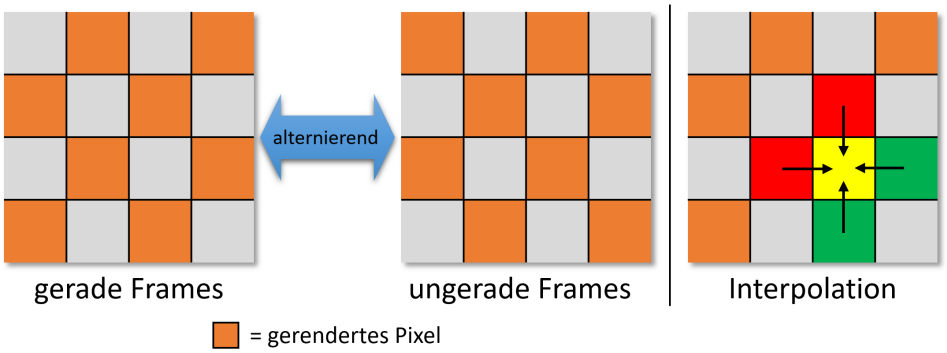

Abbildung 17: Checkerboard Rendering. Links: Alternierend gerenderte Pixel. Rechts: Berechnung der nicht gerenderten Pixel

Dadurch, dass in einem Render-Durchgang nur das halbe Bild berechnet wird, erhält man logischerweise auch etwa die doppelte Frame-Rate als beim „vollen" Rendering. Diese Eigenschaft kann nun vor allem dazu genutzt werden, in einem Shader-Aufruf direkt mehrere Samples zu berechnen, wodurch das in einem Render-Durchgang berechnete Bild rauschfreier wird und der zusätzliche Aufwand zum Starten des Shaders wegfällt. Die effektive Zahl der berechneten Strahlen pro

Sekunde wird dabei allerdings nicht nennenswert verändert; die Interaktivität verbessert sich sogar.

Ein wichtiger Sonderfall bei diesem Verfahren ist der erste Render-Durchgang, nachdem das inkrementelle Path Tracing zurückgesetzt wurde. In diesem Fall liegen nur die Farbinformationen jedes zweiten Pixels vor, alle anderen Pixel wurden noch nicht berechnet. Dadurch würde es in diesem Frame zu einem schachbrettartigen Muster im Bild kommen. Um dies zu verhindern wird für das erste Frame im Compositing-Pass die Farbe eines noch nicht gerenderten Pixels durch Interpolation der vier angrenzenden Pixel berechnet. Im ersten Frame fällt dies in der Regel nicht auf, da schon ab dem zweiten Frame alle Bildinformationen vorliegen und nicht mehr interpoliert werden muss.

8.6 Shadow Mapping

Für das Shadow Mapping wird von jeder Lichtquelle aus eine Tiefentextur in Richtung der Normalen der Lichtquelle gerendert. Anschließend kann während des Path Tracings der Schattentest anhand der erzeugten „Schattentexturen" durchgeführt werden. Dafür wird die Position eines Schnittpunkts zuerst in das Koordinatensystem der zu testenden Lichtquelle transformiert und anschließend der Tiefenwert dieses Punktes mit dem entsprechenden Eintrag in der Schattentextur verglichen. Ist der Tiefenwert des Schnittpunktes größer als der Wert aus der Schattentextur, so ist der Punkt bezüglich der untersuchten Lichtquelle verschattet.

Um Aliasing zu vermeiden, werden die Texturkoordinaten für den Schattentexturzugriff um einen zufälligen Wert verschoben, was dem Grundsatz des *Percentage Closer Filtering (PCF)* [Fer04, Kap. 11] entspricht. Außerdem wird auf den transformierten Tiefenwert des Schnittpunktes ein *Slope Scale Depth Bias* addiert. Dieser ist abhängig vom Winkel zwischen Lichtvektor und Oberflächennormale und beseitigt die beim Shadow Mapping auftretenden Artefakte (*Shadow Acne*).

Zwar sind die so berechneten Schatten nicht immer physikalisch korrekt, da besonders bei flächigen Lichtquellen keine korrekten weichen Schatten erzeugt werden. Allerdings ist dieses Verfahren weitaus schneller als das Verfolgen eines Schattenfühlers, weil hier im Grunde nur eine Transformation, ein Texturzugriff und eine Vergleichsoperation pro Lichtquelle nötig sind, um zu entscheiden, ob ein Objekt im Schatten liegt oder nicht.

Teil III
Evaluation

In diesem Teil der Arbeit werden die implementierten Techniken bzw. Programme nach verschiedenen Kriterien analysiert und bewertet.

9 Test-Setup

9.1 Testsystem

Das Testsystem bestand Hardware-seitig aus einem Intel Core i7-6800K (3.4 GHz) Prozessor und 32GB DDR4 RAM (2133 MHz). Als Grafikkarte diente eine NVidia GeForce GTX 1080 mit einem Basistakt von 1746 MHz, 8GB GDDR5 VRAM und 2560 Prozessoren (*CUDA-Cores*). Software-seitig wurden Microsoft Windows 10 als Betriebssystem und der NVidia Grafikkartentreiber der Version 376.53 verwendet.

