UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Light-Injection und Global Illumination mittels GPU und der Linespace Datenstruktur

Bachelorarbeit

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

vorgelegt von

Alexander Maximilian Nilles

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

Koblenz, im November 2017

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

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

Koblenz, 15. November 2017 .....
(Ort, Datum) (Unterschrift)

Zusammenfassung

In dieser Arbeit werden zwei Verfahren zur Berechnung der globalen Beleuchtung vorgestellt. Das Erste ist eine Erweiterung von *Reflective Shadow-Maps* um einen Schattentest, womit Verdeckungsbehandlung erreicht wird. Das zweite Verfahren ist ein neuer, auf Light-Injection basierender, bidirektionaler Ansatz. Dabei werden Strahlen aus Sicht der Lichtquelle verfolgt und in der Linespace Datenstruktur in Schächten gespeichert, die eine Diskretisierung der Raumrichtungen darstellen. Die Linespaces sind dabei in ein Uniform Grid eingebettet. Beim Auslesen der vorberechneten indirekten Beleuchtung sind im Idealfall keine Traversierung der Datenstruktur und keine weitere Strahlverfolgung mehr notwendig. Damit wird eine Varianzreduzierung und eine schnellere Berechnung im Vergleich zu Pathtracing erzielt, wobei sich insbesondere Vorteile in stark indirekt beleuchteten Bereichen und bei Glas ergeben. Die Berechnung der globalen Beleuchtung ist allerdings approximativ und führt zu sichtbaren Artefakten.

Abstract

This thesis presents two methods for the computation of global illumination. The first is an extension of *Reflective Shadow Maps* with an additional shadow test in order to handle occlusion. The second method is a novel, bidirectional Light-Injection approach. Rays originating from the light source are traced through the scene and stored inside the shafts of the Linespace datastructure. These shafts are a discretization of the possible spatial directions. The Linespaces are embedded in a Uniform Grid. When retrieving this pre-calculated lightning information no traversal of datastructures and no additional indirection is necessary in the best-case scenario. This reduces computation time and variance compared to Pathtracing. Areas that are mostly lit indirectly and glas profit the most from this. However, the result is only approximative in nature and produces visible artifacts.

Inhaltsverzeichnis

Teil I
Einleitung

Die Berechnung realistischer Bilder ist eine zentrale Fragestellung in der Computergrafik. Seit den Anfängen in den 1950er Jahren hat sich die Hardware rasant entwickelt, womit Ansätze, die direkt auf der Physik basieren – wie z. B. Raytracing und Pathtracing – mittlerweile echtzeitfähig sind.

Eine große Herausforderung stellt dabei allerdings immer noch die globale Beleuchtung dar, die unter anderem durch diffuse Materialien entsteht. Die aktuellen Ansätze weisen meist eine hohe Varianz auf, die zu einem Bildrauschen führt. Um rauschfreie Bilder zu erzeugen, sind weiterhin sehr lange Berechnungszeiten notwendig. Alternativ dazu werden zum Beispiel in Computerspielen starke Approximationen verwendet, die weiterhin echtzeitfähig sind.

Die globale Beleuchtung ist auch das Thema in dieser Bachelorarbeit. Der Fokus liegt dabei auf GPU-basierten Light-Injection Methoden und deren Kombination mit der Linespace Datenstruktur, die ein aktuelles Forschungsgebiet der AG Computergraphik ist. Dabei soll insbesondere die Varianz im Vergleich mit traditionellem Pathtracing reduziert und die Berechnung beschleunigt werden.

Im Folgenden werden in dieser Ausarbeitung zunächst die Grundlagen des Raytracings und der globalen Beleuchtung mittels Pathtracing, sowie der relevanten Beschleunigungsdatenstrukturen – insbesondere der Linespace – erläutert. Anschließend werden zwei Verfahren vorgestellt, konzeptionell erläutert und deren Implementation auf der GPU beschrieben. Diese Verfahren werden dann hinsichtlich ihrer Performance und der visuellen Ergebnisse anhand der Zielsetzung evaluiert, um die Frage zu beantworten, ob die Varianz wirklich reduziert wird und die Berechnung schneller ist. Dabei liegt insbesondere der Fokus auf Artefakten, die durch die vorgestellten Verfahren entstehen.

Teil II
Grundlagen

1 Raytracing

1.1 Grundprinzip

Das Grundprinzip des Raytracings ist die Verfolgung von Strahlen durch eine Szene. Trifft ein Strahl auf ein Objekt, erzeugt dies neue Strahlen, die ebenfalls weiterverfolgt werden. Eine Folge solcher zusammenhängender Strahlen wird Pfad genannt. Ein Pfad beginnt also am Ursprung seines Strahls. Dabei gibt es zwei verschiedene Ansätze: *Forward Raytracing* verfolgt Pfade mit Ursprung an der Lichtquelle. Wenn ein solcher Pfad in einem Schnittpunkt mit der Bildebene endet, dann ist das verfolgte Licht für die Kamera sichtbar. Die Wahrscheinlichkeit, dass die Bildebene getroffen wird, ist allerdings sehr gering. Deshalb wurde der zweite Ansatz, genannt *Backward Raytracing*, eingeführt. Dabei beginnen die Pfade stattdessen an der Kamera und werden bis zu einem Schnittpunkt mit der Lichtquelle verfolgt.

Die Grundoperation des Raytracings ist also das Berechnen des vordersten Schnittpunktes eines Strahls mit der Szene. *Backward Raytracing* kann außerdem um sogenannte Schattentests erweitert werden. Diese ermitteln für jeden Schnittpunkt zusätzlich, ob er von der Lichtquelle aus sichtbar ist. Wenn ja, wird eine direkte Beleuchtung durchgeführt. Dadurch kann ein Pfad auch ein sichtbares Ergebnis liefern ohne in einem Schnittpunkt mit der Lichtquelle zu enden. Der Unterschied zwischen einem Schattentest und dem Berechnen des vordersten Schnittpunktes ist dabei zum einen die Beschränkung der Distanz (nur bis zur Lichtquelle), zum anderen kann ein Schattentest abgebrochen werden, sobald *irgendein* Schnittpunkt gefunden wird [PJH16, Kap. 1].

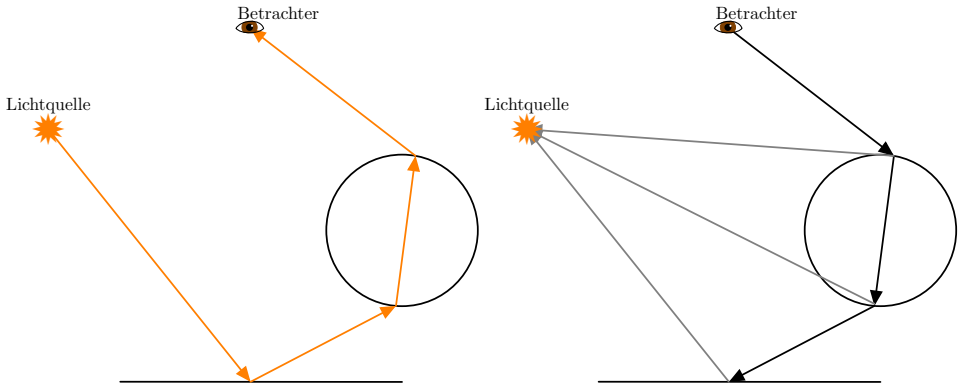


Abbildung 1: Links: Forward Raytracing von der Lichtquelle aus. Rechts: Backward Raytracing von der Kamera aus, mit Schattentests (grau).

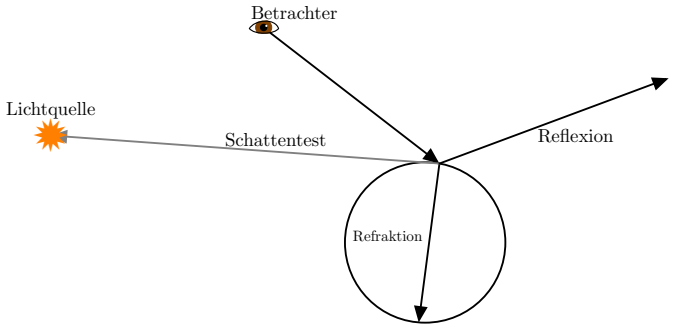


Abbildung 2: Whitted Raytracing. Ein Strahl spaltet sich rekursiv in einen reflektierten und refraktierten Strahl sowie einen Schattentest auf.

1.2 Raytracing nach Whitted

Whitted-Raytracing, auch rekursives Raytracing genannt, wurde von Turner Whitted in [Whi79] beschrieben. Dabei werden für jeden Pixel Pfade ausgehend von der Kamera verfolgt. An einem Schnittpunkt spaltet sich ein Pfad je nach Material in einen reflektierten und einen refraktierten Strahl, sowie einen Schattentest pro Lichtquelle auf. Dadurch ergibt sich insgesamt ein Baum aus Pfaden. Die Rekursion muss irgendwann abgebrochen werden, womit sich für jeden Baum eine maximale Tiefe ergibt.

Mit dieser Methode kann indirekte Beleuchtung, die sich insbesondere durch diffuse Materialien ergibt, nicht korrekt berechnet werden, denn ein Pfad kann sich immer nur in endlich viele Unterpfade aufspalten. Bei einem diffusen Material existieren allerdings unendlich viele mögliche Unterpfade.

2 Beschleunigungsdatenstrukturen

Die Berechnung des vordersten Schnittpunktes eines Strahls mit einer Szene erfordert einen Schnittpunkttest mit allen Primitiven in der Szene. Die Laufzeit ist also ein $O(n)$ Problem und steigt damit linear mit der Szenenkomplexität an. Da Szenen sehr komplex werden können, stellt dies ein großes Problem für die Laufzeit dar. Ziel einer Beschleunigungsdatenstruktur ist es, die notwendige Anzahl an Schnittpunkttests zu reduzieren. Im Idealfall reduziert man dabei sogar die Laufzeitkomplexität auf beispielsweise $O(\log n)$.

2.1 Uniform Grid

Das Uniform Grid ist eine sehr simple Datenstruktur, die zuerst von Fujimoto et al. [FTI86] beschrieben wurde. Dabei wird die Szene in ein reguläres, dreidimensionales Gitter unterteilt. Jede Zelle dieses Gitters wird *Voxel* genannt. Für jedes Voxel wird eine Liste mit allen Primitiven, die ganz oder teilweise darin liegen, angelegt (die *Kandidatenliste*).

Um einen Strahl mit der Szene zu schneiden, wird er zunächst mit den Voxeln geschnitten. Die Voxel werden dann in der Reihenfolge betrachtet, in der sie vom Strahl geschnitten werden. Nun wird für jedes dieser Voxel der vorderste Schnittpunkt mit seiner Kandidatenliste bestimmt. Existiert ein solcher Schnittpunkt, ist dies auch der vorderste Schnittpunkt für den Strahl und die restlichen Voxel müssen nicht mehr getestet werden.

Die optimale Auflösung des Uniform Grids ist dabei szenenabhängig. Es gibt zwar Formeln, die in vielen Fällen für eine Szene eine gute Auflösung ermitteln, allerdings kann eine manuelle Anpassung dennoch notwendig sein [TS+05].

2.2 Linespace

Der Linespace wurde 2016 von K. Keul, P. Lemke und S. Müller in [KLM16] beschrieben und ist eine sehr neue Datenstruktur. Es handelt sich dabei um ein Raumquader, dessen Seitenflächen regulär in ein Gitter unterteilt sind. Eine Zelle auf einem dieser Gitter wird *Patch* genannt.

Ohne Beschränkung der Allgemeinheit wird im Folgenden der Einfachheit halber ein Linespace als Würfel betrachtet, wobei jede Seite quadratisch mit der selben Auflösung N unterteilt ist. Das bedeutet also, dass jede Seitenfläche aus N^2 Patches mit den gleichen Ausmaßen besteht. Insgesamt gibt es also $6N^2$ Patches pro Linespace.

Man betrachtet nun die Verbindung eines jeden Patches (dem Start-Patch) zu je allen anderen Patches (den End-Patches) auf den fünf anderen Seitenflächen. Eine solche Verbindung wird *Schacht* genannt. Im zweidimensionalen Raum bildet ein Schacht jeweils ein Viereck, bzw. ein Dreieck für Schächte zwischen angrenzenden Patches in den Ecken. Die Situation im dreidimensionalen ist komplexer. Versucht man auf naive Weise jeweils vier Punkte zu einer Seitenfläche des Schachts zu verbinden, um insgesamt sechs Flächen zu erhalten, funktioniert dies nicht immer. Vier Punkte im 3D ergeben nicht notwendigerweise eine Ebene. Man muss also manche dieser Verbindungen durch ein oder zwei Dreiecke ersetzen, um die korrekte konvexe Hülle zu erhalten. Insgesamt gibt es vier solcher Seiten, bei denen dies passieren kann. Die anderen beiden Seiten sind die beteiligten Patches selbst, die per Definition bereits in einer Ebene liegen. Die konvexe Hülle eines Schachtes kann also mit maximal zehn Ebenen beschrieben werden.

Die beschriebene Konstruktion führt zu $5 \cdot 6 \cdot N^2 \cdot N^2$, also $30N^4$ Schächten pro Linespace. Dabei zählt man allerdings jeden Schacht doppelt, nämlich einmal für das Start-Patch und einmal für das End-Patch. Es sind also tatsächlich nur $15N^4$ Schächte.

Für jeden Schacht kann man nun Informationen abspeichern. Eine Möglichkeit ist das Speichern einer Kandidatenliste für jeden Schacht, wofür ein Clipping der Geometrie an den Ebenen der konvexen Hülle des Schachts

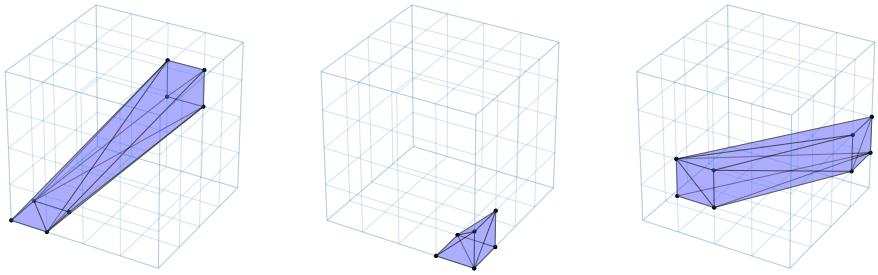


Abbildung 3: Visualisierung eines Linespaces mit $N = 4$ und der triangulierten konvexen Hülle verschiedener Schächte.

notwendig ist. Eine andere Variante speichert stattdessen nur eine 1-Bit Information, nämlich, ob der Schacht Geometrie enthält, oder nicht. Diese Variante wird auch als *Linespace mit vorberechneter Sichtbarkeit* bezeichnet. In dieser Arbeit wird einerseits die Kandidatenliste verwendet, andererseits werden in den Schächten Beleuchtungsstärken gespeichert, wobei je ein Eintrag für die beiden Hauptrichtungen durch den Schacht gespeichert wird.

Der Linespace allein reicht als Beschleunigungsdatenstruktur in den meisten Fällen nicht aus. Stattdessen wird er mit anderen Datenstrukturen kombiniert. Keul et. al verwenden in [KLM16] und [KKM17] einen *N-tree*, auch als *Recursive Grid* bekannt. Dabei handelt es sich um eine Verallgemeinerung eines *Octrees*, die eine Unterteilung von n^3 anstelle von der festen Unterteilung in 2^3 Unterknoten pro Knoten vornimmt. Jeder Knoten dieses N-Trees enthält dabei einen Linespace mit vorberechneter Sichtbarkeit, wodurch eine effizientere Traversierung des N-Trees möglich wird. Hierbei handelt es sich also – im Gegensatz zu Uniform Grids – um eine hierarchische Datenstruktur.

Diese Arbeit verfolgt einen anderen Ansatz als Keul et al. und bettet den Linespace in ein Uniform Grid ein. Es wird also pro Voxel – genauer pro gefülltem Voxel – ein Linespace angelegt.

3 Globale Beleuchtung

3.1 Die Rendergleichung

Die Rendergleichung wurde zeitgleich von Kajiya [Kaj86] und Immel et al. [ICG86] 1986 formalisiert. Mit dieser Gleichung wird die globale Beleuchtung beschrieben. Diese wird als Strahldichte L, die ein Oberflächenpunkt x in Richtung $\vec{\omega}$ abgibt, beschrieben. Dazu wird das von x selbst emittierte Licht L_e mit dem Licht, das von allen Raumrichtungen $\vec{\omega}'$ der Hemisphäre an x ankommt und in Richtung $\vec{\omega}$ reflektiert wird, summiert. Die Gleichung

lautet wie folgt:

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

θ ist dabei der Winkel zwischen $\vec{\omega}'$ und der Normalen $\vec{n}$ am Oberflächenpunkt x. Es gilt also $\cos\theta = \langle \vec{\omega}', \vec{n} \rangle$ (Skalarprodukt).

Betrachtet man das Integral ohne f_r, lässt es sich in die Beleuchtungsstärke E umformen [PJH16, Kap. 5]:

$$\begin{aligned} L &= \frac{d^2\Phi}{dA \cdot \cos\theta \cdot d\vec{\omega}'} \\ d^2\Phi &= L \cdot dA \cdot \cos\theta \cdot d\vec{\omega}' \\ d\Phi &= dA \cdot \int_\Omega L \cdot \cos\theta \cdot d\vec{\omega}' \\ E = \frac{d\Phi}{dA} &= \int_\Omega L \cdot \cos\theta \cdot d\vec{\omega}' \end{aligned}$$

Die Rendergleichung kann also auch durch Skalierung der Beleuchtungsstärke mit f_r – der *Bidirectional Reflectance Distribution Function* (BRDF, dt. Bidirektionale Reflektanzverteilungsfunktion) – berechnet werden. Die BRDF beschreibt dabei das Material am Oberflächenpunkt, indem sie angibt, wie viel Licht aus Richtung $\vec{\omega}'$ in Richtung $\vec{\omega}$ reflektiert wird.

Die ursprüngliche Rendergleichung, wie hier beschrieben, beschreibt nur Reflexionen. Um auch transmittiertes Licht zu berechnen, kann die Gleichung wie folgt verändert werden:

$$L(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_{\mathcal{S}^2} f(x, \vec{\omega}', \vec{\omega}) \cdot L(x, \vec{\omega}') \cdot |\cos\theta| \cdot d\vec{\omega}' \tag{2}$$

Die Integration erfolgt also nicht mehr über die Hemisphäre Ω, sondern über die ganze Sphäre $\mathcal{S}^2$. Anstelle der BRDF tritt dann die BSDF (*Bidirectional Scattering Distribution Function*, dt. Bidirektionale Streuungsverteilungsfunktion), die sowohl Reflektanz als auch Transmission beschreibt. Sie setzt sich also zusammen aus einer BRDF für die Reflektanz und einer BTDF (*Bidirectional Transmittance Distribution Function*, dt. Bidirektionale Transmissionsverteilungsfunktion) für die Transmission [PJH16, Kap. 1 u. 5].

Mit dieser Erweiterung sind noch immer nicht alle Beleuchtungseffekte abgedeckt. Man kann die Rendergleichung beispielsweise noch um Volumenstreuung erweitern. Dadurch wird aus dem 2D Integral allerdings ein 4D Integral (zusätzliche Integration über die Fläche) und es wird wieder eine neue Funktion benötigt. Diese nennt man BSSRDF (*Bidirectional Scattering Surface Reflectance Distribution Function*) und sie kann alles beschreiben, was sich mit einer BRDF beschreiben lässt, sowie zusätzlich das Volumenstreuungsverhalten [PJH16, Kap. 5].

3.2 Das Phong-Beleuchtungsmodell

Das Beleuchtungsmodell nach Phong [Pho75] ist ein bis heute benutztes Beleuchtungsmodell in der Computergrafik. In der ursprünglichen Fassung beschreibt es keine globale Beleuchtung – die Rendergleichung existierte zu diesem Zeitpunkt noch nicht – es lässt sich allerdings zu einer BRDF erweitern, sodass es in der Rendergleichung verwendet werden kann. Dies findet in dieser Arbeit Anwendung, wobei diese Phong-BRDF mit perfekter Reflexion und Refraktion zu einer BSDF kombiniert wird.