9.2 Vorgehensweise beim Testen

Alle Tests wurden mit der gleichen Viewport-Auflösung von 1280x720 gerendert. Das Testmodell bzw. die Testszene wurde gewechselt, jedoch hatte das zentrale Objekt immer das gleiche Material: 90% diffus, 10% spiegelnd, 0% transparent und eine Glanzzahl von 50. Die Voxel-Auflösung wurde bei jeder Szene so angepasst, dass beim Path Tracing die besten Ergebnisse bezüglich der Performance je nach Datenstruktur erzielt wurden. Die Line Space Auflösung betrug in allen Tests $N = 5$ und es wurde der PNMT Line Space verwendet. Die Shadow-Map-Auflösung betrug 1024x1024. Beim Testen wurden zum einen jeweils die Generierungszeiten für das Voxel-Gitter und den Line Space und zum anderen die benötigte Zeit pro Frame für das Path Tracing bei verschiedenen Berechnungsmethoden gemessen. Aus Letzterem wurden dann die Frame-Rate (FPS) und die Anzahl an verfolgten Strahlen pro Sekunde (MRPS, in Millionen) berechnet.

9.3 Testszenen

Insgesamt wurden sieben verschiedene Szenen erstellt und mit diesen getestet. Dabei wurde versucht eine möglichst gute Diversität bezüglich Polygonanzahl und offenen und geschlossenen Szenen zu erreichen. Die verwendeten Testszenen sind die Folgenden:

1. **Teapot**: Offene Szene, 15.704+8 Dreiecke, drei Lichtquellen, max. Pfadlänge: 2
2. **Bunny**: Offene Szene, 69.451+8 Dreiecke, drei Lichtquellen, max. Pfadlänge: 2

3. **Dragon**: Offene Szene, 871.306+8 Dreiecke, drei Lichtquellen, max. Pfadlänge: 2
4. **Cornell-Box**: Geschlossene Szene, 36 Dreiecke, eine Lichtquelle, max. Pfadlänge: 3, ein diffuser und ein spiegelnder Quader innerhalb der Box
5. **Cornell-Box mit Teapot**: Geschlossene Szene, 15.704+12 Dreiecke, eine Lichtquelle, max. Pfadlänge: 3
6. **Cornell-Box mit Bunny**: Geschlossene Szene, 69.451+12 Dreiecke, eine Lichtquelle, max. Pfadlänge: 3
7. **Cornell-Box mit Dragon**: Geschlossene Szene, 871.306+12 Dreiecke, eine Lichtquelle, max. Pfadlänge: 3

10 Generierung

10.1 Generierungszeiten

Die für die Generierung des Voxel-Gitters und des Line Spaces benötigte Zeit hängt von vielen Faktoren ab. Tabelle 1 zeigt Generierungszeiten für verschiedene Szenen, Voxel-Auflösungen und Line Space Auflösungen.

Tabelle 1: Generierungszeiten für Voxel-Gitter und Line Space in Millisekunden

Szene	Voxel-Auflösung	LS-Auflösung	$T_{Gen}(Voxel)$	$T_{Gen}(LineSpace)$
4	8x8x8	4	43	154
4	8x8x8	5	43	269
5	8x8x8	5	58	644
5	32x32x32	5	47	1795
6	8x8x8	5	51	3915
6	64x64x64	5	71	10081
7	8x8x8	5	95	94393
7	72x72x72	5	88	53675

Zum einen hat die Komplexität der Szene eine direkte Auswirkung auf die Generierungszeit. Je mehr Polygone in der Szene enthalten sind, umso mehr Kandidaten müssen in die Kandidatenlisten des Voxel-Gitters einsortiert werden bzw. umso aufwändiger ist die Suche nach dem vordersten und hintersten Schnittpunkt pro Shaft für den PNMT Line Space.

Zum anderen führen sowohl höhere Voxel-Auflösungen als auch höhere Line Space Auflösungen zu längeren Generierungszeiten für den Line Space, nicht aber für das Voxel-Gitter. Das Voxel-Gitter ist bei höheren Auflösungen tendenziell sogar etwas schneller gefüllt, da pro Voxel weniger Kandidaten in die Kandidatenliste einsortiert werden müssen. Der Line Space braucht bei höheren Auflösungen länger, da dadurch deutlich mehr Shafts bei der Generierung behandelt werden müssen.

10.1.1 Speicherverbrauch

Der Speicherverbrauch des Voxel-Gitters ist sowohl abhängig von dessen Auflösung, als auch von der Polygonanzahl der Szene. Eine höhere Auflösung bedeutet eine größere Voxel-Textur, welche mehr Speicherplatz benötigt. Je nach Polygonanzahl muss die Kandidatenliste entsprechend viele Kandidaten enthalten, was bei komplexen Szenen ebenfalls zu einem höheren Speicherverbrauch führt. Da sowohl in der Voxel-Textur, als auch in der Kandidatenliste nur ein 32 Bit unsigned Integer pro Eintrag gespeichert wird, ist der Speicherplatzverbrauch mit einigen Megabyte für das Voxel-Gitter recht gering.