Phong beschreibt die Beleuchtung mit einem diffusen und spekularen Beleuchtungsterm, wobei zusätzlich ein konstanter ambienter Term verwendet werden kann. Ein Material wird dann mit einem diffusen Reflexionskoeffizienten $k_d \in [0, 1]$ sowie einem spekularen Reflexionskoeffizienten $k_s \in [0, 1]$ beschrieben, wobei $k_d + k_s \leq 1$.

Die beiden Terme sind abhängig von der Lichtquelle, die Gesamtbeleuchtung an einem Punkt ergibt sich dann also aus der Summe der diffusen und spekularen Terme über alle Lichtquellen.

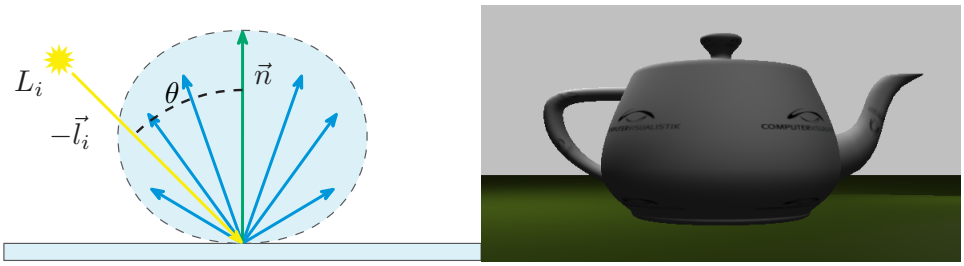


Abbildung 4: Diffuse Beleuchtung nach Phong. Links: Schematische Darstellung der diffusen Beleuchtung. Rechts: Vollständig diffus beleuchteter Teapot ($k_d = 1$).

3.2.1 Diffuser Beleuchtungsterm

Der diffuse Beleuchtungsterm gewichtet das einfallende Licht mit dem Cosinus des Winkels zwischen Oberflächennormale $\vec{n}$ und Lichtvektor $\vec{l}$. Dies korrespondiert zur Lichtstärke eines Lambert-Strahlers, die ebenfalls mit dem Cosinus abnimmt und dadurch eine konstante Leuchtdichte erreicht. Deshalb spricht man auch vom *Lambert-Term*. Ein diffuses Material erscheint also aus allen Blickrichtungen gleich hell. Der diffuse Term aller Lichtquellen ergibt sich also wie folgt, wobei $\text{col}(L_i)$ die Farbe der Lichtquelle ist:

$$L_d(x) = \sum_{\text{Lichter } L_i} (\text{col}(L_i) \cdot k_d \cdot \cos\theta) = \sum_{\text{Lichter } L_i} \left(\text{col}(L_i) \cdot k_d \cdot \langle \vec{l}_i, \vec{n} \rangle\right) \quad (3)$$

Der Term wird dann zusätzlich mit der diffusen Materialfarbe komponentenweise multipliziert. Diese Multiplikation bezeichnet man als Hadamard-Produkt ($x \circ y$). In dieser Arbeit wird auf diese Schreibweise verzichtet.

Stattdessen meint $x \cdot y$ das Hadamard-Produkt, wenn x und y die gleiche Dimension haben.

3.2.2 Spekularer Beleuchtungsterm

Im Gegensatz zum diffusen Term ist der spekulare Term abhängig von der Blickrichtung $\vec{v}$. Er modelliert eine nicht perfekte Reflexion (Glanzreflexion), die um eine Keulenform vom perfekt reflektierten Lichtvektor $\vec{r}$ abweicht. Wie stark die Abweichung ist, wird dabei über die Glanzzahl bzw. den Glanzexponenten n festgelegt:

$$L_s(x) = \sum_{\text{Lichter } L_i} (\text{col}(L_i) \cdot k_s \cdot \cos^n \varphi) = \sum_{\text{Lichter } L_i} (\text{col}(L_i) \cdot k_s \cdot \langle \vec{v}, \vec{r}_i \rangle^n) \quad (4)$$

Auch für den spekularen Term findet eine komponentenweise Multiplikation statt, diesmal mit der spekularen Materialfarbe.

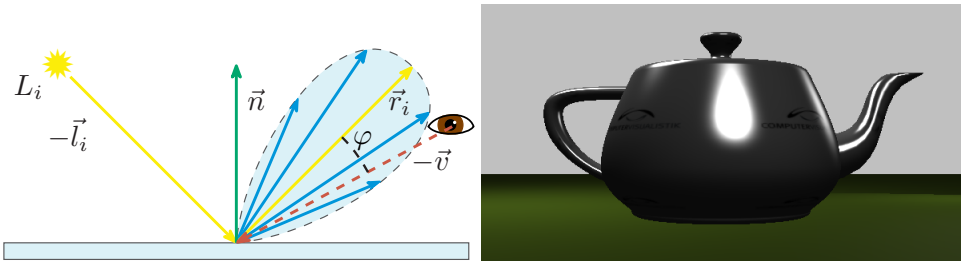


Abbildung 5: Spekulare Beleuchtung nach Phong. Links: Schematische Darstellung der spekularen Beleuchtung. Rechts: Halb diffus, halb spekular beleuchteter Teapot ($k_d = \frac{1}{2}, k_s = \frac{1}{2}, n = 40$).

3.2.3 Alternativen

Das Phong-Modell ist sehr simpel und lässt sich vergleichsweise schnell berechnen. Für Photorealismus ist es allerdings unbrauchbar, da sich nur sehr einfache Materialien beschreiben lassen. Des Weiteren ist die Phong-BRDF nicht physikalisch korrekt, denn sie missachtet den Energieerhaltungssatz [DF97]. Beispiele für Modelle, die komplexere Materialien beschreiben können und sich näher an der Physik orientieren, sind *Cook-Torrance* (auch als *Torrance-Sparrow* bekannt), *Oren-Nayar* oder das *Schlick*-Modell. Diese modellieren dabei Oberflächen, die aus Mikrofacetten bestehen und/oder machen Gebrauch von den Fresnelschen Formeln [PJH16, Kap. 9].

3.3 Pathtracing

Pathtracing basiert auf dem zuvor beschriebenen Raytracing. Es ist ein Ansatz, mit dem versucht wird, die Integrale der Rendergleichung mithilfe von

numerischen Methoden zu lösen. Bevor das eigentliche Pathtracing erklärt wird, werden im Folgenden zunächst die dafür notwendigen Grundlagen beschrieben.

3.3.1 Wahrscheinlichkeitsdichtefunktion

Mit einer Wahrscheinlichkeitsdichtefunktion (kurz PDF aus engl. Probability Density Function) wird die Verteilung von Zufallsvariablen X_i beschrieben. $p(x)$ gibt dann an, mit welcher relativen Wahrscheinlichkeit X_i den Wert x annimmt. Das Integral einer PDF über ihrem Definitionsbereich muss also immer 1, also 100%, ergeben. Für gleichverteilte Zufallszahlen ist die PDF konstant [PJH16, Kap. 13].

3.3.2 Monte-Carlo Estimator

Der Monte-Carlo Estimator zählt zu den *Monte-Carlo-Methoden*. Bei diesen handelt es sich um Verfahren aus der Stochastik, die im Allgemeinen dazu dienen, analytisch nicht oder sehr schwer lösbare Probleme auf Basis des *Gesetzes der großen Zahlen* numerisch zu lösen. Dies geschieht, indem man sehr viele Zufallsexperimente durchführt. Im Fall von Rechnern verwendet man stattdessen generierte Pseudozufallszahlen. Ein sehr bekanntes Beispiel für diese Methoden ist die Annäherung von π.

Bei dem Monte-Carlo Estimator im Speziellen geht es um die Lösung von Integralen, im Falle des Pathtracings also um das Lösen der Rendergleichung. Hierzu verwendet man gleichverteilte Zufallsvariablen $X_i \in [a, b]$. Der Erwartungswert $E[F_N]$ des Estimators

$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i) \tag{5}$$

ist dann gleich dem Integral $\int_a^b f(x)dx$. Dies gilt allerdings nur, wenn die Zufallsvariable X_i eine konstante PDF von $\frac{1}{b-a}$ im Intervall $[a, b]$ aufweist. Man kann den Estimator wie folgt auf nicht gleichverteilte Zufallsvariablen erweitern [PJH16, Kap. 13]:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)} \tag{6}$$

Ein Monte-Carlo Estimator führt also eine Abtastung des Integrals durch, wobei ein solcher Abtastwert auch *Sample* genannt wird. Wie viele Samples notwendig sind, bis sich der Erwartungswert dem Integral annähert, wird *Konvergenz* genannt. Die Varianz des Estimators spiegelt sich im Bild als für Pathtracing charakteristischem Rauschen wieder, wenn zu wenig Samples genommen wurden.

Mit dem Monte Carlo Estimator wird die Rendergleichung zu:

$$L(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \frac{1}{N} \sum_{i=1}^{N} \left(\frac{f_r(x, \vec{\omega}_i', \vec{\omega}) \cdot L(x, \vec{\omega}_i') \cdot \cos\theta}{p(\vec{\omega}_i')} \right) \tag{7}$$

3.3.3 Importance Sampling

Wie im vorherigen Abschnitt gezeigt, kann man die PDF für die Zufallsvariable X_i im Monte-Carlo Estimator frei wählen. Mit einem Blick auf die Rendergleichung aus Abschnitt 3.1 fällt auf, dass in dieser die BRDF f_r selbst eine Verteilung beschreibt. Aufgrund des Energieerhaltungssatzes muss das Integral der BRDF auch 1 ergeben, wir können die BRDF selbst also als PDF betrachten. Wählt man die PDF der Zufallsvariable nun so, dass sie der BRDF entspricht, wird die BRDF im Monte-Carlo Estimator aus der Gleichung gekürzt. Dieses Prinzip nennt man *Importance Sampling*. Die PDF muss nicht zwangsweise exakt der BRDF entsprechen, bereits eine Annäherung bringt Vorteile. Insgesamt reduziert Importance Sampling die Varianz des Monte-Carlo Estimators, was zu einer schnelleren Konvergenz führt [PJH16, Kap. 13]. Mit perfektem Importance Sampling lässt sich die Render-Gleichung weiter vereinfachen:

$$L(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \frac{1}{N} \sum_{i=1}^{N} \left(L(x, \vec{\omega}_i') \cdot \cos\theta \right) \tag{8}$$

3.3.4 Russisches Roulette

Mit *russischem Roulette* ist die zufällige Auswahl einer Möglichkeit aus einer Menge von Möglichkeiten gemeint. Dies kann beispielsweise die Auswahl des Folgestrahls beim Pathtracing sein, oder die Entscheidung, ob ein Strahl weiterverfolgt wird oder nicht. Bei einer Phong-BRDF kann man z. B. abhängig von k_d und k_s entscheiden, ob man einen diffus reflektierten oder spekular reflektierten Strahl weiterverfolgt. Ebenso kann man abhängig vom Beitrag eines Strahls zum Gesamtergebnis entscheiden, ob dieser weiterverfolgt wird oder nicht. Das Verfahren findet sehr oft Anwendung in Monte-Carlo-Methoden.

3.3.5 Pathtracing

Das eigentliche Pathtracing löst die Rendergleichung nun mithilfe des Monte-Carlo Estimators (siehe Gleichung 7). Hierzu verfolgt man wie beim Raytracing Pfade, generiert aber für jeden Schnittpunkt nur einen Folgestrahl sowie einen Schattentest, falls das Material beleuchtet wird. Es wird also immer nur ein Licht- bzw. Kamerapfad je Pixel verfolgt, anstelle eines ganzen Pfadbaumes wie bei Whitted-Raytracing. Damit ist ein Durchgang bei Pathtracing schneller berechnet als bei Whitted-Raytracing. Die Folgestrahlen werden hier nun zufällig ausgewählt, wenn mehrere möglich sind. Dies ist

beispielsweise bei diffusen Materialien der Fall. Dadurch ist es mit Pathtracing möglich, globale Beleuchtung zu berechnen, die durch diffuse Reflexionen oder ähnliche Materialien entsteht. Im Sinne des Importance Samplings versucht man dabei, den neuen Strahl möglichst entsprechend der BRDF des Materials auszuwählen. Ein Pfad ist dabei ein Sample des Monte-Carlo Estimators. Pro Durchgang wird also ein neuer Pfad pro Pixel verfolgt, die akkumulierten Pfade müssen dann durch die Sampleanzahl N geteilt werden [PJH16, Kap. 14].

Analog zum Raytracing unterscheidet man zwischen Forward und Backward Pathtracing, wobei letzteres aus den gleichen Gründen wie in Abschnitt 1 beschrieben in der Regel eher Anwendung findet.

Da Pathtracing stochastisch vorgeht, sind je nach Materialien sehr viele solcher Samples notwendig, bis das aufgrund der Varianz vorhandene Rauschen im Bild verschwindet und das Verfahren konvergiert. Dies muss in Kauf genommen werden, da sich die Rendergleichung im Allgemeinen nicht analytisch lösen lässt [PJH16, Kap. 1].

3.4 Bidirektionale Ansätze

Die bisher beschriebenen Ansätze verfolgen Strahlen entweder beginnend bei der Kamera oder den Lichtquellen. Eine weitere Möglichkeit stellen bidirektionale Verfahren dar, die beide Varianten kombinieren. Dazu zählen sowohl die später vorgestellten *Reflective Shadow-Maps*, als auch das in dieser Arbeit vorgestellte neue Verfahren der *Linespace Injection*. Im Folgenden werden noch zwei weitere bidirektionale Ansätze beschrieben. Der Vorteil von bidirektionalen Ansätzen zeigt sich vor allem in Szenen, in denen ein großer Teil der Beleuchtung nur indirekt erfolgt. Auch Kaustiken zählen zu den Beleuchtungseffekten, die von bidirektionalen Ansätzen profitieren. Backward Pathtracing allein weist bei solchen Szenen eine hohe Varianz auf, da es sehr unwahrscheinlich ist einen Pfad zu samplen, der einen direkt beleuchteten Schnittpunkt aufweist.

3.4.1 Bidirektionales Pathtracing

Beim *Bidirektionalen Pathtracing* werden in einem ersten Durchgang mit Forward Pathtracing Pfade von der Lichtquelle aus verfolgt und anschließend abgespeichert. Im zweiten Durchgang erfolgt dann Backward Pathtracing, wobei an jedem Schnittpunkt ein Lichtpfad ausgewählt wird und dieser mittels Schattentest mit allen Schnittpunkten dieses Pfades verbunden wird. Dadurch beleuchtet man die Schnittpunkte nicht nur mit direktem Licht, sondern auch mit indirekten Licht, was die Varianz für die zuvor beschriebenen indirekten Beleuchtungssituationen verbessert. Andererseits erhöht sich die Berechnungszeit, da sowohl Licht- als auch Kamerapfade verschickt werden und viele Verbindungen zwischen diesen Pfaden hergestellt werden – anstelle

nur einer Verbindung direkt zur Lichtquelle [PJH16, Kap. 16].

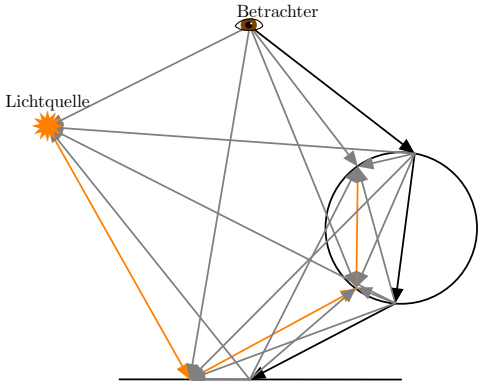


Abbildung 6: Bidirektionales Pathtracing. Alle Schnittpunkte des Kamerapfades werden mit allen Schnittpunkten des Lichtpfades (orange) mittels Schattentest (grau) verbunden.

3.4.2 Photon-Mapping

Im Gegensatz zu bidirektionalem Pathtracing verfolgt Photon-Mapping keine Pfade aus Sicht der Lichtquelle, sondern *Partikel*. Dieses Vorgehen wird im Allgemeinen *Particle Tracing* genannt. Im Falle des Photon-Mapping bezeichnet man die Partikel als *Photonen*. In einem ersten Pass werden also viele solcher Photonen von den Lichtquellen verschossen und durch die Szene verfolgt. An jedem Schnittpunkt, an dem Absorption stattfindet – also beispielsweise auf diffusen Materialien –, speichert man den absorbierten Anteil der Energie des Photons in einer Datenstruktur, der *Photon-Map*, ab.

Anschließend erfolgt Backward Raytracing oder Pathtracing. Perfekt reflektierte und refraktierte Strahlen werden wie gewohnt weiter verfolgt. Bei diffusen Materialien wird kein Folgestrahl ermittelt. Stattdessen zählt man die Photonen in einem bestimmten Radius und interpoliert damit die Beleuchtung an diesem Punkt. Alternativ kann man die Anzahl an Photonen festlegen und den dazu notwendigen Radius ermitteln. Aufgrund der Interpolation nennt man Photon-Mapping auch *biased*, im Gegensatz zu (Bidirektionalem) Pathtracing, was *unbiased* ist [PJH16, Kap. 16].

3.5 Reflective Shadow-Maps

Reflective Shadow-Maps (RSM) [DS05] sind eine Erweiterung von Shadow-Maps [Wil78], mit denen indirekte Beleuchtung in dynamischen Szenen plausibel berechnet werden kann. Das Verfahren behandelt jeden Pixel einer Shadow-Map als Lichtquelle. Die Summe dieser approximiert die erste Indirektion. Dachsbacher und Stamminger erreichen in ihrer Implementation interaktive Frameraten.

Zusätzlich zum Tiefenwert d_p eines Pixels p aus Sicht der Lichtquelle wird in einer RSM die Weltposition x_p, die Normale $\vec{n}_p$ sowie der reflektierte Strahlungsfluss Φ_p gespeichert. Letzterer berechnet sich aus dem Strahlungsfluss durch das Pixel, der abhängig von der verwendeten Lichtquelle ist, multipliziert mit dem Reflexionskoeffizienten der zugehörigen Oberfläche. Dabei werden alle Oberflächen als diffuse Reflektoren angenommen.

Die Pixel der RSM werden als Lichter behandelt, deren Helligkeit mit dem Strahlungsfluss Φ_p festgelegt ist. Die Abstrahlcharakteristik ist durch die Normale bestimmt. Damit ergibt sich die Strahlungsintensität I_p in Raumrichtung $\vec{\omega}$ als

$$I_p(\vec{\omega}) = \Phi_p \max\{0, \langle \vec{n}_p, \vec{\omega} \rangle\}, \text{wobei } p \text{ unendlich klein ist.} \tag{9}$$

Die Pixellichter strahlen also nur in den vorderen Halbraum ab. Damit kann die Bestrahlungsstärke eines Punktes x mit Normale $\vec{n}$ von Pixellicht p berechnet werden:

$$E_p(x, \vec{n}) = \Phi_p \frac{\max\{0, \langle \vec{n}_p, x - x_p \rangle\} \max\{0, \langle \vec{n}, x_p - x \rangle\}}{\|x - x_p\|^4} \tag{10}$$

Der Vektor vom Pixellicht zum Punkt wurde hier für ω in Gleichung 9 eingesetzt und das Ergebnis zusätzlich für die diffuse Beleuchtung mit dem Cosinus zwischen Normale und dem umgekehrten Vektor gewichtet, wobei vom hinteren Halbraum aus nicht beleuchtet wird. Weiterhin wird mit dem inversen der quadratischen Distanz gewichtet. Da in den beiden Skalarprodukten der Distanzvektor nicht normalisiert ist, findet bereits implizit eine Gewichtung mit der quadratischen Distanz statt. Deshalb wird in Gleichung 10 durch das Biquadrat der Distanz geteilt.

Summiert man die Beleuchtung eines Punktes durch alle Pixellichter in der RSM auf, erhält man die angenäherte indirekte Bestrahlungsstärke an diesem Punkt. Hierbei werden Schnittpunkte mit der Umgebung ignoriert, die Pixellichter scheinen also auch durch andere Geometrie hindurch.

Die Auswertung dieser Summe für alle Pixel einer RSM bedeutet für übliche Texturgrößen über eine Millionen Texturzugriffe je beleuchtetem Pixel, was für Echtzeitanwendungen zu viel ist. Dachsbacher et al. reduzieren deshalb diese Summe auf wesentlich weniger Pixellichter. Dabei treffen sie die Annahme, dass die Distanz zwischen Pixellicht und dem zu beleuchtenden Punkt in Texturkoordinaten (s, t) der RSM eine grobe Approximation der Distanz in Weltkoordinaten darstellt. Die Pixellichter werden deshalb zufällig in einem Radius um diesen Punkt ausgewählt, womit bevorzugt nahe gelegene Pixellichter relevant sind. Zusätzlich nimmt die Größe der Samples mit der quadratischen Distanz in Texturkoordinaten zu, weshalb die Samples mit dieser multipliziert werden müssen. Nachdem alle Samples akkumuliert wurden, werden sie mit der Summe dieser Distanzen normalisiert. Abbildung 7 gibt ein Beispiel für ein Sampling-Pattern.

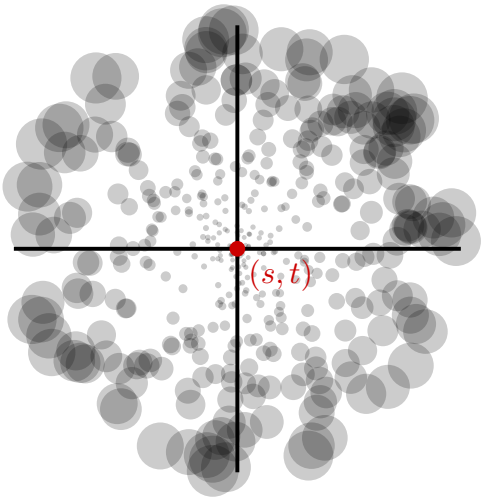


Abbildung 7: Beispiel eines Sampling-Patterns. Der Radius der Kreise nimmt vom Mittelpunkt aus zu und gibt die Gewichtung und Samplegröße an [Grafik rekonstruiert aus [DS05].].

Die Autoren berechnen in ihrer Implementation ein Sampling-Pattern vor und verwenden dieses für alle Berechnungen. Dadurch erreichen sie zeitliche Kohärenz, welche Flackern im Bild reduziert. Gleichzeitig sorgt die räumliche Kohärenz für Artefakte in Form von Streifenbildung bei niedriger Sampleanzahl.

Um das Verfahren schneller zu machen, nutzen Dachsbacher et al. Screen-Space Interpolation. Dabei wird die indirekte Beleuchtung zunächst nur für ein niedriger aufgelöstes Bild berechnet. Anschließend wird für jeden Pixel im höher aufgelösten Bild entschieden, ob die indirekte Beleuchtung aus der bereits berechneten interpoliert werden kann. Hierzu wird geprüft, wie ähnlich Position und Normale des Pixels im niedrig aufgelösten Bild mit dem in voller Auflösung sind. Bei zu großer Abweichung wird die indirekte Beleuchtung für dieses Pixel neu berechnet. Erst mit dieser Interpolation erreichten Dachsbacher et al. interaktive Frameraten.

Reflective Shadow-Maps berechnen indirekte Beleuchtung nur approximativ. Insbesondere wird Verdeckung vernachlässigt, was zu auffälligen Fehlern führen kann. Weiterhin kann nur die erste Indirektion berechnet werden. Dennoch liefert das Verfahren gute Ergebnisse für interaktive Anwendungen, in denen Photorealismus vernachlässigt werden kann. Die Autoren erwähnen außerdem, dass das Verfahren auf nicht diffuse Reflektoren erweitert werden kann. Dadurch werden zwar höhere Sampleanzahlen notwendig [DS05], allerdings steigen damit auch die Verwendungsmöglichkeiten des Verfahrens.

Teil III
Konzeption

4 Verfahren

4.1 RSM mit Schattentest

Die Ergebnisbilder aus [DS05] zeigen, dass Reflective Shadow-Maps die indirekte diffuse Beleuchtung approximieren können, ohne dabei ein Rauschen im Bild zu erzeugen. Dies macht das Verfahren attraktiv für eine schnellere, wenn auch ungenauere Lösung der Rendergleichung.

Ein Nachteil der Reflective Shadow-Maps ist die Nichtbehandlung von Verdeckung bei der Berechnung des indirekten Lichtes. Dieses Problem soll das hier vorgestellte Verfahren lösen, wobei keine Anforderung auf Echtzeitfähigkeit gestellt wird.

Um Verdeckung zu berücksichtigen, wird Gleichung 10 angepasst zu:

$$E_p(x,\vec{n}) = \begin{cases} \Phi_p \frac{\max\{0,\langle\vec{n}_p, x-x_p\rangle\}\max\{0,\langle\vec{n}, x_p-x\rangle\}}{\|x-x_p\|^4} & \text{, f. } \neg\,\text{intersect}(x,x_p) \\ 0 & \text{, f. intersect}(x,x_p) \end{cases} \tag{11}$$

Dabei berechnet intersect(x, x_p), ob ein Schnittpunkt zwischen Szenengeometrie und dem Strahl vom Punkt x nach x_p vorliegt.

Außerdem wird auf die im ursprünglichen Verfahren eingesetzte Screen-Space Interpolation zugunsten eines genaueren Ergebnisses verzichtet. Das Verfahren unterscheidet sich ansonsten nicht von den in Abschnitt 3.5 beschriebenen Reflective Shadow-Maps von Dachsbacher et al. [DS05].

4.2 Linespace Injection

Der Linespace diskretisiert Raumrichtungen in eine Anzahl von Schächten. Die Grundidee von Linespace Injection ist es, die indirekte Beleuchtung mit mehreren Indirektionen in diesen Schächten abzuspeichern. Dabei gibt es zwei Einträge pro Schacht, je einen pro Hauptrichtung. Die indirekte Beleuchtung eines Oberflächenpunktes aus einer Richtung kann dann mit dem im zugehörigen Schacht gespeicherten Wert approximiert werden. Das Verfahren gliedert sich also in einen Injection-Pass, der die indirekte Beleuchtung vorberechnet, sowie einen Sampling-Pass, der diese ausliest.

4.2.1 Datenstruktur

Die Basisdatenstruktur für das Verfahren ist der Linespace. Im Gegensatz zu Keul et al. in [KLM16] werden die Linespaces nicht in eine N-Tree Datenstruktur eingebettet, sondern in ein Uniform Grid. Mit anderen Worten wird also die Szene voxelisiert und anschließend ein Linespace pro Voxel erzeugt. Da

ein Oberflächenpunkt, dessen Beleuchtung mit den Schächten des Linespaces approximiert werden soll, immer selbst innerhalb der relevanten Schächte liegt, reicht es aus die Informationen nur für Schächte zu speichern, die Geometrie enthalten. Insbesondere muss auch kein Linespace für ein leeres Voxel aufgestellt werden. Pro nicht leerem Schacht existieren dann zwei Einträge, einer je Hauptrichtung. In diesen Einträgen wird eine Farbe und eine natürliche Zahl gespeichert. Die Zahl steht dabei dafür, wie oft im Injection-Pass etwas in diesem Schacht gespeichert wurde (im Folgenden Zähler genannt) und dient der späteren Normalisierung des eingetragenen Farbwertes. Die Menge aller solchen Einträge wird nachfolgend als Injectionliste bezeichnet.

Das hier vorgestellte Verfahren kommt ohne Traversierung der Datenstruktur beim Auslesen der indirekten Beleuchtung aus. Dennoch wird eine Datenstruktur für die Berechnung der Schnittpunkte sowohl im Injection-Pass als auch im Sampling-Pass benötigt. Die Wahl dieser Datenstruktur ist unabhängig vom Verfahren und kann frei erfolgen.

Speichert man zusätzlich zu den Einträgen für die indirekte Beleuchtung eine Kandidatenliste pro Schacht für die Schnittpunktberechnung, wie in Abschnitt 2.2 beschrieben, so kann man dies für eine Optimierung im Sampling-Pass verwenden. Dies ist aber nicht zwingend notwendig und kann entfallen, wenn Speicher gespart werden muss. Liegt eine Kandidatenliste vor, könnte man zusätzlich die Linespace Datenstruktur als Beschleunigungsdatenstruktur für die Schnittpunktberechnungen verwenden. Da aber ein Uniform Grid die Basis bildet, kann dies von Nachteil sein, da es bei der Verwendung von Uniform Grids eine szenenabhängige optimale Auflösung gibt [TS+05]. Im Gegensatz dazu ist es für das hier vorgestellte Verfahren von Vorteil, eine möglichst große Auflösung zu wählen, damit die Approximation nicht zu grob ist. Diese Auflösung könnte größer sein als die optimale Auflösung für die Traversierung, sie könnte allerdings aufgrund von Speicherlimitationen auch kleiner sein.

4.2.2 Injection-Pass

Im Injection-Pass wird die indirekte Beleuchtung ausgehend von potentiell mehreren Indirektionen vorberechnet und in den Schächten der Linespaces gespeichert. Hierzu wird Licht von den Lichtquellen aus nach einem ähnlichen Prinzip wie dem Forward Pathtracing durch die Szene verfolgt, allerdings ohne dabei Schnittpunkte mit der Bildebene zu ermitteln. Bei den abgespeicherten Werten handelt es sich um die Beleuchtungsstärke E. Diese Entscheidung wurde getroffen, da in der Rendering-Equation direkt über diese integriert wird und damit keine weiteren Umrechnungen beim späteren Sampling mehr notwendig sind (siehe Abschnitt 3.1). Die Light-Injection erfolgt nun, indem für jede Lichtquelle der folgende Prozess mehrmals abgearbeitet wird:

1. Wähle einen zufälligen Punkt auf der Lichtquelle.

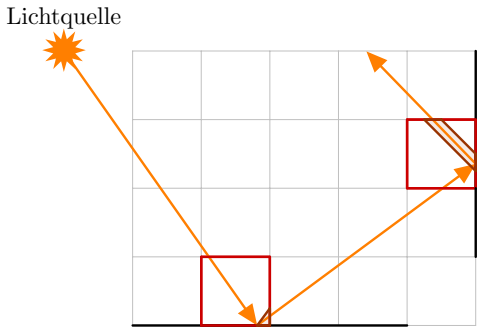


Abbildung 8: Visualisierung des Injection-Passes. Ein Strahl wird von der Lichtquelle aus durch die Szene verfolgt. Pro Voxel (graues Gitter) existiert ein Linespace (rot). An jedem Schnittpunkt wird die Beleuchtungstärke im eingezeichneten Schacht gespeichert.

2. Wähle eine zufällige Lichtrichtung ausgehend von diesem Punkt. Die Richtungen können dabei auf solche beschränkt werden, für die die Lichtstärkeverteilungskurve (LVK) nicht 0 ist.

3. Ermittle die Lichtstärke I für diesen Punkt und diese Richtung mithilfe der LVK.

4. Ermittle den vordersten Schnittpunkt mit der Geometrie für diesen Lichtstrahl.

5. Berechne die Beleuchtungsstärke $E = \frac{I \cdot \cos \theta_e}{d^2}$ resultierend aus diesem Lichtstrahl am Oberflächenpunkt, abhängig vom Winkel θ_e zwischen Lichtstrahl und Normale und der zurückgelegten Distanz d.

   → Falls dies nicht die direkte Beleuchtung ist, ermittle den Schacht sowie die Hauptrichtung und addiere die Beleuchtungsstärke der bestehenden hinzu; erhöhe außerdem den Zähler des Schachts um eins.

6. Wiederhole ab (2), bis die gewünschte Anzahl an Indirektionen erreicht wurde. Dabei wird der aktuelle Oberflächenpunkt erneut als Lichtquelle behandelt; die BRDF tritt anstelle der LVK.

Der Injection-Pass ist unabhängig von der Kameraposition. Die gespeicherte indirekte Beleuchtung muss also nicht neu berechnet werden, wenn sich die Eigenschaften der Kamera ändern. Eine Neuberechnung ist nur dann erforderlich, wenn sich Lichtquellen, Geometrie oder Materialien ändern.

4.2.3 Sampling-Pass

Der Sampling-Pass baut direkt auf Backward Pathtracing auf (siehe Abschnitt 3.3). Der einzige Unterschied ist das Vorgehen bei diffusen Reflexionen.

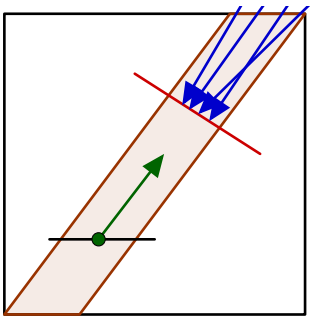


Abbildung 9: Verdeckung des injizierten Lichtes (blau) innerhalb eines Schachts durch ein Objekt (rot). Auslesen der gespeicherten Beleuchtungsstärke für den grünen Oberflächenpunkt würde die Verdeckung nicht berücksichtigen und deshalb zu einem Fehler führen.

Anstatt den neuen Strahl weiterzuverfolgen, erfolgt dabei ein Sampling des Linespaces, d.h. Schacht und Hauptrichtung werden aus Position und Richtung ermittelt und die indirekte Beleuchtungsstärke ausgelesen. Dabei erfolgt eine Normalisierung mithilfe des im Schacht eingetragenen Zählers. Es werden also alle Lichtpfade aus dem vorherigen Pass, die einen Schnittpunkt im ausgewählten Schacht besitzen, gemittelt und mit dem aktuellen Kamerapfad verbunden. Wird die ausgelesene Beleuchtungsstärke als indirekte Beleuchtung verwendet, kann das Pathtracing frühzeitig abgebrochen werden, was die notwendige Anzahl an Indirektionen reduziert.

Alternativ kann dies auch auf Glanzreflexionen erweitert werden. Im Falle einer Phong-BRDF kann man dann zwei Grenzwerte n_1 und n_2 mit $n_1 \leq n_2$ für den Glanzexponenten n definieren. Ein Sampling des Linespaces erfolgt dann, wenn $n < n_1$. Für $n \geq n_2$ wird der Strahl normal weiterfolgt. Für Werte zwischen den Grenzwerten erfolgt es anteilig. Es wird also zwischen beiden Varianten interpoliert.

Ein Problem bei diesem Verfahren stellen die Ungenauigkeiten innerhalb eines Schachts bzw. innerhalb eines Linespaces dar. Da pro Schacht jeweils nur zwei Werte für die beiden Hauptrichtungen gespeichert werden, gehen Detailinformationen des Reflexions- und Verdeckungsverhaltens innerhalb der Schächte verloren. Abbildung 9 veranschaulicht dieses Problem: Das im Schacht gespeicherte Licht ist aus Sicht der zu beleuchtenden Oberfläche vollständig von einem anderen Objekt verdeckt. Damit würde die Oberfläche fehlerhaft beleuchtet werden, wenn man die gespeicherte Beleuchtungsstärke verwendet. Um diese Probleme zu reduzieren, wird zusätzlich der vorderste Schnittpunkt innerhalb des Schachts ermittelt. Ist pro Schacht eine Kandidatenliste vorhanden, lässt sich dies auf Schnittpunkttests mit den Schacht-Kandidaten reduzieren, sodass keine Traversierung einer Datenstruktur notwendig ist. Ohne Kandidatenliste muss der Schnittpunkttest auf die

Grenzen des Linespaces eingeschränkt werden, was je nach verwendeter Datenstruktur andere Optimierungen ermöglichen kann. Falls ein Schnittpunkt gefunden wurde, wird die Beleuchtungsstärke des Schachts nicht verwendet. Stattdessen wird der Schnittpunkt für die nächste Indirektion im Pathtracing verwendet.

Beim Auslesen der Beleuchtungsstärke aus einem Schacht kann es vorkommen, dass der Zähler 0 ist, also kein Licht in diesem Schacht gespeichert wurde. Gründe dafür sind entweder, dass zu wenige Strahlen im Injection-Pass verschossen wurden, oder dass es tatsächlich unmöglich ist, diesen Schacht von einer beliebigen Lichtquelle aus zu erreichen. Zu wenige Indirektionen sind eine weitere Möglichkeit. Für die Behandlung dieser Schächte gibt es zwei unterschiedliche Varianten:

(a) Die Beleuchtungsstärke wird als schwarz interpretiert.

(b) Der Wert wird ignoriert und stattdessen der Strahl weiterverfolgt. Diese Variante wird als *Linespace Injection mit Fallback (auf Pathtracing)* bezeichnet.

Um Artefakte, die durch die Approximation der indirekten Beleuchtung und Diskretisierung in Schächte entstehen, zu reduzieren, erfolgt zusätzlich eine Weichzeichnung. Dies geschieht nicht als Post-Process, sondern bereits vor dem Auslesen der indirekten Beleuchtung aus den Schächten. Hierzu wird die Position des zu beleuchtenden Oberflächenpunktes zufällig auf einer Kreisscheibe mit Radius r orthogonal zur Normalen verschoben. Die Berechnung des Schachts erfolgt dann mit dieser veränderten Position. Der Radius r kann dabei abhängig von Szene und Auflösung des Uniform Grids sowie Linespaces passend gewählt werden.

Eine Konsequenz dieser Weichzeichnung ist die Möglichkeit, dass auf einen Schacht zugegriffen wird, der keine Geometrie enthält. Würde man diese als schwarz interpretieren, würde dies zu einer Verdunklung an den Rändern von Geometrie führen. Stattdessen muss die indirekte Beleuchtung des Samples in diesem Fall komplett ignoriert werden, sodass sich die indirekte Beleuchtung dann durch insgesamt weniger Samples ergibt, als tatsächlich genommen wurden.

5 Implementation

5.1 RSM mit Schattentest

Das Verfahren wurde in OpenGL 4.5 implementiert und gliedert sich in drei Renderpasses:

```
aperture = light->getSpotCutoff();
projection = glm::perspective(2 * aperture, 1.f, near, far);
up = glm::vec3(0,1,0);
if(glm::abs(light->getSpotDirection()) == glm::vec3(0,1,0))
   up = glm::vec3(0,0,1);
viewmatrix = glm::lookAt(light->getPosition(),
      light->getPosition() + light->getSpotDirection(),
      up);
```

Listing 1: Auszug aus der Klasse `LightCamera`, die eine Kamera mit einem Spotlight synchronisiert.

5.1.1 1. Pass

Im ersten Renderpass wird die RSM erstellt. Hierzu wurde eine Klasse `LightCamera` geschrieben, die sich mit der Lichtquelle synchronisiert. Implementiert wurden dabei nur Spotlights. Der Öffnungswinkel für die perspektivische Transformation ergibt sich aus dem Cutoff-Winkel des Spotlights, wobei das Seitenverhältnis immer quadratisch ist. Für die View-Matrix ergibt sich die Kameraposition aus der Position des Lichtes und die Blickrichtung aus dem Richtungsvektor des Spotlights. Als Up-Vektor für `glm::lookAt` wird $(0, 1, 0)$ verwendet. Für den Fall, dass die Richtung des Spotlights selbst die (negative) y-Achse ist, wird stattdessen $(0, 0, 1)$ verwendet. Listing 1 zeigt die Berechnung der perspektivischen Transformation und View-Matrix in C++ anhand der Bibliothek `glm`.