Deutlich höher ist der Speicherplatzbedarf des Line Spaces. Dieser hängt neben der Voxel-Anzahl bzw. -Auflösung auch von der Auflösung des Line Spaces ab. Während sich der Speicherverbrauch bei höheren Voxel-Auflösungen linear erhöht, steigt er bei einer höheren Line Space Auflösung quartisch (wegen $15N^4$). Dies liegt an der daraus resultierenden größeren Anzahl an Shafts. Insgesamt beträgt der benötigte Speicher für den Line Space (inklusive Kandidaten bzw. PNMT-Textur) je nach gewählten Auflösungen und gespeicherten Daten in der Regel zwischen einigen hundert Megabyte bis hin zu mehreren Gigabyte an VRAM.

11 Echtzeitfähigkeit

Im Folgenden wird die Performance und somit die Echtzeitfähigkeit des Path Tracings unter Verwendung des Line Spaces in verschiedenen Szenarien untersucht. Echtzeitfähigkeit wird laut [AMHH08, Kap. 1] bereits bei einer Frame-Rate von 15 FPS erreicht, für eine gute Interaktivität ist allerdings mindestens die doppelte (30) oder sogar vierfache (60) Bildwiederholungsrate notwendig.

11.1 Große Szenen

Der Line Space kommt beim Path Tracing deutlich besser mit großen Szenen zurecht als das Voxel-Gitter. Dies liegt daran, dass deutlich weniger Objekte pro Shaft als pro Voxel auf Schnittpunkte getestet werden müssen. Beim PNMT Line Space sind sogar gar keine Objekte zu testen, sondern nur Textur-Look-Ups nötig.

Diagramm 1 zeigt einen Vergleich der Frame-Raten einer komplett mit dem Line Space und einer komplett mit dem Voxel-Gitter gerenderten Szene. Hier ist gut zu erkennen, dass bei steigender Polygonanzahl in der Szene die Performance des Voxel-Gitters stark abfällt, da selbst bei hohen Voxel-Auflösungen viele Dreiecke in einem Voxel enthalten sind und auf Schnittpunkte getestet werden müssen. Für den Line Space steigt zwar die Generierungszeit, das Path Tracing wird jedoch nicht langsamer, da die Informationen im PNMT Line Space unabhängig von der Polygonanzahl sind. Auch bei einem Line Space mit Kandidatenliste wäre zwar ein leichter Performance-Rückgang zu erwarten, jedoch nicht so stark wie beim Voxel-Gitter.

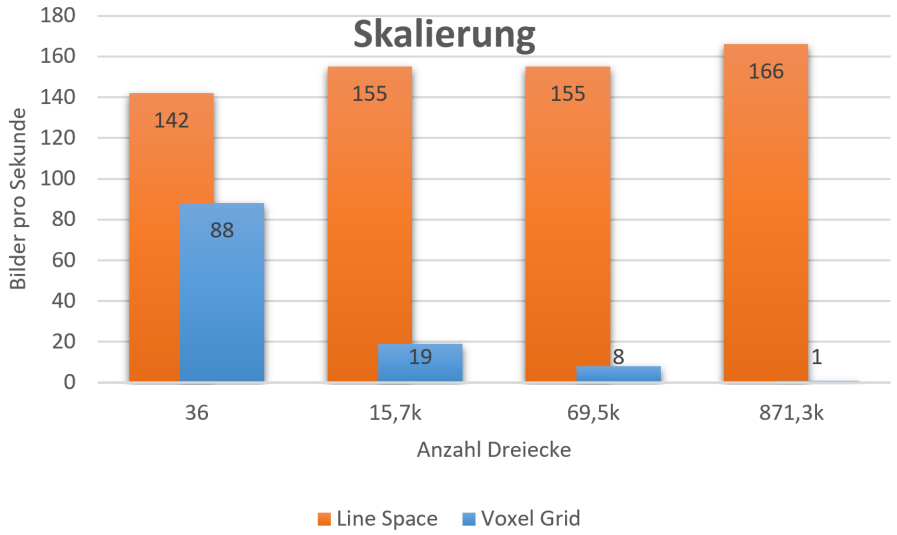

Diagramm 1: Bilder pro Sekunde (FPS) bei verschiedenen Szenengrößen. Blau: Sekundär- und Schattenstrahlen mit Voxel-Gitter. Orange: Sekundär- und Schattenstrahlen mit Line Space

Höhere Voxel-Auflösungen resultieren generell in einer besseren Performance des Voxel-Gitters und des normalen Line Spaces, da dann pro Voxel und Shaft weniger Geometrie enthalten ist. Auch eine höhere Line Space Auflösung ist vorteilhaft für den normalen Line Space mit Kandidatenlisten. Für den PNMT Line Space sind hingegen niedrigere Voxel-Auflösungen gut für die Performance, da dann insgesamt über weniger Voxel iteriert werden muss.