Die Szene wird anschließend aus Sicht dieser Kamera gerendert. Dabei werden Multiple Render-Targets genutzt, um ein Framebuffer-Object (FBO) im Fragment-Shader mit Position sowie Normale in Weltkoordinaten und dem Strahlungsfluss Φ_p zu füllen. Gleichzeitig wird auch automatisch ein Depthbuffer angelegt, sodass Shadow-Mapping ermöglicht wird. Der Vertex-Shader reicht hierzu die Position und Normale in Weltkoordinaten, die View-Matrix und die Position in Kamerakoordinaten an den Fragment-Shader weiter. Der Strahlungsfluss ergibt sich dann aus dem Cosinus des Winkels zwischen normalisiertem Positionsvektor und der Richtung des Spotlights, potenziert mit dem Exponenten des Spotlights. Winkel, die größer als der Cutoff-Winkel des Lichtes sind, ergeben automatisch einen Strahlungsfluss von 0. Listing 2 zeigt den GLSL-Code zur Berechnung des Strahlungsflusses.

5.1.2 2. Pass

Im zweiten Renderpass wird die Szene aus Sicht der Kamera rasterisiert. Dabei wird die direkte Beleuchtung ausgewertet, wobei der Depthbuffer des FBOs aus dem ersten Pass für Shadow-Mapping benutzt wird. Dazu wird der Depthbuffer im Fragment-Shader als `sampler2DShadow` gebunden. Texturzugriffe

```
1  vec3 lightVector = normalize(passPosition);
2  // Kamerakoordinaten
3
4  // Spotlight
5  float spot = 1.0f;
6  float cos_phi_spot = max(dot(lightVector, mat3(viewMatrix) * ↩
       light[i].spot_direction), 0.0f);
7  if(cos_phi_spot >= cos(light[i].spot_cutoff))
8     spot = pow(cos_phi_spot, light[i].spot_exponent);
9  else
10    spot = 0.0f;
11
12 // Farbe
13 fluxOutput = vec4(mat.kd * spot * diffuse_color * light[i].↩
       col, 1.f);
```

Listing 2: Berechnung des Strahlungsflusses eines Spotlights im Fragment-Shader.

auf einen solchen Sampler führen den Schattentest implizit mit mehreren Texturesamples durch und liefern als Ergebnis einen Wert im Intervall $[0, 1]$ [Khr17, S. 274]. Für den Texturzugriff muss der zu beleuchtende Punkt in die Shadow-Map projiziert werden. Dafür wird der Punkt zunächst mit der Projektions- und View-Matrix des Lichtes transformiert, die im ersten Pass ermittelt wurden. Anschließend muss eine manuelle homogene Division durchgeführt werden, da diese Transformation im Fragment-Shader stattfindet und dies deshalb nicht automatisch geschieht. Daraufhin findet eine Transformation vom Intervall $[-1, 1]$ nach $[0, 1]$ statt. Um Schattenartefakte zu vermeiden, wird das *Slope Scale Depth Bias* benutzt, d.h. der Tiefenwert wird abhängig vom Tangens des eingeschlossenen Winkels zwischen Oberflächennormale und Vektor zum Licht gewichtet. Listing 3 zeigt dieses Vorgehen in GLSL. Es wird wieder in ein FBO gerendert, wobei Position und Normale in Weltkoordinaten, sowie das beleuchtete Bild herausgeschrieben werden.

5.1.3 3. Pass

Im letzten Renderpass wird die indirekte Beleuchtung ermittelt. Dafür wird ein Screen-Filling Quad gerendert. Der Fragment-Shader erhält als Eingabe alle sechs Texturen aus den beiden FBOs der ersten beiden Renderpasses. Damit stehen Positionen und Normalen aus Sicht der Lichtquelle und Kamera, sowie der Strahlungsfluss und die beleuchtete Szene zur Verfügung. Zusätzlich werden auf der CPU Zufallszahlen vorberechnet und dem Shader als *Shader Storage Buffer-Object* (SSBO) zur Verfügung gestellt. Die Zufallszahlen dienen dabei als Polarkoordinaten ξ_1 und ξ_2 für die Position der Samples in der RSM. Es gibt also je ein Paar dieser Zufallszahlen pro Sample i. Die Weltposition eines Punktes x aus Sicht der Kamera wird je Sample in die RSM projiziert,

```
1  // Projektion in die Shadow-Map
2  vec4 shadowTest = lightProjectionMatrix * lightViewMatrix * ↩
       passWorldPosition;
3  shadowTest /= shadowTest.w;
4  shadowTest = shadowTest * vec4(0.5, 0.5, 0.5, 1) + vec4(0.5, ↩
       0.5, 0.5, 0.0);
5
6  // Slope Scale Depth Bias
7  float cos_phis = max(dot(passNormal,lightVector),0.0f);
8  float bias = -0.0002 * tan(acos(cos_phis));
9
10 // Schattentest mittels sampler2DShadow
11 float shadow = texture(shadowMap, shadowTest.xyz + vec3↩
       (0.0,0.0,bias));
```

Listing 3: Shadow-Mapping in GLSL: Der Punkt wird in die Shadow-Map projiziert, anschließend erfolgt der Texturzugriff mit einem Slope Scale Depth Bias und liefert einen Schattenwert im Intervall $[0, 1]$.

analog zum Shadow-Mapping aus dem vorherigen Renderpass (siehe Listing 3). Damit ergeben sich Koordinaten (s, t), sodass sich die Texturkoordinate für diesen Sample abhängig vom maximalen Radius wie folgt ergibt [DS05]:

$$(u, v)_{i,x} = (s_x + r_{\max}\xi_{1,i} \sin(2\pi\xi_{2,i}), t_x + r_{\max}\xi_{1,i} \cos(2\pi\xi_{2,i})) \tag{12}$$

$r_{\max}\xi_1$ gibt dabei die Entfernung des Samples (u, v) von (s, t) an. Da die Dichte der Samples mit der quadratischen Distanz abnimmt [DS05] – mit anderen Worten die Größe mit dieser zunimmt (Gleichung 13) – lässt sich daraus das MipMap-Level berechnen, auf dem der Texturzugriff stattfinden soll. Dafür wird die maximale Seitenlänge N_{px} der Textur in Pixeln und der Logarithmus zur Basis 2 benötigt:

$$d_{i,x} = N_{\text{px}}(r_{\max}\xi_{1,i})^2, \text{ der Sampledurchmesser in Pixeln} \tag{13}$$

$$\text{level}(d) = \max(\log_2(d), 0), \text{ das MipMap-Level zu } d \tag{14}$$

Die Texturzugriffe auf die RSM für dieses Sample erfolgen dann mit den Koordinaten (u, v) auf MipMap-Level level(d).

Damit kann Gleichung 11 für das Sample ausgewertet werden. Um die unterschiedlichen Samplegrößen zu kompensieren, findet eine zusätzliche Gewichtung mit ξ_1^2 statt. Diese Gewichtungen werden für alle Samples aufaddiert und dienen der Normalisierung des akkumulierten indirekten Lichtes auf eine Texturfläche von 1, unabhängig von der Anzahl an Samples.

In Gleichung 11 findet zusätzlich eine Division durch das Quadrat der Distanz zwischen Oberflächenpunkt und Pixellicht statt (siehe Abschnitt 3.5). Für sehr kleine Distanzen wird dies problematisch, da dann sehr große

```
float Npx = max(textureSize(fluxMap,0).x, // Texturgröße
textureSize(fluxMap,0).y);
for(int u = 0; u < sampleCount; u++) {
   // Polarkoordinaten
   float xi1 = samples[u].x;
   float xi2 = samples[u].y;
   // Sampledurchmesser und MipMap-Level
   float d = rmax * xi1 * rmax * xi1 * Npx;
   float sampleLevel = max(log2(d),0);
   // Akkumulieren der Samplegrößen
   area += xi1 * xi1;
   // Koordinaten des Samples und Zugriffe in der RSM
   vec2 uv = st.xy + vec2(rmax * xi1 * sin(PI2 * xi2), rmax * xi1 * cos(PI2 * xi2));
   if(uv.x > 1 || uv.y > 1 || uv.x < 0 || uv.y < 0){
      // Sample liegt außerhalb der Textur
      continue;
   }
   vec3 Lpos = vec3(textureLod(posMap, uv, sampleLevel));
   vec3 Lnorm = vec3(textureLod(normMap, uv, sampleLevel));
   vec3 flux = vec3(textureLod(fluxMap, uv, sampleLevel));
   // Distanzgewichtung ab Distanz > 1
   float dist2 = max(dot(pos - Lpos, pos - Lpos),1.0f);
   // Schattentest
   Ray ray;
   ray.origin = wposition + Rayeps * wnormal;
   ray.dir = (pos + Rayeps * norm) - ray.origin;
   if(isIntersection(ray, 1.0))
      continue;
   // Auswerten der indirekten Beleuchtung
   indirect +=  xi1 * xi1 * flux *
   max(0,dot(Lnorm, normalize(pos - Lpos)) *
   max(0,dot(norm, normalize(Lpos - pos)) / dist2;
}
fragmentColor.rgb += indirect / area;
```

Listing 4: Akkumulation des indirekten Lichtes mittels RSM im Fragment-Shader.

Werte angenommen werden. Deshalb findet in dieser Implementation eine Distanzgewichtung nur für Distanzen größer 1 statt.

Der Schattentest selbst kann mit einer beliebigen Datenstruktur durchgeführt werden. Hier wird der Linespace [KLM16] mit voller Kandidatenliste pro Schacht und einem Linespace je Voxel in einem Uniform Grid verwendet, wie in Abschnitt 2.2 beschrieben.

Die akkumulierte, normalisierte indirekte Beleuchtung wird abschließend auf die direkte Beleuchtung aus dem zweiten Renderpass aufaddiert. Listing 4 zeigt die Schleife über alle Samples aus dem Fragment-Shader.

5.2 Linespace Injection

Die Implementation erfolgte in OpenGL 4.5. Im Folgenden wird zunächst der Aufbau von Szenen, Materialien und Lichtquellen beschrieben. Anschließend wir die Implementation des Injection- und Sampling-Passes erläutert.

5.2.1 Szenen

Die gesamte Szenenbeschreibung wird den Shadern in Form von mehreren SSBOs zur Verfügung gestellt. Dabei handelt es sich um einen Buffer für die Materialien der Szene, einen weiteren für alle Lichtquellen, sowie einen Buffer

```
struct Intersection {
   Ray ray; // Strahl für den SP-Test
   float lambda; // Entfernung des gefundenen SPs
   uint objIndex; // Index des geschnittenen Objektes
   bool enter; // Dringt der Strahl in das Objekt ein?
   vec3 location; // SP-Koordinaten
   vec2 tcoord; // (interpolierte) Texturkoordinaten des SPs
   vec3 normal; // (interpolierte) Normale am SP
   bool intersection; // Ob ein SP gefunden wurde
} hit;
```

Listing 5: Global definiertes Struct, das als Schnittstelle zwischen Funktionen für die Schnittpunktberechnung dient.

für die Objekte der Szene. Ein Objekt ist dabei entweder ein Dreieck, eine Kugel oder ein Kegel. Für jedes dieser drei Primitive gibt es einen weiteren Buffer, der die Beschreibungen der Primitive enthält, sowie eine Funktion, die Schnittpunkte mit diesem Primitiv-Typ berechnet. Ein Objekt gibt den Typ des Primitivs, seine Bounding-Box, den Index des Materials und den Index des Primitivs im entsprechenden Buffer an. Im Folgenden wird diese Unterscheidung der Einfachheit halber nicht mehr getroffen und stattdessen nur von Dreiecken geredet.

Für die Schnittpunktberechnung kann eine beliebige Datenstruktur verwendet werden. Hier wird eine Implementation von binären BVHs aus der Diplomarbeit von Robin Schrage [Sch16] verwendet, auf die nicht weiter eingegangen wird, da sie nicht im Fokus dieser Arbeit liegt. Eine Datenstruktur muss zwei Funktionen bereitstellen:

`bool isIntersection(Ray ray, float tMax)`: Test auf beliebigen Schnittpunkt eines Strahls mit der Geometrie mit maximaler Entfernung `tMax` (Schattentest).

`bool closestIntersection()`: Berechnung des vordersten Schnittpunktes. Ein- und Ausgaben erfolgen über ein pro Shaderinstanz global definiertes Struct `hit` (Listing 5).

5.2.2 Materialien

Ein Material wird durch seine diffuse und spekulare Farbe beschrieben. Anstelle einer diffusen Farbe kann auch eine Textur verwendet werden, hierfür werden die Extensions `GL_ARB_bindless_texture` sowie `GL_NV_gpu_shader5` verwendet, die es erlauben die Texturen mithilfe eines 64-Bit unsigned Integer-Handles an den Shader in einem SSBO weiterzureichen und zu nutzen, ohne sie an den OpenGL-Kontext binden zu müssen. Mit einem Wert von 0 wird dabei ein ungültiges Handle identifiziert, was bedeutet, dass die diffuse Farbe anstelle einer Textur verwendet wird. Ein Material enthält weiterhin den

Glanzexponenten für spekulare Reflexionen, einen Brechungsindex sowie die Parameter `kd, ks, kt` und `kr`, die angeben, zu welchem Anteil das Material diffus, spekular, transparent und reflektierend ist. Die BRDF für diffuse und spekulare Materialien ist dabei die Phong-BRDF.

Der Wert für `kt` muss nicht manuell eingegeben werden. Sobald ein Brechungsindex ungleich 0 vorliegt, gilt das Material als refraktierend. Sowohl `kt` als auch `kr` müssen dann abhängig vom Brechungsindex und Winkel zwischen einfallendem Licht und Normale anhand der fresnelschen Formeln neu berechnet werden. In der Implementation wird hierzu Schlicks Approximation verwendet, womit sich `kr`, also der Reflexionskoeffizient, wie folgt berechnet [Sch94]:

$$\mathrm{kr}_0 = \left(\frac{n_1 - n_2}{n_1 + n_2}\right)^2 \tag{15}$$

$$\mathrm{kr}(\theta) = \mathrm{kr}_0 + (1 - \mathrm{kr}_0)(1 - \cos\theta)^5 \tag{16}$$

Der Refraktionskoeffizient `kt` ist entsprechend $1 - \mathrm{kr}$. n_1 und n_2 sind dabei die Brechungsindizes der beteiligten Medien. In dieser Implementation wird dabei immer angenommen, dass Luft eine der beiden Medien ist, d.h. n_1 ist immer 1. θ ist der Winkel zwischen einfallendem Licht und Normale. Für normalisierte Vektoren entspricht $\cos\theta$ also dem Skalarprodukt zwischen diesen.

5.2.3 Lichtquellen

Das Verfahren wurde für Punktlichtquellen und Spotlights implementiert. Das SSBO enthält pro Lichtquelle eine Farbe, Position, Richtung, CutOff-Winkel $\theta_{\max}$, Exponent n und die Grundlichtstärke I_0. Ein CutOff-Winkel größer 0 gibt dabei an, dass es sich um ein Spotlight handelt, ansonsten liegt eine Punktlichtquelle vor. Für letztere ist die LVK für alle Richtungen konstant gleich I_0. Spotlights verwenden die folgende LVK:

$$I(\theta, \phi) = \begin{cases} I_0 \cdot (\cos\theta)^n & \text{für } \theta \leq \theta_{\max} \\ 0 & \text{für } \theta > \theta_{\max} \end{cases} \tag{17}$$

5.2.4 Datenstruktur

Listing 6 zeigt die an Erstellung und Nutzung der Datenstruktur beteiligten Buffer mit Kurzbeschreibung in GLSL. Dabei fällt auf, dass einige Buffer mit dem Schlüsselwort `coherent` versehen sind. Bei diesen handelt es sich um Buffer, die im Zusammenhang mit `atomic`-Operationen verwendet werden. Solche Buffer müssen laut OpenGL-Spezifikation immer `coherent` deklariert werden.

Die Grundlage für die Datenstruktur ist ein Uniform Grid. Hierzu wird die Szene zunächst voxelisiert, wobei für jedes Voxel eine Kandidatenliste

```
struct VoxelToLSIndexMapEntry {
   uint offset_LS; // Start-Index des zugehörigen Linespaces
   uint offset_LSCandList; // Start-Index der Linespace-Kandidatenliste
};

struct InjectionEntry {
   vec3 color; // Akkumulierte Beleuchtungsstärke
   uint num_samples; // Anzahl Injektionen
};

struct ShaftGeo {
   vec3 a, n; // Punkt und Normale einer Ebene
   bool valid; // Ob die Ebene Teil der konvexen Hülle des Schachtes ist
};

// Je Voxel der Start-Index in der Voxelkandidatenliste
layout( binding = 0, r32ui) uniform readonly uimage3D grid;

layout(std430, binding = 7) buffer VoxelToLSInjIndexMap_SSBO {
   uint VoxelToLSInjIndexMap[]; // Start-Index der Linespace-Injectionliste
};

layout(std430, binding = 8) buffer inj_lineSpaces_ssbo {
   // Je Schacht und Linespace: Offset in Injectionliste relativ zum Start-Index
   uint inj_lineSpaces[];
};

layout(std430, binding = 9) coherent buffer LSInjList_ssbo {
   InjectionEntry LSInjList[]; // Injectionliste
};

layout(std430, binding = 10) readonly buffer voxelCandList_ssbo {
   uint voxelCandList[]; // Voxelkandidatenliste
};

layout(std430, binding = 12) coherent buffer atomic_global_SSBO {
   uint atomicGobalCount; // Gesamtzahl an Kandidaten
   uint atomicFilledShaftCount; // Gesamtanzahl an gefüllten Schächten
};

layout(std430, binding = 13) readonly buffer shaft_geo_ssbo {
   ShaftGeo shaft_geo[]; // Schachtgeometrie, 10 Einträge pro möglichem Schacht
};

layout(std430, binding = 14) coherent buffer atomicEmptyVoxelCounterSSBO {
   uint EmptyVoxelCounter; // Anzahl leerer Voxel
};

layout(std430, binding = 15) readonly buffer VoxelToLSIndexMap_SSBO {
   // siehe Struct. Zuordnung von Voxeln in die Linespace-Buffer
   VoxelToLSIndexMapEntry VoxelToLSIndexMap[];
};

layout(std430, binding = 16) buffer lineSpaces_ssbo {
   // Je Schacht und Linespace: Offset in Kandidatenliste relativ zum Start-Index
   uint lineSpaces[];
};

layout(std430, binding = 17) buffer LSCandList_ssbo {
   uint LSCandList[]; // Kandidatenliste
};
```

Listing 6: Buffer- und Bilderdeklarationen in GLSL, die für die Generierung und Nutzung der Datenstruktur notwendig sind.

gespeichert wird (`voxelCandList`). Mit anderen Worten wird für jedes Voxel gespeichert, welche Dreiecke der Szene in diesem vollständig oder teilweise enthalten sind. Die Indizes jedes Voxels innerhalb der Kandidatenliste werden in einem 3D 32-Bit unsigned Integer Bild gespeichert (`grid`).

In einem ersten Compute-Shader werden exemplarisch für einen Linespace die Schacht-Geometrien generiert und im Buffer `shaft_geo` gespeichert. Hierzu werden für jeden Schacht alle infrage kommenden Ebenen erzeugt. Dabei wird zusätzlich geprüft, ob die Ebene tatsächlich Teil der konvexen Hülle ist, was im Feld `valid` gespeichert wird. Insgesamt enthält der Buffer zehn solcher Einträge pro möglichem Schacht. Für eine Linespace-Auflösung N sind das $15N^4$ Schächte.

Der nächste Compute-Shader ermittelt die Anzahl an leeren Voxeln und initialisiert gleichzeitig den Eintrag `offset_LS` für jeden Voxel in der `VoxelToLSIndexMap`. Für leere Voxel, erkennbar anhand der Indices in `grid`, muss kein Linespace angelegt werden. Dies wird mit einem Wert von `MAX_UINT` für `offset_LS` angezeigt. Ansonsten muss ein Linespace angelegt werden. `offset_LS` wird dann auf den aktuellen Wert des `EmptyVoxelCounter`, multipliziert mit der Anzahl Schächten ($15N^4$), gesetzt. Dabei wird `EmptyVoxelCounter` mittels `atomicAdd` um 1 erhöht, der Wert entspricht also nach der Ausführung des Shaders der Anzahl an gefüllten Voxeln. Damit lässt sich der Speicherbedarf für die Buffer `lineSpaces` und `inj_lineSpaces` ermitteln, die einen Eintrag pro möglichem Schacht je gefülltem Linespace erhalten, wobei `offset_LS` jeweils den Start-Index der Einträge eines Linespaces angibt.

Der nächste Compute-Shader wird in zwei Passes ausgeführt und dient der Generierung der Kandidatenliste, sowie der Initialisierung für die in `inj_lineSpaces` enthaltenen Indices. Im ersten Pass wird der Speicherbedarf für Kandidatenliste und Injectionliste ermittelt. Dabei behandelt eine Ausführung des Shaders N^2 Schächte. Die Größe der Workgroups ist entsprechend auf $N * N * 15$ festgelegt, sodass eine Workgroup genau einen Linespace abarbeitet. Um den Speicherbedarf zu ermitteln, erfolgt für jeden Schacht ein Clipping aller zugehörigen Voxelkandidaten gegen die Schachtgeometrie, wobei die Anzahl an Dreiecken gezählt wird, die nach Clipping-Test als innerhalb des Schachtes erkannt wurden. Diese Anzahl wird pro Schacht im Buffer `lineSpaces` gespeichert. Zusätzlich wird in `inj_lineSpaces` eine 0 oder 1 gespeichert, was jeweils angibt, ob der Schacht Geometrie enthält.

Nach diesem ersten Pass ist der Speicherbedarf für jeden Schacht ermittelt, allerdings noch nicht global. Vor dem zweiten Pass werden die Buffer `lineSpaces` und `inj_lineSpaces` deshalb in einem weiteren Compute-Shader pro Linespace in ihre Präfixsumme transformiert und dabei gleichzeitig mittels `atomicAdd` die jeweilige Gesamtsumme über alle Linespaces ermittelt. Durch diese Transformation enthalten die beiden Buffer nun die Offsets in die Kandidaten- bzw. Injectionliste. Da diese Präfixsummen nur lokal pro Linespace berechnet sind, muss zusätzlich für jeden Voxel ein weiterer Offset (`offset_LSCandList`) gespeichert werden, der den Index des ersten

```
uint faceToBlockMap[6][6] =
{
   { 0, 0, 1, 2,  3,  4},
   { 0, 0, 5, 6,  7,  8},
   { 0, 0, 0, 9, 10, 11},
   { 0, 0, 0, 0, 12, 13},
   { 0, 0, 0, 0,  0, 14},
   { 0, 0, 0, 0,  0,  0}
};

uint computeShaftID(vec2 sDim, vec2 eDim, uint startFace, uint endFace) {
   uint block, shaftID;

   int N = int( Nls); // Linespace Auflösung
   int u_index = max(0, min( int( sDim.x * N), N-1));
   int v_index = max(0, min( int( sDim.y * N), N-1));
   uint index_s = N * v_index + u_index;

   u_index = max(0, min( int( eDim.x * N), N-1));
   v_index = max(0, min( int( eDim.y * N), N-1));
   uint index_e = N * v_index + u_index;

   if(startFace > endFace) {
      block = faceToBlockMap[endFace][startFace];
      shaftID = N*N*N*N * block + N*N * index_e + index_s;
   }
   else {
      block = faceToBlockMap[startFace][endFace];
      shaftID = N*N*N*N * block + N*N * index_s + index_e;
   }
   return shaftID;
}
```

Listing 7: Berechnung der Schacht-ID relativ zum Start-Index eines Linespaces in GLSL.

Schachtkandidates des zugehörigen Linespaces innerhalb der Kandidatenliste angibt. Dieser Wert ist immer der bisher akkumulierte Wert an Kandidaten. `atomicGobalCount` enthält dann die Gesamtanzahl an Kandidaten, `atomicFilledShaftCount` die Anzahl an gefüllten Schächten. Diese Werte lassen sich auslesen, um dann den Speicher für Kandidatenliste und Injectionliste zu allozieren. Bei letzterem sind doppelt so viele Einträge nötig, wie gefüllte Schächte existieren, da jeweils zwei Werte pro Schacht gespeichert werden.

Anschließend wird im zweiten Pass wieder das gleiche Clipping durchgeführt, diesmal werden aber die Dreiecks-IDs in der durch `lineSpaces` und `offset_LSCandList` angegebenen Position in der Kandidatenliste gespeichert.

Weiterhin wird eine Funktion benötigt, die die ID, also den relativen Index, eines Schachtes innerhalb eines Linespaces berechnet. Hierfür werden die sechs Seitenflächen eines Linespaces von $0 - 5$ nummeriert. Die Organisation im Speicher erfolgt in Blöcken, wobei ein Block jeweils alle Schächte zwischen zwei Seitenflächen speichert. Da jede Seitenfläche in N^2 Patches unterteilt ist, ergeben sich damit N^4 Schächte pro Block. Zwischen einer Seitenfläche und sich selbst existieren keine Schächte. Damit gibt es $5 * 6 = 30$ Möglichkeiten für die Blöcke. Dies halbiert sich auf 15 Blöcke, da die Schächte zwischen zwei Seitenflächen unabhängig von der Reihenfolge immer die gleichen sind. Damit lässt sich die Funktion `computeShaftID` wie in Listing 7 definieren. `sDim` und

`eDim` sind dabei die Schnittpunktkoordinaten auf den Seitenflächen (im Intervall $[0, 1]$) eines gegebenen Strahls in negativer und positiver Strahlrichtung. `startFace` und `endFace` sind die Nummern der Seitenflächen.

Die Injectionliste speichert pro Schacht zwei Einträge. Ein Eintrag besteht dabei jeweils aus einer Farbe (`vec3`) und dem Zähler (`uint`). Damit ergibt sich ein Speicherbedarf von 32 Byte pro gefülltem Schacht für die Injectionliste. Dazu kommt noch der Speicher für die restlichen Buffer-Strukturen, die zur Indizierung der Listen dienen. Im Speicher ist es so angelegt, dass zunächst der erste Eintrag jedes Schachts gespeichert wird und anschließend die zweiten Einträge. Diese sind also jeweils um `atomicFilledShaftCount` verschoben. Das Ordnungskriterium für die Entscheidung, ob es sich um den ersten oder zweiten Eintrag handelt, ist `startFace` $<$ `endFace`. Diese Bedingung wird für den Sampling-Pass umgedreht, da hier ausgehende Strahlen statt einfallenden Strahlen vorliegen. Listing 8 zeigt die Berechnung des Indices in der Kandidaten- und Injectionliste für den Sampling-Pass.

5.2.5 Zufallszahlen

Sowohl Injection-Pass als auch Sampling-Pass basieren auf Pathtracing und sind deshalb auf Zufallszahlen angewiesen. Da diese deterministisch berechnet werden und sich somit voraussagen lassen, bezeichnet man sie als Pseudozufallszahlen. Eine Zahl auf der GPU oder CPU kann nur eine begrenzte Anzahl an Zuständen annehmen, weshalb bei der Generierung von Zufallszahlen irgendwann ein Zyklus auftritt. Für gute Zufallszahlen muss dieser Zyklus also möglichst lang sein, weiterhin muss die Verteilung der Zufallszahlen möglichst gleichmäßig sein.

In dieser Arbeit erfolgt die Generierung der Zufallszahlen auf der GPU mithilfe eines Generators nach [Ngu07, Kap. 37]. Dieser Generator kombiniert den Linear Congruential Generator (LCG) [Knu69] mit dem Combined Tausworthe Generator (CTG) [L'e96]. Der Zustand dieser Generatoren wird in einem Bild mit vier Kanälen gespeichert, das initial von der CPU mit Zufallswerten gefüllt wird. Um die nächste Zufallszahl zu generieren, werden die ersten drei Komponenten jeweils mit CTG transformiert und die vierte Komponente mit LCG. Die transformierten Werte werden dann wieder im Bild gespeichert und anschließend mit XOR-Operationen in eine Gleitkommazahl transformiert. Listing 9 zeigt den GLSL-Code für die Generierung der Zufallszahlen.

5.2.6 Injection-Pass

Der Injection-Pass wurde mit einem Vertex- und Fragment-Shader implementiert. Die Ausgabe des Fragment-Shaders ist dabei irrelevant, es ließe sich also genauso gut ein Compute-Shader verwenden. Gerendert wird ein Screen-Filling Quad, womit die Anzahl an Pixeln – abhängig von der gewählten

```
uint getInjID() {
   uint startFace, endFace;
   Ray ray = hit.ray;
   vec2 sDim, eDim;

   /* Berechne startFace und endFace (SP des Strahls mit dem Voxel in + und - Richtung)
      Berechne sDim, eDim (UV-Koordinaten auf start- und endFace) */
   [...]

   // Ermittle Start-Index des zugehörigen Linespaces
   uint offset_LS = VoxelToLSIndexMap[ voxelID].offset_LS;

   if(offset_LS != MAX_UINT) { // MAX_UINT zeigt leere Voxel an, ignorieren
      uint offset_Shaft = 0;
      uint offset_InjShaft = 0;
      uint ShaftID = computeShaftID (sDim, eDim, startFace, endFace);
      if(ShaftID != 0) {
         /* Offsets in die Kandidaten- und Injectionliste
            relativ zu Start-Index in diesen Listen für diesen Linespace */
         offset_Shaft = lineSpaces[offset_LS + ShaftID - 1];
            offset_InjShaft = inj_lineSpaces[offset_LS + ShaftID - 1];
      }
      /* Anzahl an Kandidaten ergibt sich aus Differenz mit dem
         Offset des nächsten Schachts */
      count_LSCandidates = lineSpaces[offset_LS + ShaftID] - offset_Shaft;
      // Addiere relativen Offset zum Start-Index
      offset_LSCandList = VoxelToLSIndexMap[voxelID].offset_LSCandList + offset_Shaft;
      offset_LSInjList = VoxelToLSInjIndexMap[voxelID] + offset_InjShaft;
   }

   /* Bis hier ist das Vorgehen gleich zu getInjID2() (Injection-Pass)
      Da die Richtungen beim Sampling entgegengesetzt zu denen bei Injection sind,
      wird die Bedingung für die korrekte Hauptrichtung umgedreht */
   bool shaftDir = startFace > endFace ? true : false;

   if(count_LSCandidates != 0) {
      // Bestimme vordersten Schnittpunkt mit Schachtkandidaten
       hit.ray.origin += Rayeps * hit.ray.dir; // Selbstverschattung vermeiden
       hit.intersection = lineSpace_candidates_closestIntersection(offset_LSCandList, ↩
            count_LSCandidates, tmin, tmax);
       if(shaftDir) {
         // Verschieben des Offsets abhängig von Hauptrichtung
         offset_LSInjList += atomicFilledShaftCount;
       }
       return offset_LSInjList;
   }
   return MAX_UINT; // Schacht ohne Kandidaten, ID ungültig
}
```

Listing 8: Ermitteln des Index in der Injectionliste mitsamt Schnittpunktberechnung mit den Kandidaten des Schachtes in GLSL.

```glsl
// Reference: GPU Gems 3 Chapter 37
layout( binding = 1, rgba32ui) uniform restrict uimage2D rndImage;

uint TauswortheStep(uint z, int S1, int S2, int S3, uint M) {
   uint b = (((z << S1) ^ z) >> S2);
   return (((z & M) << S3) ^ b);
}

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

float Random() {
   uvec4 seed = uvec4(imageLoad(rndImage, pixelPos));
   seed.x = TauswortheStep(seed.x, 13, 19, 12, 4294967294);
   seed.y = TauswortheStep(seed.y, 2, 25, 4, 4294967288);
   seed.z = TauswortheStep(seed.z, 3, 11, 17, 4294967280);
   seed.w = LCGStep(seed.w, 1664525, 1013904223);

   imageStore(rndImage, pixelPos, seed);

   float rndNumber = 2.3283064365387e-10 * float(seed.x ^ seed.y ^ seed.z ^ seed.w);

   return rndNumber;
}
```

Listing 9: Generierung von Zufallszahlen in GLSL.

Auflösung – angibt, wie viele Fragment-Shader Instanzen ausgeführt werden.

Im Fragment-Shader wird in einer Schleife über alle Lichtquellen der Szene iteriert. Für jede Lichtquelle wird ein Strahl mit Ursprung auf der Lichtquelle und einer zufälligen Richtung erzeugt. Bei einer Punktlichtquelle ist dies eine beliebige Richtung, bei einem Spotlight beschränken sich die Richtungen auf jene, die maximal mit dem CutOff-Winkel von der Lichtrichtung abweichen. Die Farbe der Lichtquelle wird mit der Lichtstärke I, die sich wie beschrieben aus der LVK ergibt, multipliziert und zusammen mit dem Strahl durch die Szene verfolgt.

An jedem Schnittpunkt wird die bisher akkumulierte Farbe mit dem Cosinus des Winkels zwischen negativer Lichteinfallsrichtung und Normale am Schnittpunkt multipliziert und durch das Quadrat der zurückgelegten Distanz geteilt, um die Beleuchtungsstärke zu berechnen. Die Distanz wird nach unten auf 1 beschränkt, um zu vermeiden, dass unendlich große Werte entstehen.

Anschließend wird abhängig von den Materialeigenschaften und der BRDF ein neuer Strahl berechnet. Dabei kommt das Prinzip des russischen Roulettes zum Einsatz, das heißt es wird eine Zufallszahl im Intervall $[0, 1]$ erzeugt und dann abhängig von den Koeffizienten `kd`, `ks`, `kt` und `kr` ausgewählt, ob eine diffuse, spekulare oder perfekte Reflexion stattfindet, oder das Licht gebrochen wird. Bei diffuser Reflexion wird die akkumulierte Farbe mit der diffusen Farbe multipliziert, bei spekularer Reflexion entsprechend mit der spekularen Farbe und zusätzlich mit der PDF:

$$\mathrm{PDF}(n, \theta) = \frac{n+2}{n+1} \cos\theta \tag{18}$$

Wobei n der Glanzexponent und θ der Winkel zwischen Normale und der spekular reflektierten Richtung ist. Sowohl bei diffusen als auch bei spekularen Materialien wird dabei die reflektierte Richtung zufällig ausgewählt. Im diffusen Fall ist keine Gewichtung mit der PDF notwendig, da die Richtungen bereits im Sinne des Importance Samplings mit einer Cosinus-Verteilung generiert werden.

Das Verfahren bricht ab, wenn kein Schnittpunkt gefunden wurde oder eine vorgegebene Anzahl an Indirektionen erreicht wurde. Am ersten Schnittpunkt, also dem direkten Licht, findet keine Injection statt. Bei allen weiteren Schnittpunkten wird die Schacht-ID in der Injectionliste ermittelt und anschließend mit vier `atomicAdd`-Operationen die akkumulierte Farbe auf die bestehende Farbe im Schacht addiert und der Zähler um 1 erhöht. Atomic-Operationen sind hier nötig, da es möglich ist, dass mehrere Shader-Instanzen gleichzeitig auf den selben Schacht zugreifen wollen. Aus diesem Grund ist die Injectionliste im Shader auch als `coherent` deklariert.

In GLSL gibt es allerdings standardmäßig kein `atomicAdd` für Gleitkommazahlen. Stattdessen müsste man also in der Injectionliste nur unsigned Integer speichern. Diese können dann mit dem Befehl `uintBitsToFloat` in eine Gleitkommazahl verwandelt werden, woraufhin die Farbe hinzuaddiert werden kann und dann mit `floatBitsToUint` wieder in einen unsigned Integer konvertiert werden kann. Bei diesem Vorgehen kann allerdings kein `atomicAdd` verwendet werden. Stattdessen müsste je Kanal ein `atomicCompSwap` innerhalb einer while-Schleife ausgeführt werden. Das beschriebene Vorgehen ist umständlich. Hier schafft die Extension `GL_NV_shader_atomic_float` Abhilfe, die GLSL um ein `atomicAdd` für Gleitkommazahlen erweitert und damit die Probleme beseitigt.

Listing 10 zeigt die main-Methode des Fragment-Shaders für den Injection-Pass, wobei der Code auf das wesentliche reduziert wurde.

5.2.7 Sampling-Pass

Der Sampling-Pass wird ebenfalls durch Rendern eines Screen-Filling Quads gestartet. Im Fragment-Shader wird dann zunächst für jeden Pixel ein Kamerastrahl generiert. Hierzu wird mittels `gl_FragCoord` der Mittelpunkt des Pixels abgefragt. Weiterhin erhält der Fragment-Shader die Kamerainformationen in Form eines `uniform struct`. In diesem Struct wird die untere linke Ecke der Bildebene, die Kameraposition, sowie zwei Vektoren, die die Bildebene aufspannen, übergeben. Dabei liegen diese Daten in Weltkoordinaten vor, die Länge der Vektoren entspricht jeweils der Breite bzw. Höhe eines Pixels. Mithilfe dieser Angaben lässt sich nun ein Strahl durch einen zufälligen Punkt auf dem Pixel wie in Listing 11 gezeigt berechnen. Damit wird erreicht, dass über die gesamte Fläche des Pixels integriert wird, was Aliasing verhindert.

An den Fragment-Shader werden zusätzlich zwei vier-kanalige Bilder mit 32-Bit pro Kanal gebunden. Das eine Bild speichert dabei die akkumulierte

```
void main() {
pixelPos = ivec2(gl_FragCoord); // Für Random()
for(int i = 0; i < nLights; ++i) {
   vec3 colAccum;
   // Generierung des Lichtstrahls und Lichtstärke * Farbe
   hit.ray = GenerateLightRay(i, colAccum);
   bool intersection = closestIntersection();
   if(intersection) {
      // Direktes Licht - nicht speichern
      // Ermittle diffuse Materialfarbe (evtl. Texturzugriff)
      [...]
      // Berechnung der Beleuchtungsstärke
      float cosThetaReceiver = max(dot(normalize(hit.normal), -hit.ray.dir), 0.f);
      float distanceToLightSquare_inv = min(1.f / (hit.lambda * hit.lambda), 1.f);
      colAccum *= cosThetaReceiver * distanceToLightSquare_inv;

      // Indirektionen, mindestens eine, da sonst nichts injiziert wird
      int depth = 0;
      Ray ray; // Neuer Strahl
      do {
         // Fresnel für Neuberechnung von kt, kr
         if(currentMat.ior > 0.001f) {[...]}

         float randRR = Random();
         /* Russisches Roulette */
         // perfekte Reflexion
         if(randRR < currentMat.kr) {[...]}
         // Refraktion
         else if(randRR < currentMat.kr + currentMat.kt) {[...]}
         else if(randRR < currentMat.kr + currentMat.kt + currentMat.kd) {
            colAccum.rgb *= diffuse_color;
            [...] // diffuse Reflexion
         }
         else if(randRR < currentMat.kr + currentMat.kt + currentMat.kd + currentMat.ks) {
            // spekulare Reflexion, ausführlich
            vec3 rand_dir = SamplePhongDirection(hit.normal, hit.ray.dir, currentMat.↩
                 shininess);
            float cos_theta = dot(rand_dir, hit.normal);
            if(cos_theta < Eps)
               break; // Strahl im hinteren Halbraum, verwerfen
            else {
               ray.origin = hit.location + Rayeps * rand_dir;
               ray.dir = rand_dir;
               colAccum.rgb *= (currentMat.shininess + 2) / (currentMat.shininess + 1) * ↩
                    cos_theta * currentMat.specColor;
            }
         }
         hit.ray = ray;
         intersection = closestIntersection();
         uint index = getInjID2(); // Index in Injektionliste
         if(intersection) {
            // Beleuchtungsstärke
            distanceToLightSquare_inv = min(1.f / (hit.lambda * hit.lambda), 1.f);
            cosThetaReceiver = max(dot(normalize(hit.normal), -hit.ray.dir), 0.f);
            if(index != MAX_UINT) { // Index valide, addiere Beleuchtungsstärke zum Schacht
               atomicAdd(LSInjList[index].color.r, colAccum.r);
               atomicAdd(LSInjList[index].color.g, colAccum.g);
               atomicAdd(LSInjList[index].color.b, colAccum.b);
               atomicAdd(LSInjList[index].num_samples, 1);
            }
         // Ermittle diffuse Materialfarbe für neuen SP
         [...]
         }
         else
            break;
         depth++;
      } while(depth < Maxdepth);
   }
}
// fragmentColor interessiert nicht
}
```

Listing 10: Main-Methode für den Injection-Pass in GLSL.

```
1 ray.dir = normalize(frustum.bottomleft + (pixel.x - 0.5f + ↩
      Random()) * frustum.steprightvec + (pixel.y - 0.5f + ↩
      Random()) * frustum.stepupvec - frustum.view);
2 ray.origin = frustum.view;
```

Listing 11: Berechnung eines Strahls durch einen zufälligen Punkt auf einem Pixel der Bildebene in GLSL.