11.2 Diffuse Strahlen

Besonders bei sehr diffusen Szenen wird die Stärke des PNMT Line Spaces deutlich. Werden alle diffusen Sekundärstrahlen mit dem PNMT Line Space verfolgt, so wird mehr als die 20-fache Performance erreicht, als bei der Strahlverfolgung über das Voxel-Gitter, wie Diagramm 2 zeigt. Befinden sich in einer Szene größtenteils diffuse Objekte, so können nahezu alle Sekundärstrahlen mit dem Line Space berechnet werden, was die Frame- bzw. MRPS-Rate stark erhöht.

Leider sinkt die Performance, sobald sich stark spiegelnde oder transparente Objeke in der Szene befinden, da dafür die Genauigkeit des PNMT Line Spaces nicht ausreicht und auf das Voxel-Gitter zurückgegriffen werden muss, was die Strahlverfolgung verlangsamt. Ein Line Space mit Kandidatenlisten würde hier einen Mittelweg darstellen, wäre jedoch in vielen Fällen etwas langsamer als der PNMT Line Space.

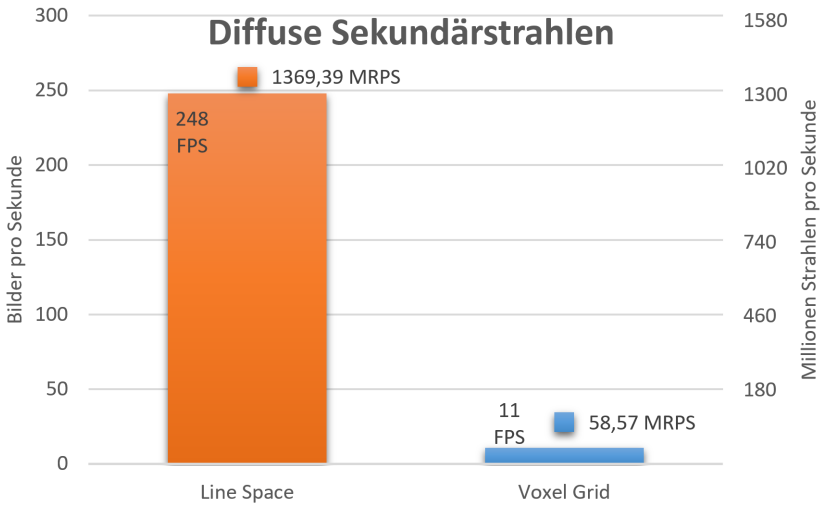

Diagramm 2: Bilder pro Sekunde (FPS) und MRPS bei verschiedenen Strahlverfolgungstechniken für diffuse Sekundärstrahlen. Szene: Cornell-Box mit Bunny, Schattentests mit Shadow-Maps

11.3 Schatten

Auch die Schattenberechnung profitiert stark vom Line Space. Hier kann mit dem PNMT Line Space unabhängig von den in der Szene enthaltenen Objekten und Materialien etwa die zehnfache Leistung verglichen zum Voxel-Gitter erreicht werden. Der Grund für diese Beschleunigung ist derselbe wie für die diffusen Strahlen, wobei auch hier der PNMT Line Space den auf Kandidatenlisten basierenden Line Space bezüglich der Performance überholt. Noch schneller als der Line Space ist die Verwendung von Shadow-Mapping für Schattentests. Allerdings muss beim Shadow-Mapping auf die physikalisch korrekt berechneten weichen Schatten verzichtet werden, welche mit dem Line Space und dem Voxel-Gitter durchaus dargestellt werden können. Ein Performance-Vergleich der drei Techniken ist in Diagramm 3 zu sehen.

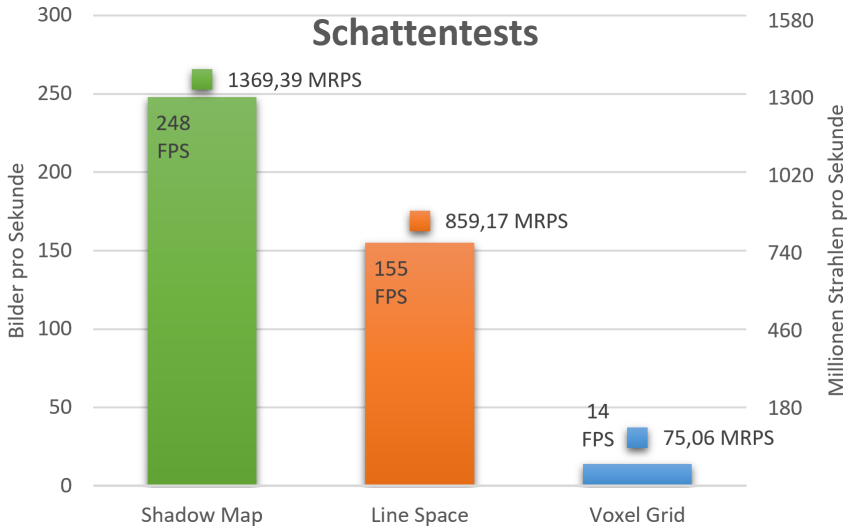

Diagramm 3: Bilder pro Sekunde (FPS) und MRPS bei verschiedenen Strahlverfolgungstechniken für Schattenfühler. Beim Shadow-Mapping wurden Schattentexturabfragen als Strahlen gewertet. Szene: Cornell-Box mit Bunny, diffuse Sekundärstrahlen mit Line Space