Farbe, die sich durch die Pathtracing-Anteile im Sampling-Pass ergibt, in um die Sampleanzahl normalisierter Form. Das sind also genau die Anteile, die sich durch die direkte Beleuchtung an jedem Schnittpunkt eines Pfades ergeben. Die Sampleanzahl wird dabei mit einem `uniform` in jedem Durchgang neu an den Shader übergeben. Um die Farbe c des Samples i (bei 0 beginnend gezählt) wieder in diesem Bild abzuspeichern, was dann gleichzeitig die Farbe des Fragments ist, muss die Normalisierung rückgängig gemacht werden und nach der Addition neu normalisiert werden:

$$c_{\text{neu}} = \frac{i \cdot c_{\text{alt}} + c_i}{i + 1} \tag{19}$$

Das zweite Bild speichert ausschließlich die Farben, die durch das Sampling des Linespaces zustande kommen. Da es dabei möglich ist, dass ein Schacht, der keine Kandidaten enthält, fälschlicherweise ausgewählt wird (siehe Abschnitt 4.2.3), kann das zuvor beschriebene Normalisierungsschema nicht verwendet werden. Stattdessen werden diese Farben im Bild ohne Normalisierung aufaddiert, wobei für jeden Sample die vierte Komponente 0 ist, wenn dieser Sample ungültig war, und ansonsten 1. Damit enthält die vierte Komponente in diesem Bild je Pixel die Anzahl an gültigen Linespace-Samples. Es muss also nur noch durch diese dividiert werden und das Ergebnis auf c_{neu} addiert werden. Damit erhält man die finale Farbe des Fragments.

Der eigentliche Vorgang der Strahlverfolgung und des Samplings ist in Listing 12 gegeben. Der Einfachheit halber wurde hier auf den Code verzichtet, der für die Erweiterung des Samplings auf spekulare Materialien notwendig ist. Hierzu werden zwei Uniforms n_1 und n_2 für die Schwellwerte des Glanzexponenten an den Shader übergeben. Die Entscheidung, ob der Linespace gesamplet wird, findet dann wie in Abschnitt 4.2.3 beschrieben statt. Für den Fall, dass $n_1 \leq n < n_2$ gilt, erfolgt die anteilige Aufteilung mit russischem Roulette. Dabei wird n relativ zu den Schwellwerten auf $[0, 1]$ normalisiert:

$$n' = \frac{n - n_1}{n_2 - n_1} \tag{20}$$

Ist die generierte Zufallszahl $\xi \in [0, 1]$ kleiner als n', findet kein Sampling des Linespaces statt, ansonsten schon.

```
vec4 Trace(in Ray ray, out vec4 direct) {
   direct_col = vec4(0,0,0,1); // Farbe die durch PT zustande kommt
   vec4 indirect_col = vec4( 0.f,0.f,0.f,0.f); // Farbe durch Sampling des Linespace
   bool intersection;
   bool skip = false;
   int depth = 0;
   vec4 colAccum = vec4(1); // Akkumulation von Materialfarben und Gewichtungen
   while( depth < (Maxdepth + 1)) {
      if(!skip) { // Berechne Schnittpunkt
         hit.ray = ray;
         intersection = closestIntersection();
      } else { // Schnittpunkt bereits bei Sampling des Linespace ermittelt
         intersection = true;
         skip = false;
      }
      if(intersection) {
         // Bestimme diffuse Materialfarbe (evtl. Textur)
         [...]
         direct_col += colAccum * shade(diffuse_color); // Direktes Licht
         // Fresnel und Russisches Roulette wie im Injection-Pass
         [...]
         // Diffus: Sampling des Linespace
         else if( randRR < currentMat.kr + currentMat.kt + currentMat.kd) {
            colAccum.rgb *= diffuse_color;
            vec3 rand_dir = SampleDiffuseDirection( hit.normal);
            // Verschiebe Position zufällig auf Kreisscheibe orthogonal zur Normalen
            vec3 disk_offset = sampleDisk(Random(), Random(), hit.normal, diskRadius);
            hit.ray.origin = hit.location + disk_offset;
            hit.ray.dir = rand_dir;
            // Ermittle Index in Injektionliste, gleichzeitig SP-Test mit Schachtkandidaten
            uint idx = getInjID();
            if(idx < MAX_UINT) {
               if(hit.intersection) {
                  // SP, ignoriere Wert im Schacht und übernehme ihn für nächste ↩
                     Indirektion
                  depth++;
                  skip = true;
                  continue;
               }
               InjectionEntry entry = LSInjList[idx];
               indirect_col.rgb += entry.color;
               indirect_col.w += entry.num_samples;
               if(indirect_col.w == 0) { // Kein Licht im Schacht gespeichert
                  if(use_pt_on_zero){ // Sampling mit Fallback auf PT, langsamer
                     ray.origin = hit.location + Rayeps * rand_dir;
                     ray.dir = rand_dir;
                     depth++;
                     continue;
                  }
                  // Schneller: Nehme ein schwarzes Sample
                  else{
                     indirect_col = vec4(0,0,0,1);
                  }
               }
               indirect_col /= indirect_col.w; // Normalisierung der Beleuchtungsstärke
               indirect_col *= colAccum;
               break;
            } else // Ungültige ID (leerer Schacht!) Ignoriere diesen Sample und zähle ihn ↩
                nicht
               break;
         }
         // Spekular
         [...]
      } else { // Kein Schnittpunkt. Schwarz (Hintergrund) für PT und indirektes Licht.
         direct_col += vec4(0,0,0,1);
         indirect_col += vec4(0,0,0,1);
         break;
      }
      depth++;
   }
   return indirect_col;
}
```

Listing 12: Verfolgung der Strahlen mit Linespace Sampling in GLSL.

Teil IV
Evaluation

6 RSM mit Schattentest

6.1 Testumgebung

Die Implementation der RSM mit Schattentest wurde auf einem Rechner mit Nvidia GeForce GTX Titan X Grafikkarte, Intel i7-6700 CPU sowie 32GB RAM evaluiert.

Getestet wurde die Implementation anhand von zwei Szenen. Die erste Szene ähnelt der, die Dachsbacher et al. in [DS05] verwendet haben und besteht aus der Ecke eines Raumes ohne Decke mit unterschiedlich farbigen Wänden und der Lucy-Statue. Bei der Statue handelt es sich um eine niedrigere Auflösung der Originaldaten des Stanford 3D Scanning Repositories mit insgesamt 22452 Eckpunkten und 44883 Dreiecken. Die Lichtquelle ist ein Spotlight mit Cutoff-Winkel 60° und Exponent 2.

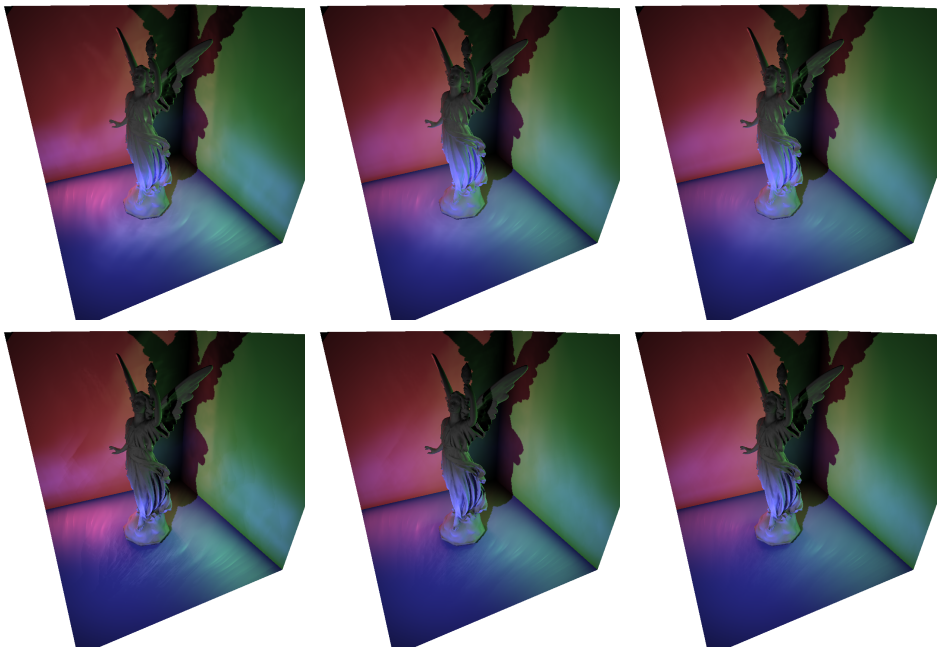
Die zweite Szene ist die Cornell-Box. Anstelle einer flächigen Lichtquelle an der Decke kommt ebenfalls ein Spotlight mit den gleichen Parametern der Lucy-Szene zum Einsatz, da flächige Lichtquellen für dieses Verfahren nicht implementiert sind. Das Spotlight befindet sich dabei in der vorderen linken Ecke am Boden und strahlt in Richtung der hinteren rechten Ecke der Decke.

Tabelle 1 zeigt die Renderzeiten der beiden Szenen mit unterschiedlicher Anzahl an RSM-Samples pro Pixel, je mit aktiviertem und deaktiviertem Schattentest. Dabei wurde auf jeweils zwei relevante Stellen gerundet. Für die Lucy-Szene wurde ein Linespace mit Auflösung 4 und Voxelauflösung 50x50x50 verwendet, während für die Cornell-Box eine Voxelauflösung von 1x1x1 benutzt wurde. Bild und indirekte Beleuchtung wurden jeweils in Full HD berechnet, die RSM in Auflösung 1080x1080. Die zur Tabelle gehörigen Ergebnisbilder zeigt Abbildung 10. Dabei wurde die indirekte Beleuchtung verstärkt, um Artefakte und Effekte sichtbarer zu machen.

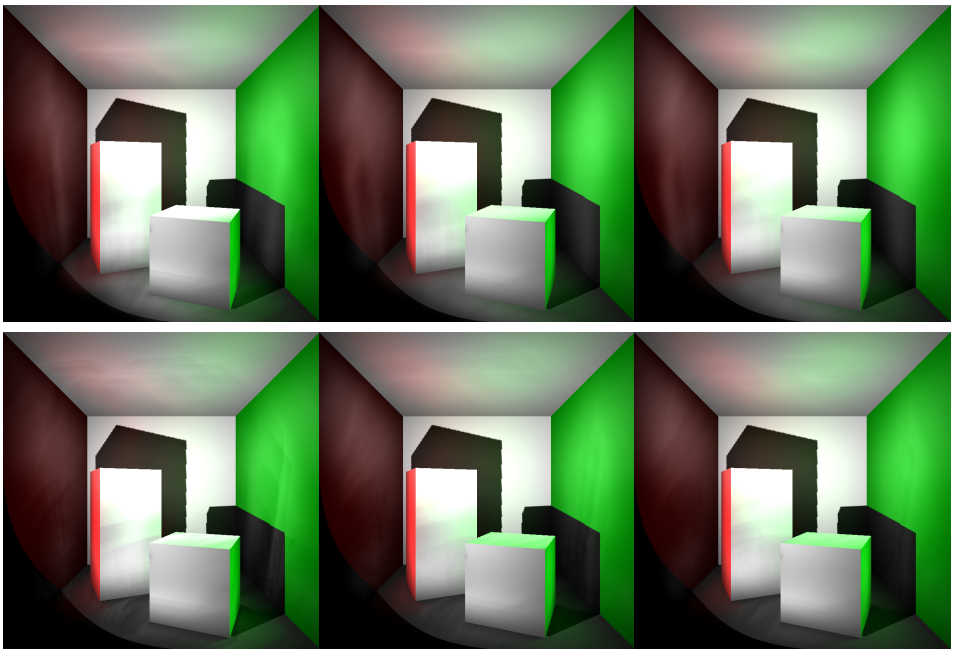
6.2 Renderzeiten

Ohne Schattentest werden in der Cornell-Box ca. 70% der FPS verglichen mit der Lucy-Szene erreicht. Dies lässt sich damit begründen, dass in der Lucy-Szene ein größerer Teil des Bildes aus Hintergrund besteht, für den keine indirekte Beleuchtung ausgewertet werden muss. Da die Berechnung des indirekten Lichtes ohne Schattentest unabhängig von der Szenenkomplexität ist [DS05], führt dies zu einem Geschwindigkeitszuwachs.

Die Renderzeiten sind um einen Faktor von 2-4 kürzer als in den Ergebnissen von Dachsbacher et al., obwohl hier die Auflösung höher ist (1920x1080 statt 512x512 für das Bild bzw. 1080x1080 statt 512x512 für die RSM) und



(a) Obere Reihe: Lucy-Szene mit je 100, 225 und 400 RSM-Samples pro Pixel. Untere Reihe: Das Gleiche mit aktiviertem Schattentest.



(b) Obere Reihe: Cornell-Box mit je 100, 225 und 400 RSM-Samples pro Pixel. Untere Reihe: Das Gleiche mit aktiviertem Schattentest.

Abbildung 10: Zwei Szenen mit indirekter Beleuchtung mittels RSM, gerendert mit verschiedener Anzahl an Samples und Schattentest an- bzw. ausgeschaltet.

FPS	Samples/Pixel	Schattentest
67	100	—
1.8	100	✓
37	225	—
0.81	225	✓
23	400	—
0.45	400	✓

(a) Lucy (Bilder siehe Abbildung 10a)

FPS	Samples/Pixel	Schattentest
49	100	—
7.5	100	✓
26	225	—
3.4	225	✓
16	400	—
1.9	400	✓

(b) Cornell-Box (Bilder siehe Abbildung 10b)

Tabelle 1: Renderzeiten zu Abbildung 10 für RSM mit verschiedener Anzahl an Samples und Schattentest an- bzw. ausgeschaltet. Die Bilder wurden in Full HD gerendert, die RSM in Auflösung 1080x1080.

auf die Interpolation des indirekten Lichtes aus einer wesentlich niedrigeren Auflösung verzichtet wurde. Da sich seit ihrer Veröffentlichung die Performance von Grafikkarten um ein Vielfaches gesteigert hat, sind diese Ergebnisse erwartungskonform.

Mit aktivem Schattentest liegen die FPS in der Lucy-Szene bei ca. 2% der FPS ohne Schattentest. In der Cornell-Box werden hingegen ca. 13% erreicht. Verglichen mit der Lucy-Szene rendert die Cornell-Box mit Schattentest insgesamt ca. 4 mal so schnell, was den Ergebnissen ohne Schattentest widerspricht. Die Ursache für diese Beobachtung ist, dass durch den Schattentest über Schnittpunkte mit der Geometrie die Renderzeit nun abhängig von der Szenenkomplexität und der verwendeten Datenstruktur ist. Die Cornell-Box besteht aus 34 Dreiecken, während Lucy mit 44883 Dreiecken wesentlich komplexer ist. Es sind also mehr Schnittpunkttests nötig. Zusätzlich muss die Linespace-Datenstruktur in der Cornell-Box nicht traversiert werden, da die Szene mit nur einem Voxel gerendert wurde, während die Voxelauflösung von 50x50x50 für die Lucy-Szene auch einen Zeitaufwand für die Traversierung erfordert.

Die Renderzeiten bei aktivem Schattentest zeigen, dass dieses Verfahren für interaktive Anwendungen ungeeignet ist. Mit einem Vorgehen ähnlich des Pathtracings, also indem man pro Frame nur ein RSM-Sample nimmt, ließe sich dieses Problem lösen. Der Nachteil dabei ist, dass sich die indirekte Beleuchtung dann erst mit der Zeit aufbaut, während die Kamera stillsteht. Außerdem müsste die Lichtquelle dann statisch sein. Ob diese Nachteile ein Problem darstellen, hängt von der Anwendung ab. Eine weitere Möglichkeit, um das Verfahren zu beschleunigen, stellt die Verwendung von anderen Datenstrukturen da. Hier sind insbesondere Ansätze interessant, die die Sichtbarkeit vorberechnen und damit Schnittpunkttests obsolet machen. Ein solcher Ansatz wird von Kevin et al. in [KKM17] beschrieben, der ebenfalls den Linespace benutzt, wobei ein Schacht jeweils nur ein Bit speichert, nämlich ob der Schacht gefüllt ist, oder nicht.

6.3 Ergebnisbilder

Die Ergebnisbilder ohne Schattentest in Abbildung 10 zeigen die gleichen Artefakte, die auch Dachsbacher et al. beschreiben. Dies ist zum einen die Streifenbildung, die insbesondere bei niedrigen Sampleanzahlen auftritt und auf die räumliche Kohärenz durch gleichbleibende Zufallszahlen zurückzuführen ist. Zum anderen gibt es Probleme in den Ecken zwischen zwei Wänden aufgrund der Singularität im Beleuchtungsintegral. Letzteres Problem verbesserten die Autoren, indem sie die Pixellichter um einen konstanten Betrag in Richtung der negativen Normalen verschieben. Dies ist allerdings nur möglich, weil sie keine Verdeckung berücksichtigen und kommt deshalb für die hier vorgestellte Variante mit Schattentest nicht in Frage [DS05].

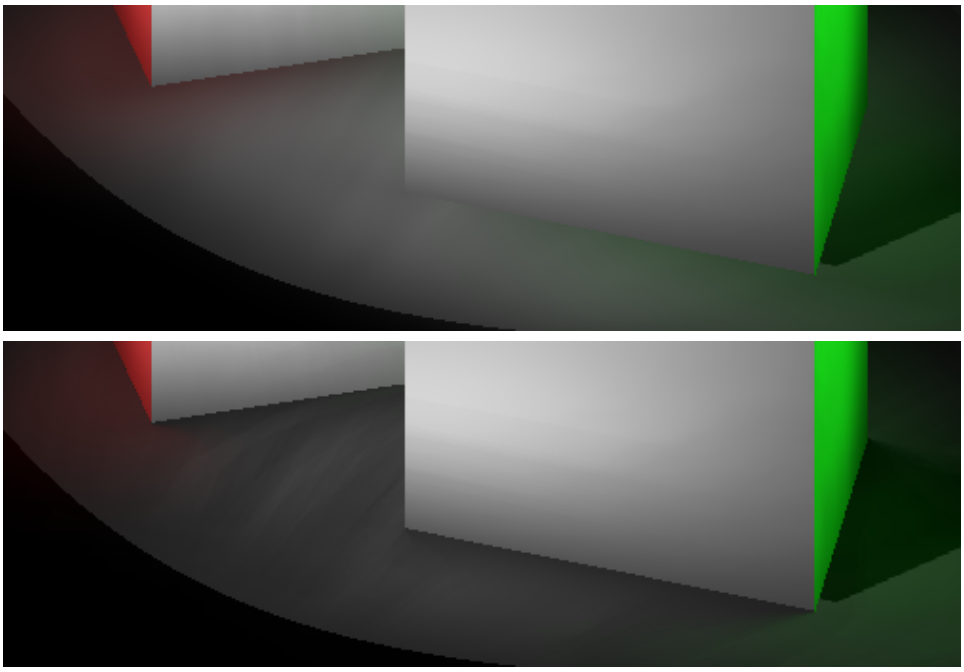


Abbildung 11: Vergrößerter Ausschnitt aus Abbildung 10b mit 400 RSM-Samples. Die indirekte Beleuchtung durch die Wände wird bei aktivem Schattentest von den beiden Quadern verdeckt, führt aber auch zu stärkerer Streifenbildung.