12 Visuelle Qualität

Nach dem Betrachten der Echtzeitfähigkeit wird nun die visuelle Qualität der mit dem Line Space gerenderten Bilder untersucht. Dabei wird besonders auf physikalische Korrektheit und das Vorhandensein von Artefakten geachtet. Als Referenz dient hier immer das Voxel Gitter, da die damit berechneten Bilder keine Fehler enthalten. Hervorzuheben ist, dass die im Folgenden beschriebenen Artefakte alle durch die beschleunigenden Vereinfachungen des PNMT Line Spaces entstehen und bei einem Line Space mit Kandidatenliste nicht auftreten; letzterer ist dafür allerdings langsamer.

12.1 Indirekte Beleuchtung

Für die indirekte Beleuchtung bzw. die Verfolgung von diffusen Strahlen kann der Line Space in fast allen Fällen verwendet werden. Die Präzision des PNMT Line Spaces reicht in der Regel aus, da diffuse Strahlen ohnehin zufällig über die obere Hemisphäre verteilt sind und somit die Struktur des Line Spaces nicht sichtbar wird. In Abbildung 18 ist die indirekte Beleuchtung im Vergleich zu einer reinen direkten Beleuchtung dargestellt.

Der einzige (recht seltene) Fall, in dem die Struktur des Line Spaces in der indirekten Beleuchtung sichtbar wird, ist an Stellen gegeben, wo das diffuse Material eine hohe Glanzzahl hat (also eher spiegelnd wirkt) und zwei Objekte nah aneinander sind. In diesem Fall sind vor allem bei diffusen Spiegelungen von planaren Flächen Artefakte sichtbar, wie in Abbildung 19 zu sehen ist. Diese entstehen durch ein Voxel-Offset, das beim PNMT Line Space notwendig ist, um die

„Selbstbeleuchtung” eines Objekts zu verhindern. Diesen Artefakten kann jedoch durch eine höhere Auflösung des Voxel-Gitters entgegengewirkt werden.

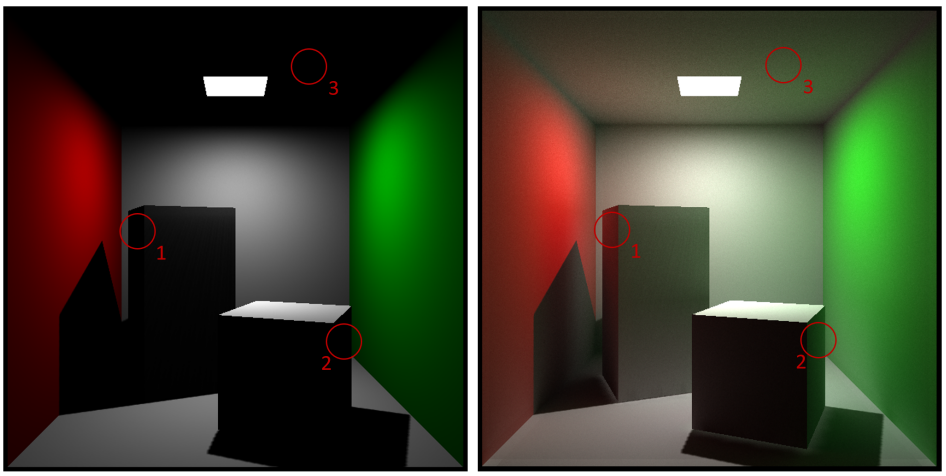

Abbildung 18: Direkte und indirekte Beleuchtung. Die rot markierten Stellen 1 und 2 können nur indirekt beleuchtet werden, da sie verdeckt oder abgewandt von der Lichtquelle sind. Die Decke der Cornell-Box (3) befindet sich sogar hinter der Lichtquelle

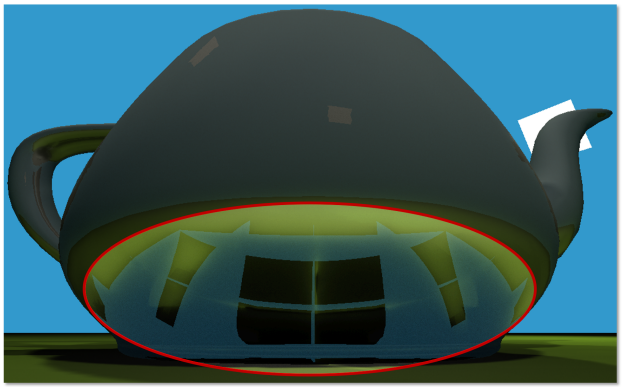

Abbildung 19: Bei der der diffusen Strahlverfolgung entstehende Artefakte durch den PNMT Line Space (die rot umrandete Fläche wurde zur Verdeutlichung stark aufgehellt). Das Material des Teapots hat eine Glanzzahl von 50

12.2 Schatten

Die Qualität der Schatten, welche durch den Line Space berechnet werden, ist in den meisten Fällen sehr nah oder sogar identisch mit den Ergebnissen des Voxel Gitters. Bei ausreichend groß gewählten Voxel- und Line Space Auflösungen entstehen nahezu keine Artefakte. Außerdem ist die physikalische Korrektheit der Schatten bezüglich weicher Schattenübergänge beim Line Space (wie auch beim Voxel Gitter) vorhanden, während sie bei den mit Shadow-Mapping berechneten Bildern gänzlich fehlt. Abbildung 20 zeigt einen Vergleich dieser drei verschiedenen Schattendarstellungen.

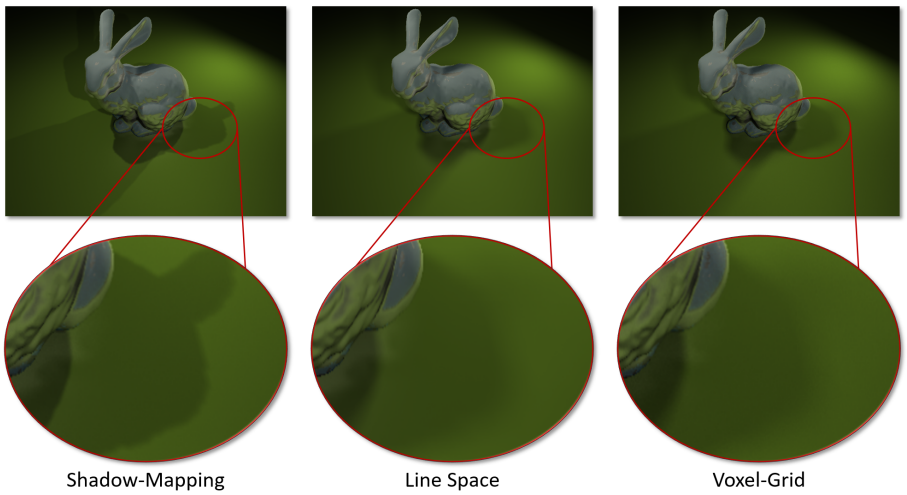

Abbildung 20: Vergleich der Schattenqualität bei verschiedenen Berechnungsmethoden (beim Shadow-Mapping werden die Lichtquellen als Punktlichtquellen angenommen)

Problematisch ist die Schattenberechnung mit dem Line Space, wenn die Voxel- bzw. Line Space Auflösung zu niedrig gewählt wird. Dann kann es zu deutlich sichtbaren Schattenartefakten kommen, wie Abbildung 21 zeigt. Diese Artefakte entstehen dadurch, dass beim PNMT Line Space für den Schattentest keine Schnittpunkte mit der tatsächlichen Geometrie gesucht werden, sondern ein „gefüllter" Shaft als Verdeckungskriterium dient. Sind die Shafts aber durch eine zu geringe Auflösung zu groß, so ist die Struktur des Line Spaces analog zur indirekten Beleuchtung an den Rändern von Schatten sichtbar. Auch kann es aufgrund des bereits erwähnten Voxel-Offsets zu fehlenden Schatten in Ecken und an Kanten der Szene kommen. Diese Probleme würden bei einem Line Space mit einer Kandidatenliste nicht entstehen. Auch beim PNMT Line Space können diese aber recht einfach durch eine bessere Voxel- und Line Space Auflösung gelöst werden, was jedoch auch eine längere Generierungszeit und einen höheren Speicherverbrauch mit sich bringt.

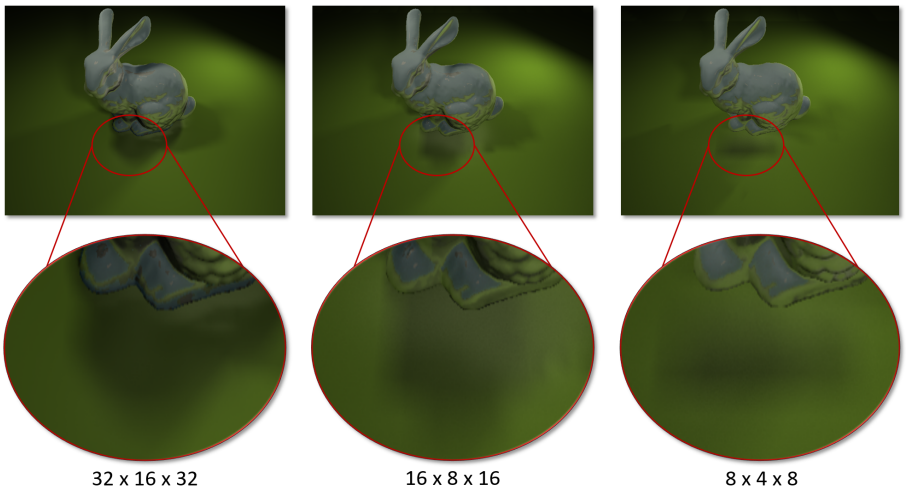

Abbildung 21: Schattenartefakte bei verschiedenen Voxel-Gitter-Auflösungen ($N = 5$)

12.3 Andere Beleuchtungseffekte

Durch die dynamische Auswahl der Strahverfolgungsmethode können trotz der (wenngleich selten auftretenden) Probleme mit Artefakten beim PNMT Line Space kompliziertere Materialien und Beleuchtungseffekte nahezu artefaktfrei dargestellt werden. Abbildung 22 zeigt die Qualität der Darstellung von transparenten und spiegelnden Materialien unter Einfluss einer Skybox.