Bei aktivem Schattentest ist die Streifenbildung in Bereichen, die stark verdeckt sind, stärker als bei gleicher Sampleanzahl ohne Schattentest. Abbildung 11 zeigt einen vergrößerten Ausschnitt aus der Cornell-Box mit 400 Samples. Die Verdeckung durch die beiden Quader sorgt hier für starke Streifenbildung zwischen diesen, verglichen mit der Variante ohne Schattentest. Ähnliches lässt sich in der Lucy-Szene im Bereich vor der Statue beobachten. Die Begründung für diese Beobachtung liegt darin, dass durch den Schattentest insgesamt weniger Samples zur indirekten Beleuchtung beitragen,

als ohne. In stark verdeckten Bereichen ist die Anzahl an Samples, die den Schattentest bestehen, also besonders gering, womit trotz hoher Sampleanzahl Streifenbildung auftritt. Ein weiteres Problem stellt die Verwendung von MipMapping für die unterschiedlichen Größen der Samples dar. Hierdurch erhält man eine gemittelte Position der Pixellichter, die nicht notwendigerweise zu einer Oberfläche im Raum korrespondiert. Da je Sample nur ein Schattentest durchgeführt wird, wird ein solches Sample entweder akzeptiert oder abgelehnt, obwohl dies nicht für alle Pixellichter zutreffen muss, aus denen das Sample gemittelt wurde. Das Verfahren bleibt also approximativ.

Dennoch liefert der Schattentest einen Mehrwehrt. Abbildung 11 zeigt die Problematik von RSM ohne Schattentest: Die indirekte Beleuchtung aufgrund der Wände durchdringt die beiden Quader. Dies kann die menschliche Tiefenwahrnehmung täuschen, sodass die Quader ohne weitere Tiefenhinweise – wie z. B. Schatten – wirken, als würden sie über dem Boden schweben. Bei aktivem Schattentest verschwindet dieses Problem. Die Quader werfen nun sekundäre (weiche) Schatten innerhalb der indirekten Beleuchtung. Dies wird insbesondere an den Ecken der Quader in Abbildung 11 deutlich.

In den Details der Lucy-Statue (Abbildung 10a) zeigen sich ebenfalls Vorteile durch den Schattentest in Form von Selbstverschattung in den Furchen. In Echtzeitanwendungen approximiert man diesen Effekt mit Ambient-Occlusion Methoden, welche auch Dachsbacher et al. ergänzend zu ihrem Verfahren vorschlagen [DS05].

Gegen die Streifenbildung in stark verdeckten Bereichen kann vorgegangen werden, indem noch mehr Samples genommen werden. Die weitere Problematik, die sich durch MipMapping ergibt, kann verbessert werden, indem die Samplegrößen reduziert werden, oder indem MipMapping ganz weggelassen wird. Dadurch geht aber auch der Vorteil der schnelleren Konvergenz der Approximation verloren. Eine Möglichkeit wäre die Verwendung von RSM mit Schattentest in einem Pathtracer ohne MipMapping, wobei über die gesamte RSM integriert wird. Da die indirekte Beleuchtung pro Sample nur einen Schattentest und Texturzugriffe bedeutet – im Gegensatz zur Berechnung des vordersten Schnittpunktes und Textur- oder Bufferzugriffen sowie erneuter Auswertung des direkten Lichtes in einem herkömmlichen Pathtracer –, könnte dies einen Geschwindigkeitszuwachs bedeuten. Dennoch würde auch dieses angepasste Verfahren nur die erste Indirektion berechnen.

7 Linespace Injection

7.1 Testumgebung

Die Implementation wurde auf dem gleichen System evaluiert wie bereits in Abschnitt 6 beschrieben.

Getestet wurde mit zwei verschiedenen Szenen. Die erste Szene ist die CornellBox, wobei diesmal anstelle der üblichen flächigen Lichtquelle eine

Punktlichtquelle mittig unmittelbar unter der Decke platziert wurde, da flächige Lichtquellen in der Implementation noch nicht unterstützt sind. Die Szene wurde in insgesamt vier Varianten getestet, wobei das Material des hinteren Quaders variiert wurde. Dabei kamen die folgenden Materialien zum Einsatz:

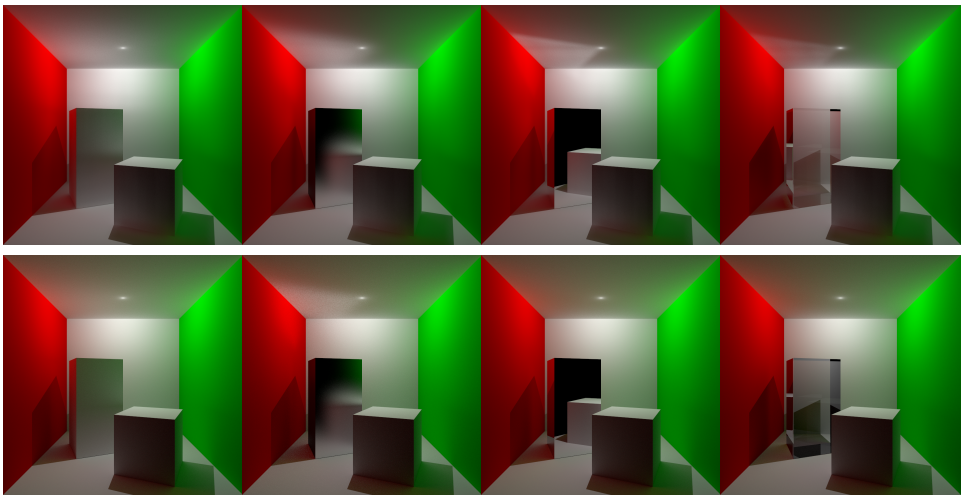
- vollständig diffus (Phong, weiß)
- vollständig spekular (Phong, Glanzexponent 40, weiß)
- perfekte Reflexion
- Glas (Brechungsindex 1.45)

Als zweite Testszene wurde die Sponza Atrium-Szene mit 145185 Eckpunkten und 262267 Dreiecken eingesetzt. Hierbei befindet sich eine Punktlichtquelle mittig über der Szene, sowie zwei weitere Punktlichtquellen im Erdgeschoss links und rechts des Löwenkopfes. Die Szene wurde um Faktor 100 herunterskaliert, da von einer Angabe der Koordinaten in Zentimeter ausgegangen wurde und die Implementation Meter voraussetzt. Damit ergibt sich für die Szene eine Grundfläche von ca. 36m x 22m.

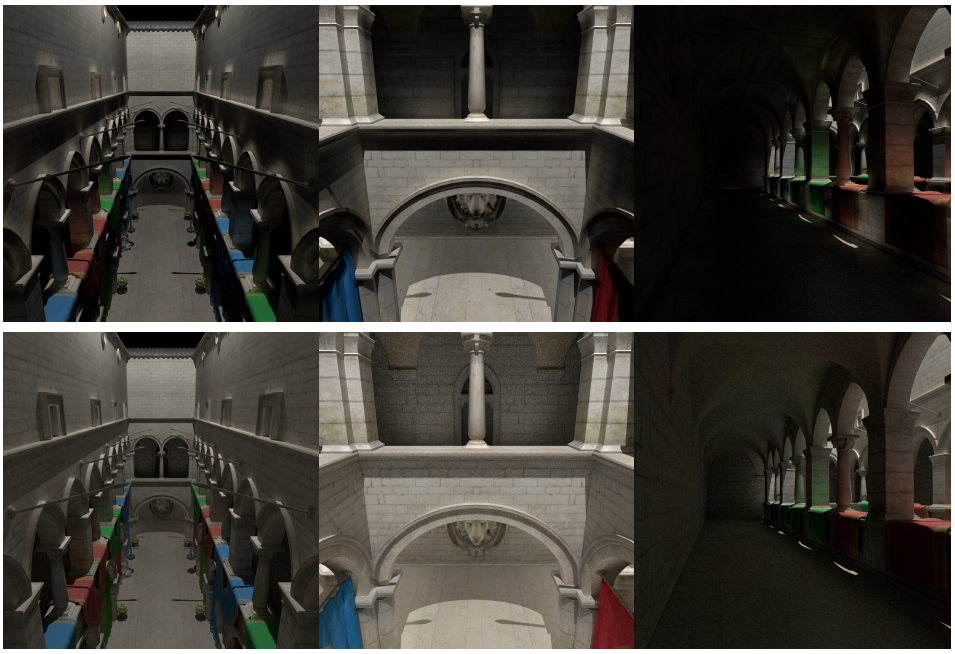
Tabelle 2 zeigt die Renderzeiten der beiden Szenen. Dabei wurde Sponza in Full HD und CornellBox in 1080x1080 gerendert. Bei der Sponza-Szene kamen drei verschiedene Blickwinkel zum Einsatz (siehe Abbildung 12b), weshalb für den Injection-Pass die FPS nur einmal angegeben wurden, da dieser unabhängig von der Kameraposition ist. Er musste also nur einmal ausgeführt werden. Für die CornellBox wurde eine Voxelauflösung von 50x50x50 mit Linespace-Auflösung 4 gewählt, während die Voxelauflösung bei Sponza 68x28x42 betrug. Die Zeiten wurden jeweils gemessen für Pathtracing, den Linespace Injection-Pass, Linespace Injection-Sampling sowie Injection-Sampling mit Pathtracing als Fallback für Schächte, in denen kein Licht gespeichert wurde. In allen vier Varianten wurden Pfade mit maximal vier Schnittpunkten zugelassen. Das Sampling des Linespaces erfolgte dabei jeweils nur für diffuse Materialien, wobei der Radius der Kreisscheibe in der CornellBox 0.07 und in Sponza 0.15 betrug. Die FPS in Tabelle 2 geben die durchschnittliche Anzahl an Samples pro Sekunde an, gerundet auf zwei signifikante Stellen. Für die CornellBox wurden jeweils 1000 Samples genommen, für Sponza 100. Im Injection-Pass wurden insgesamt eine Milliarde Strahlen pro Lichtquelle verschossen.

7.2 Renderzeiten

Tabelle 2 zeigt, dass der Injection-Pass ähnliche Renderzeiten wie Pathtracing aufweist, obwohl er atomare Additionen erfordert und in jedem Durchgang einen Pfad für jede Lichtquelle verfolgt. Der Pathtracer verfolgt stattdessen jeweils nur einen Pfad von der Kamera aus, führt aber zusätzlich an jedem



(a) Obere Reihe: CornellBox mit vier verschiedenen Materialien (diffus, spekular, spiegelnd, Glas) gerendert mit Linespace Injection (1000 Samples, ca. eine Milliarde verschossene Lichtstrahlen). Untere Reihe: Jeweils die gleiche Szene gerendert mit Pathtracing.



(b) Obere Reihe: Sponza aus drei Blickwinkeln gerendert mit Linespace Injection (100 Samples, ca. eine Milliarde verschossene Lichtstrahlen). Untere Reihe: Jeweils der gleiche Blickwinkel gerendert mit Pathtracing.

Abbildung 12: Vergleich zwischen Pathtracing und Linespace Injection für die in Tabelle 2 aufgeführten Szenen.

	FPS			
Szene	PT	LS-INJ	LS-INJ mit Fallback	INJ
Cornell Diffus	21	45	38	17
Cornell spekular	21	42	34	17
Cornell Spiegel	21	43	36	17
Cornell Glas	21	40	32	17
Sponza (1)	0.62	3	2.4	0.65
Sponza (2)	0.6	2.2	1.9	—
Sponza (3)	0.75	3.1	2.3	—

Tabelle 2: Renderzeiten der Testszenen CornellBox mit verschiedenen Materialien und Sponza aus drei Blickwinkeln mit Pathtracing (PT), Injection-Sampling (LS-INJ), Injection-Sampling mit Fallback auf Pathtracing für leere Schächte, sowie dem Injection-Pass selbst (INJ), jeweils in Full HD für Sponza und in 1080x1080 für Cornell.

Schnittpunkt einen Schattentest zu allen Lichtquellen durch. Die Ergebnisse in der Sponza-Szene zeigen, dass trotz drei Lichtquellen die Renderzeit sehr ähnlich ist. In der CornellBox ist der Injection-Pass im Gegensatz dazu etwa 20% langsamer, obwohl nur eine Lichtquelle vorliegt. Eine Problematik bei diesen Vergleichen stellt allerdings die unterschiedliche Ausrichtung und Position der Lichtquellen verglichen mit der Kamera dar, denn bei den Renderzeiten spielt das (frühzeitige) Treffen des Hintergrundes ebenfalls eine Rolle.

Das Linespace Injection-Sampling ist für die CornellBox doppelt so schnell wie Pathtracing, in Sponza sogar bis zu fünf mal so schnell. Dies ist auf den verfrühten Abbruch der Pfade beim Sampling des Linespaces zurückzuführen, was durch die wesentlich höhere Szenenkomplexität in Sponza einen größeren Vorteil einbringt. Im Idealfall müssen nur die vordersten Schnittpunkte aus Sicht der Kamera, sowie die vordersten Schnittpunkte für den nächsten Strahl – allerdings nur innerhalb eines Schachts ohne Traversierung einer Datenstruktur – ermittelt werden. Materialien, die nicht diffus sind, führen allerdings zu mehr Indirektionen. Dies spiegelt sich auch in Tabelle 2 wieder: Das Verfahren ist am schnellsten, wenn nur diffuse Materialien vorhanden sind. Spekulare und reflektierende Materialien verlangsamen das Verfahren, da sie Pfade mit mehr Schnittpunkten hervorrufen. Materialien wie Glas benötigen die längste Rechenzeit, da mindestens zwei Indirektionen nötig sind, bis ein diffuses Material erreicht werden kann.

Wird Pathtracing als Fallback beim Sampling leerer Schächte verwendet, werden nur noch ca. 80% der FPS ohne Fallback erreicht. Das Verfahren bleibt aber schneller als Pathtracing. Dies zeigt, dass selbst nach einer Milliarde verschossener Lichtstrahlen pro Lichtquelle noch Schächte vorhanden sind, in denen kein Licht gespeichert wurde.

7.3 Ergebnisbilder

Abbildung 12 zeigt die Ergebnisbilder zu Tabelle 2, ohne die Bilder für Linespace Injection-Sampling mit Fallback. Dabei wurden die Bilder in HDR gespeichert und mit einem externen Tool [Mü17] auf 8 Bit pro Farbkanal normalisiert, jeweils mit den gleichen Parametern für Pathtracing und Linespace-Injection einer Szene. Da manche Unterschiede in den Bildern nur schwer zu erkennen sind, zeigt Abbildung 13 den relativen quadratischen Fehler von Linespace Injection-Sampling zu Pathtracing.

In der CornellBox zeigen sich in allen Bildern Helligkeitsunterschiede in den Ecken und an den Kanten, sowie in den Schatten. Die Linespace Injection liefert hier hellere Ergebnisse. Abbildung 14a zeigt einen vergrößerten Ausschnitt des vorderen Quaders. Die Kante zwischen Boden und Quader ist mit Linespace Injection wesentlich heller. Bei Verwendung des Fallbacks auf Pathtracing ist das Ergebnis sogar nochmal heller. Die vordere Ecke des Quaders vermittelt bei Linespace Injection den Eindruck, als würde sie schweben. Hier liegt ein Fehler in der Verdeckung des indirekten Lichtes vor. Diese Fehler lassen sich auch an anderen Stellen erkennen: Während beim Pathtracing korrekte sekundäre Schatten in der indirekten Beleuchtung auftreten, fehlen diese bei Linespace Injection oder fallen wesentlich schwächer aus. Einen großen Faktor bei diesen Fehlern spielt die zufällige Verschiebung der Sampleposition auf einer Kreisscheibe, die eine Weichzeichnung hervorruft und somit die indirekte Beleuchtung auch um Ecken und Kanten herum verwischt. Ein weiterer Faktor stellt die Diskretisierung der Raumrichtungen durch den Linespace dar. Diese Effekte lassen sich auch anhand der Reflexionskaustiken an der Decke in der CornellBox erkennen, die größer ausfallen als sie sein müssten und an den Grenzen weichgezeichnet sind. Des Weiteren stellt die Ungenauigkeit bei Beleuchtungseffekten, die innerhalb eines Linespaces oder Schachts stattfinden, ein Problem dar. Unbekannte Faktoren lassen sich auf dieser Basis allerdings nicht ausschließen.

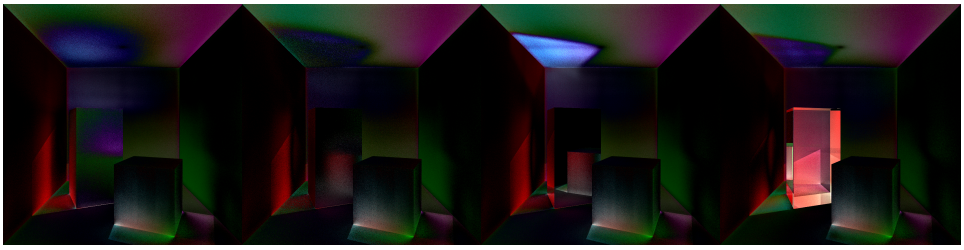


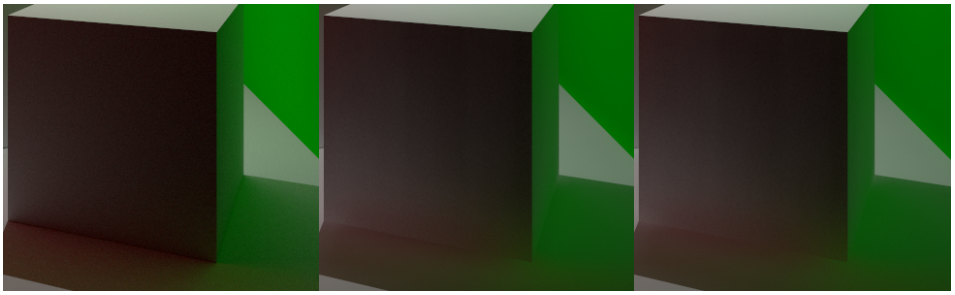
Abbildung 13: Quadratischer Fehler bei Linespace Injection-Sampling relativ zu Pathtracing. Von links nach rechts: diffus, spekular, spiegelnd, Glas.

Nicht alle Unterschiede in den Bildern lassen sich mit den erwähnten Faktoren erklären. Dazu zählen insbesondere farbliche Unterschiede in der CornellBox: Am rechten und linken Rand der Decke tritt die grüne bzw. rote

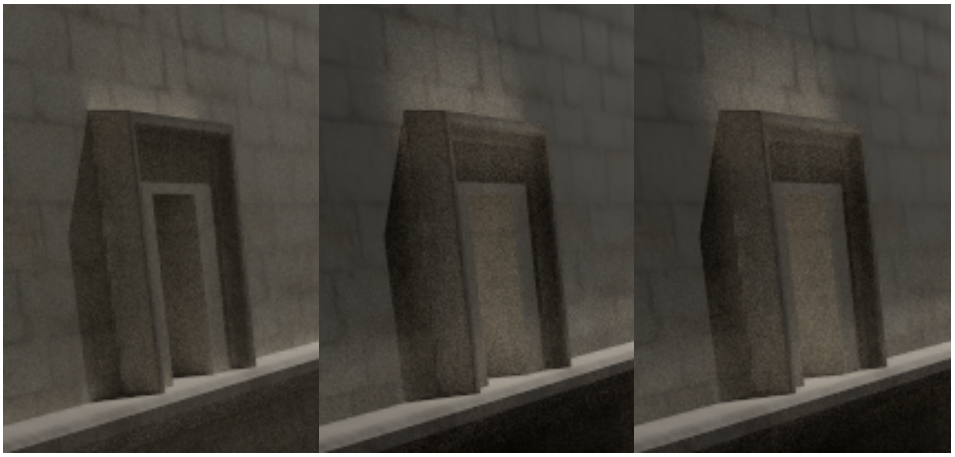
indirekte Beleuchtung bei Linespace Injection stärker hervor, als bei Pathtracing. Im Gegensatz dazu fällt bei Linespace Injection das grüne indirekte Licht am linken und rechten Rand des Bodens schwächer aus, was insbesondere im linken Bereich des Schattens des hinteren Quaders in allen vier Varianten der Szene deutlich wird. Hier lässt sich bei Linespace Injection mit bloßem Auge nur noch eine rote indirekte Beleuchtung erkennen. In der diffusen CornellBox fehlt auch ein Großteil der indirekten grünen Beleuchtung auf der Vorderseite des hinteren Quaders. Die Unterschiede in der Sponza-Szene (Abbildung 12b) sind noch auffälliger. Hier erzeugt die Linespace Injection wesentlich dunklere indirekte Beleuchtung. Im Gegensatz dazu ist allerdings die farbige indirekte Beleuchtung durch die Banner stärker. Die Ursache dieser Fehler liegt in der Normalisierung der Schacht-Informationen, zusammen mit der Verwendung mehrerer Lichtquellen. Tatsächlich müsste man die indirekte Beleuchtung ausgehend von jeder Lichtquelle getrennt normalisieren und aufaddieren. Das vorgestellte Verfahren verliert allerdings die Information der Zugehörigkeit zur Lichtquelle und normalisiert damit bereits das addierte Ergebnis. Damit fließen die indirekten Beleuchtungsstärken der einzelnen Lichtquellen nur anteilig in das Ergebnis mit ein, anstatt voll. Das Resultat ist ein dunkleres Bild. Somit eignet sich das Verfahren aktuell nur für Szenen mit einer einzigen Lichtquelle, bzw. erfordert es einen getrennten Buffer pro Lichtquelle.