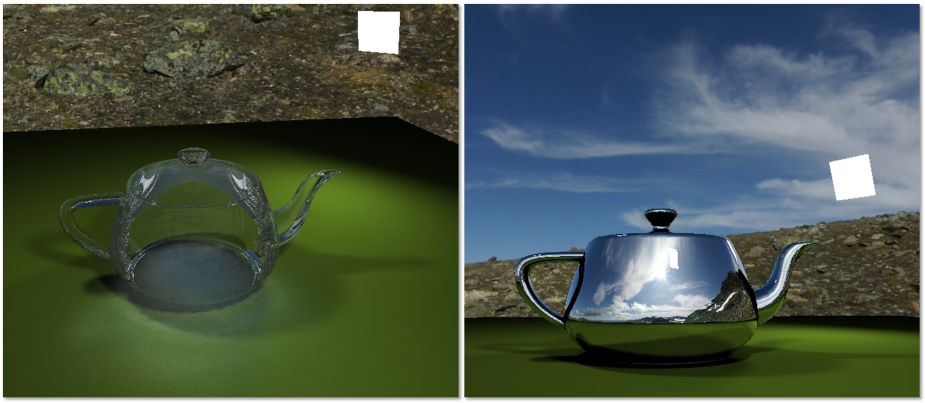

Abbildung 22: Links: Teapot aus Glas (bis zu 20 Bounces).
Rechts: Spiegelnder Teapot mit Reflexionen der Skybox

13 Ausblick

Auch wenn das Path Tracing unter Verwendung des Line Spaces schon durchaus sehr gute Ergebnisse liefert, so besteht immer noch viel Verbesserungspotenzial. Sowohl die Echtzeitfähigkeit, als auch die visuelle Qualität und die Berechnungszeiten könnten durch die nachfolgenden Verfahren verbessert werden.

13.1 Szenen Line Space

Eine weiterführende Idee des Line Spaces (welche zur Zeit von S. Müller untersucht wird) ist ein „globaler" Line Space. Dieser spannt sich über die gesamte virtuelle Szene und enthält pro Shaft eine „Signatur". Diese Signatur speichert, in welchen Intervallen der jeweilige globale Shaft gefüllt oder leer ist. Damit könnten noch größere leere Strecken traversiert werden, ohne über viele Voxel iterieren zu müssen, was abermals zu einer Beschleunigung der Strahlverfolgung führen könnte.

13.2 Objekt Line Space

Die Idee des Objekt Line Spaces (vorgeschlagen von K. Keul) ist es, die Datenstruktur nicht pro Voxel zu erstellen, sondern pro Objekt. Es würde also für jedes Objekt in der virtuellen Szene (z.B. Bunny, Teapot, etc.) ein eigener Line Space erzeugt und gespeichert werden. Damit könnte zuerst nur ein Schnittpunkttest mit der Bounding Box des Objekts durchgeführt werden. Anschließend müssten nur wenige Dreiecke des Objektes auf Schnittpunkte mit dem Strahl getestet werden. Dies hätte neben der zu erwartenden Beschleunigung der Strahlverfolgung auch den großen Vorteil, dass sich Objekte dynamisch in der Szene bewegen könnten, da die Objekttransformationen einfach auf den Objekt Line Space übertragen werden könnten.

13.3 Speichern und Laden

Um die langen Generierungszeiten des Line Spaces zu umgehen, könnte dieser einmalig pro Szene erstellt werden und anschließend in Form einer Datei auf der Festplatte abgespeichert werden. Für das Rendering einer Szene könnte dann einfach die jeweilige Line Space Datei geladen und die Daten auf die GPU transferiert werden. Dadurch müsste der Line Space nur bei einer Änderung der jeweiligen Szene neu erstellt werden und der Startvorgang des Renderings könnte beschleunigt werden.

13.4 Beleuchtungsmodelle

Zurzeit kann in der zu dieser Arbeit gehörenden Implementierung zur Beleuchtung lediglich das Phong-Modell verwendet werden, welches allerdings nicht optimal ist. Andere zwar komplexere aber auch deutlich bessere Beleuchtungsmodelle wie

das Cook-Torrance-Modell oder das Schlick-Modell könnten ohne große Umstände in das Path Tracing integriert werden. Dies würde zu einer einfacheren Materialdefinition und einer realistischeren Materialdarstellung führen.

14 Fazit

Durch die stetig zunehmende Rechenleistung von Computern, besonders von GPUs, ist Path Tracing in Echtzeit heute keine Wunschvorstellung mehr, sondern praktisch möglich. Durch die immer größeren Möglichkeiten wird die physikalisch korrekte Lichtsimulation in wenigen Millisekunden in der Computergrafik immer relevanter. In dieser Arbeit wurde gezeigt, dass globale Beleuchtung und realistische Schatten berechnet werden können, ohne auf diese Echtzeitfähigkeit verzichten oder Einbußen bezüglich der visuellen Qualität der Ergebnisbilder hinnehmen zu müssen. Die Frage, ob Path Tracing und somit die globale Lichtsimulation mit dem Line Space in Echtzeit möglich ist, kann also eindeutig mit „ja" beantwortet werden.

Der Line Space als Beschleunigungsdatenstruktur ist dabei für die Echtzeitfähigkeit von entscheidender Bedeutung. Mit ihm konnte das Path Tracing weitgehend szenenunabhängig sehr stark beschleunigt werden, wodurch zum Teil Frame-Raten im dreistelligen Bereich erreicht werden konnten. Besonders die Flexibilität der im Line Space gespeicherten Daten und die Möglichkeit, ihn mit anderen Datenstrukturen kombinieren zu können machen den Line Space zu einer sehr lohnenswerten und breit anwendbaren Datenstruktur.

Die Nachteile des Line Spaces, wie die langen Generierungszeiten, der hohe Speicherverbrauch und die Eigenschaft, nur auf statische Szenen anwendbar zu sein, sollten natürlich nicht ignoriert werden. Allerdings könnten diese Probleme dank der aktuellen Entwicklung der Hardware und vor allem durch die zur Zeit weiter erforschten Verfahren in Verbindung mit dem Line Space bald der Vergangenheit angehören.

Insgesamt ist der Line Space also als eine gute und wichtige Beschleunigungsdatenstruktur für Strahlverfolgungstechniken zu bewerten. Gerade in Kombination mit Path Tracing ist er sehr gut für die Beschleunigung der Berechnung von globaler Beleuchtung geeignet. Durch die stetige Weiterentwicklung des Verfahrens könnte der Line Space in naher Zukunft eine der besten und schnellsten Beschleunigungsdatenstrukturen für Ray bzw. Path Tracing sein.

Danksagung

Vielen Dank an Prof. Dr. Stefan Müller, Kevin Keul und Nicolas Klee für die tolle Zusammenarbeit, Motivation und Unterstützung während der gesamten Erarbeitungszeit dieser Bachelorarbeit. Die gemeinsame Weiterentwicklung des Line Spaces hat sehr viel Spaß gemacht.

Außerdem möchte ich allen Leuten danken, die mir durch Korrekturlesen geholfen haben, (hoffentlich) alle Fehler zu beseitigen.

Literatur

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[AW+87] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.

[BB84] L. S. Brotman and N. I. Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, 4(10):5–14, Oct 1984.

[DS97] George Drettakis and François X Sillion. Interactive update of global illumination using a line-space hierarchy. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 57–64. ACM Press/Addison-Wesley Publishing Co., 1997.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FI85] Akira Fujimoto and Kansei Iwata. *Accelerated Ray Tracing*, pages 41–65. Springer Japan, Tokyo, 1985.

[GS87] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, May 1987.

[ICG86] David S Immel, Michael F Cohen, and Donald P Greenberg. A radiosity method for non-diffuse environments. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 133–142. ACM, 1986.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[Kle16] Nicolas Klee. Darstellung von Halbschatten durch Verwendung des Cascaded Linespace-Verfahrens, 2016.

[KML16] Kevin Keul, Stefan Müller, and Paul Lemke. Accelerating Spatial Data Structures in Ray Tracing through Precomputed Line Space Visibility. In *WSCG 2016 - 24th WSCG Conference on Computer Graphics, Visualization and Computer Vision 2016*. WSCG, 2016.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1969.

[KS97] K. Klimaszewski and T. W. Sederberg. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications*, 17(1):42–51, Jan 1997.

[L'E96] Pierre L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Math. Comput.*, 65(213):203–213, January 1996.

[Man16] Jalal Eddine El Mansouri. Rendering 'Rainbow Six | Siege'. http://www.gdcvault.com/play/1022990/Rendering-Rainbow-Six-Siege, 2016. Zugriff: 21.02.2017.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[Ngu07] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann, 2004.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[PJH16] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[Pri15] Lutz Priese. *Computer Vision: Einführung in die Verarbeitung und Analyse digitaler Bilder*. Springer-Verlag, 2015.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. Cambridge University Press, 1992.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[VL11] Toms Vulfs and Jimmy Liikala. Monte carlo path tracing. *TNCG15 - Advanced Global Illumination and Rendering*, April 2011.

[ZCEP07] Long Zhang, Wei Chen, David S. Ebert, and Qunsheng Peng. Conservative voxelization. *The Visual Computer*, 23(9):783–792, 2007.