Abbildung 14b zeigt einen vergrößerten Ausschnitt eines Fensters in der Sponza-Szene. Der helle Bereich an der Oberkante des Fensters fällt hier mit Linespace Injection wesentlich größer aus und durchdringt die Kanten. Letzteres Problem wurde bereits für die CornellBox beschrieben und ist insbesondere auf die Weichzeichnung durch die Verschiebung der Samplepositionen zurückzuführen. Der wesentlich größere hellere Bereich im Vergleich zu Pathtracing liegt hier zusätzlich an der verhältnismäßig geringen Auflösung der Voxelisierung relativ zu Größe und Detailgrad der Szene. Die Auflösung hätte also wesentlich größer gewählt werden müssen. Dies war allerdings aufgrund des hohen Speicherbedarfs auf dem verwendeten System nicht möglich. Zwar bietet die verwendete GPU genug Speicher für größere Voxelauflösungen, allerdings lag eine treiberseitige Limitierung von maximal 2 GB pro Buffer vor, die die volle Nutzung dieses Speichers verhindert. Die Implementation müsste also so angepasst werden, dass der große Buffer in mehrere kleine Buffer aufgeteilt wird. Aufgrund der niedrigen Voxelauflösung weicht auch die Helligkeit der Nische im Fenster sehr stark ab, da sie schmaler ist als ein einzelnes Voxel. Dadurch verschmilzt die indirekte Beleuchtung des umgebenden, helleren Bereiches mit der Nische und diese erscheint im Ergebnisbild wesentlich heller.

Im Gegensatz zur Nahaufnahme aus der CornellBox (Abbildung 14a) lässt sich in Abbildung 14b eine Verbesserung feststellen, wenn Linespace Injection mit Fallback verwendet wird. Hier stimmen die Helligkeitsverhältnisse an der Kante auf der Schattenseite des Fensters besser mit Pathtracing überein, als bei Linespace Injection ohne Fallback, während dies in der CornellBox genau



(a) CornellBox (diffus)



(b) Sponza

Abbildung 14: Nahaufnahmen aus CornellBox und Sponza. Von links nach rechts: Pathtracing, Linespace Injection, Linespace Injection mit Fallback auf Pathtracing für leere Schächte.

umgekehrt war. Das legt die Vermutung nahe, dass die Auswirkungen bei aktivem Fallback von Szene und Voxelauflösung abhängig sind, wobei auch der Radius der Kreisscheibe eine Rolle spielen dürfte.

Nichtsdestotrotz lassen sich auch Vorteile von Linespace Injection gegenüber Pathtracing in den Bilder erkennen. In Abbildung 12a erzeugt Linespace Injection bei spiegelndem Material und bei Glas Relexionskaustiken an der Decke, die mit Pathtracing nicht möglich sind. Der Grund liegt in der Verwendung einer Punktlichtquelle. Da ein Pathtracer für spiegelnde Materialien und Glas keine direkte Beleuchtung ermitteln kann, können solche Kaustiken nur auftreten, wenn Schnittpunkte mit der Lichtquelle möglich sind. Dies ist bei unendlich kleinen Lichtern zwar theoretisch machbar, praktisch aber hinfällig, da die Wahrscheinlichkeit eine unendlich kleine Lichtquelle zu treffen gegen 0 geht. Bei Linespace Injection werden im Gegensatz dazu auch Pfade von der Lichtquelle aus verfolgt. Der Schnittpunkt mit der Lichtquelle ist in diesen Pfaden also automatisch als Startpunkt der Pfade vorhanden.

Ein weiterer Vorteil wird in der CornellBox mit Glasquader deutlich. Bei Pathtracing treten hier dunkle Stellen im Glas auf, die heller sein müssten, da das Material kein Licht absorbiert. Dies liegt daran, dass gläserne Objekte eine hohe Anzahl an Indirektionen erfordern, um korrekt dargestellt zu werden. Ein beliebiger Pfad durch ein gläsernes Objekte benötigt bereits mindestens zwei Schnittpunkte. Damit bleiben für die eigentliche Beleuchtung der Bereiche hinter dem Glas weniger Indirektionen übrig, was dazu führen kann, dass diese dunkler aussehen, als sie sollten. Bei Linespace Injection verschwinden die dunklen Bereiche. Hier wurde zwar die gleiche maximale Anzahl Indirektionen aus Sicht der Kamera verwendet, allerdings gilt dies auch für die injizierten Pfade, die von der Lichtquelle aus verfolgt werden. Beim Sampling des Linespace wird also ein (gemittelter) Lichtpfad mit dem Kamerapfad verbunden, das Resultat sind Pfade, die bis zu doppelt so viele Indirektionen aufweisen, was das Rendering von Glas verbessert.

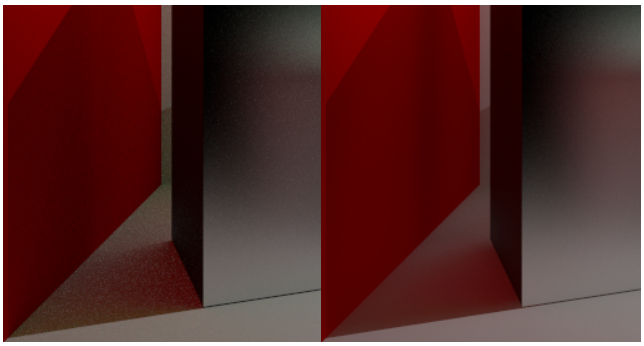


Abbildung 15: Nahaufnahme aus der CornellBox (spekular). Links: Pathtracing; Rechts: Linespace Injection. Pathtracing führt bei gleicher Sampleanzahl zu stärkerem Rauschen.

Weiterhin ergeben sich Vorteile im Rauschverhalten, also der Varianz. Abbildung 15 zeigt einen Ausschnitt aus der CornellBox mit spekularem Quader. Selbst nach 1000 Samples ist bei Pathtracing noch ein starkes Rauschen zu erkennen. Das Bild, das mit Linespace Injection gerendert wurde, weist im Gegensatz dazu ein wesentlich geringeres Rauschen auf. Dies liegt daran, dass die indirekte Beleuchtung diffuser Materialien sich hier direkt aus den relevanten Schächten ergibt, in denen sie bereits vorberechnet abgespeichert ist. Damit vereinfacht sich das Beleuchtungsintegral auf eine Diskretisierung und die Varianz fällt geringer aus.

In den bisher diskutierten Abbildungen beschränkte sich ein Sampling des Linespaces für die indirekte Beleuchtung auf diffuse Materialien. Für die Bilder in Abbildung 16 wurde der Linespace zusätzlich bei spekularen Materialien verwendet. Dabei führt bereits ein Glanzexponent von 2 zu großen Abweichungen im Vergleich zu Pathtracing. Mit zunehmendem Glanzexponenten wird der Fehler immer größer. Ein Grund dafür ist die Weichzeichnung durch das

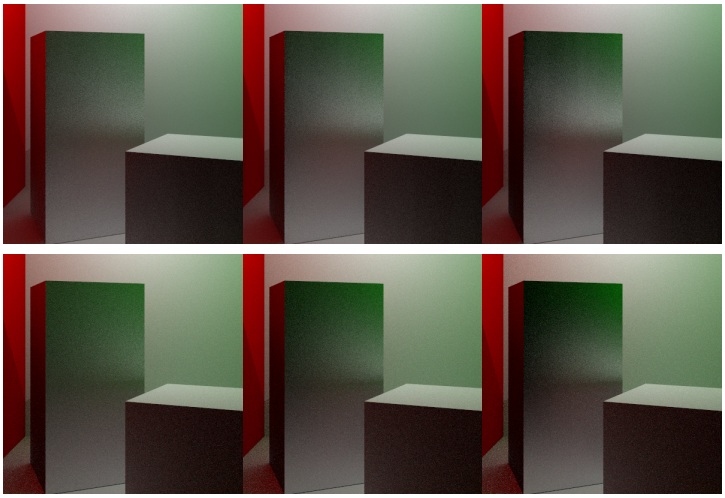


Abbildung 16: Quader in CornellBox mit Glanzexponent 2, 5, 10 (von links nach rechts). Oben: Light Injection mit Sampling des Linespaces zusätzlich für spekulare Materialien anstatt nur für diffuse Materialien. Unten: Pathtracing.

Verschieben der Sampleposition. Da bei Glanzreflexionen über einen kleineren Raumwinkel integriert wird, fällt diese Weichzeichnung stärker ins Gewicht. Ein weiteres Problem ist die Diskretisierung in Schächte, wie in Abbildung 17 veranschaulicht. Weil die indirekte Beleuchtung aus den Schächten ausgelesen wird, sind die Raumwinkel, über die integriert wird, auf diskrete Schritte beschränkt. Dadurch wird Licht aus einem größeren Bereich eingesammelt, als erwünscht. Die Problematik ließe sich verbessern, indem man die Auflösung des Linespaces erhöht und damit die Diskretisierung verfeinert. Damit steigt allerdings auch der Speicherbedarf biquadratisch an, weshalb man sehr schnell an die Grenzen der Speicherkapazität stößt. Um dies auszugleichen müsste man dann die Voxelauflösung reduzieren, was kontraproduktiv wäre. Das Sampling des Linespaces ist also nicht für Glanzreflexionen geeignet.

Die Weichzeichnung durch das Verschieben der Samplepositionen auf einer Kreisscheibe wurde bereits mehrfach als Ursache für Fehler in den Bildern erwähnt. Dennoch ist sie zwingend notwendig, wie Abbildung 18 zeigt. Ohne die Weichzeichnung treten offensichtliche Artefakte auf, die Gitter- und Streifenmuster bilden. In der Nahaufnahme der Wand kann man in den Artefakten sowohl die Grenzen der einzelnen Linespaces, als auch die Unterteilung dieser in Patches erkennen. Innerhalb eines solchen Patches ist jeweils die linke Seite indirekt rot beleuchtet, während die rechte Seite indirekt grün beleuchtet wird. Am oberen Rand eines Linespaces liegt zudem eine besonders helle Beleuchtung vor. Die Nahaufnahme des hinteren Quaders zeigt ebenfalls Artefakte, diese fallen aber schwächer aus. Im Gegensatz zur Wand liegt hier die Geometrie nicht direkt auf einer Voxelgrenze.

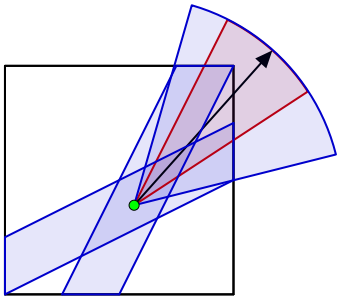


Abbildung 17: Fehler beim Sampling des injizierten Lichtes durch die Diskretisierung des Linespaces. Der Rote Kegel entspricht den Raumrichtungen, aus denen Licht eingesammelt wird, der blaue Kegel spiegelt die Raumrichtungen wieder, die beim Sampling der zugehörigen Schächte tatsächlich integriert werden.

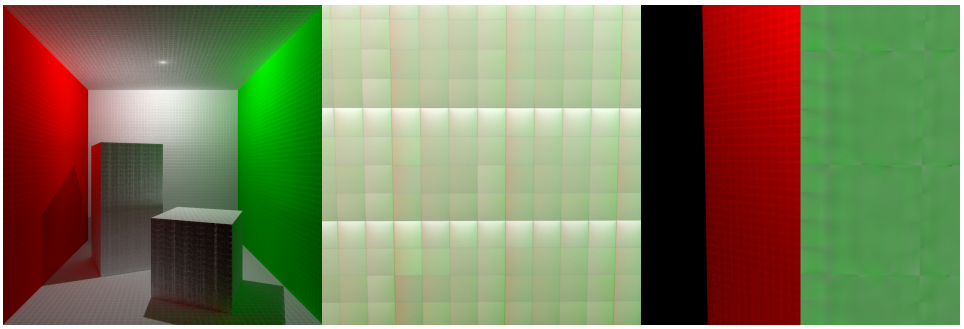


Abbildung 18: Artefakte bei Linespace Injection ohne zufälliges Verschieben der Samplepositionen auf einer Kreisscheibe. Von links nach rechts: CornellBox (diffus), Nahaufnahme der hinteren Wand, Nahaufnahme der rechten Seite des hinteren Quaders.

Abbildung 19 liefert die Erklärung für die Artefakte. Jeder Schacht sammelt im Injection-Pass Licht aus festen Raumrichtungen ein, und zwar genau alles Licht, das durch diesen Schacht verläuft und dabei auf Geometrie trifft. Dies stimmt allerdings nicht zwangsläufig mit dem Licht überein, das man beim Sampling der diffusen BRDF erwartet. Die linke Grafik zeigt anhand von drei Schächten, für welche Raumrichtungen diese ausgewählt werden, mit anderen Worten die Wahrscheinlichkeit. Dies stimmt nicht mit dem überein, was in den Schächten gespeichert wurde. Verschiebt man den grünen Punkt in der linken Grafik, ändern sich die Verhältnisse, während das gespeicherte Licht in den Schächten gleich bleibt. Damit ergibt sich eine Asymmetrie beim Sampling des Linespaces. Im Falle der Außenwände in der CornellBox kommt erschwerend hinzu, dass diese auf der Grenze eines Linespaces liegen. Damit werden für eine gegebene Sampleposition nur Schächte mit gleichem Start-Patch ausgewählt. Bewegt man sich innerhalb eines Patches weiter nach links, verschieben sich die Wahrscheinlichkeiten, sodass die indirekte

Beleuchtung durch die linke Wand überwiegt. Bewegt man sich nach rechts, dominiert jene der rechten Wand. Durch das zufällige Verschieben der Samplepositon auf einer Kreisscheibe wird dieses Problem umgangen, da nun ein größerer Bereich an Samplepositionen integriert wird und sich die Wahrscheinlichkeiten der Schächte ausgleichen. Außerdem sind dann in solchen Extremfällen, wie Geometrie die auf den Voxelgrenzen liegt, auch Schächte mit unterschiedlichem Start-Patch möglich.

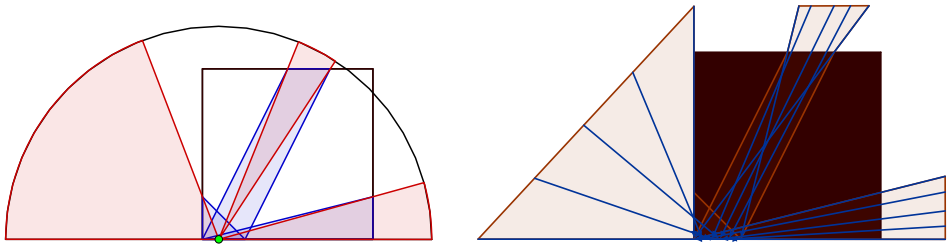


Abbildung 19: Veranschaulichung von drei Schächten in einem Linespace mit Auflösung 4 in 2D. Links: Sampling eines diffusen Oberflächenpunktes (grün). Die roten Kegel repräsentieren die Wahrscheinlichkeit, mit der die zugehörigen Schächte (blau) gesamplet werden. Verschieben des Punktes verändert die Wahrscheinlichkeiten. Rechts: Veranschaulichung des Lichts, das die selben Schächte im Injection-Pass einsammeln.

Eine Möglichkeit um ohne die Weichzeichnung auszukommen – oder zumindestens mit einer schwächeren Weichzeichnung – wäre es, die tatsächlichen Raumwinkel der End-Patches zu berechnen, mit anderen Worten die Fläche, die diese auf eine Einheitskugel projiziert einnehmen. Auf diese Weise könnten die Unterschiede in den Samplingwahrscheinlichkeiten für jeden Schacht herausgerechnet werden. Dabei müsste zusätzlich beachtet werden, welchen Raumwinkel das Licht einnimmt, das der Schacht tatsächlich einsammelt.

Teil V
Fazit

In dieser Arbeit wurden zwei verschiedene Verfahren zur Berechnung der globalen Beleuchtung vorgestellt. Es konnte gezeigt werden, dass Reflective Shadow-Maps auf moderner Hardware sogar in voller Auflösung ohne die von Dachsbacher et al. verwendeten Beschleunigungen echtzeitfähig sind. Die Ergebnisbilder weisen kein Bildrauschen auf, stattdessen führen niedrige Sampleanzahlen zu Streifenbildung. Die Erweiterung um einen Schattentest zeigt klare Verbesserungen, Verdeckungsbehandlung ist also auch mit einer RSM möglich. Allerdings ist das Verfahren dadurch nicht mehr echtzeitfähig und höhere Sampleanzahlen sind von Nöten, weshalb in Echtzeitanwendungen die Verwendung von Screen-Space Ambient Occlusion oder ähnlichen Methoden noch das Mittel der Wahl darstellt. In zukünftigen Arbeiten wäre es also insbesondere interessant, die Verdeckungsbehandlung zu beschleunigen. Der größte Nachteil des Verfahrens liegt darin, dass nur die erste Indirektion berechnet werden kann.

Die vorgestellte neue Methode der Linespace Injection zeigt einen mit der Szenenkomplexität zunehmenden Geschwindigkeitsvorteil im Vergleich zu traditionellem Pathtracing bei größtenteils diffusen Szenen. Damit wird eine Verwendung in Echtzeitanwendungen eine realistische Möglichkeit. Eine Vorberechnung, der Injection-Pass, ist dazu allerdings vonnöten. Die Vorteile kommen also erst dann zur Geltung, wenn die Szenen statisch sind und die Vorberechnung somit nur ein einziges mal unabhängig von der Kamera erfolgen muss.

Das Verfahren ermöglicht ähnliche Vorteile wie die des Bidirektionalen Pathtracings. Größtenteils indirekt beleuchtete Bereiche weisen eine geringere Varianz – und damit ein geringeres Rauschen – auf. Außerdem werden Kaustiken möglich. Dennoch ist das Ergebnis nur approximativ, wobei insbesondere die Weichzeichnung ein großes Problem darstellt. Das Verfahren bietet in diesem Bereich aber noch viel Potential zur Verbesserung, was interessant für weitere Forschung ist, ebenso wie die weitere Beschleunigung des Verfahrens.

Das größte Problem des Verfahrens ist der große Speicherverbrauch, der die Einsetzbarkeit insbesondere für sehr große Szenen einschränkt. Des Weiteren skaliert das Verfahren schlecht mit der Anzahl an Lichtquellen, da hierfür ein getrennter Buffer pro Lichtquelle notwendig wäre. Damit würde der Speicherbedarf zusätzlich linear mit der Anzahl an Lichtern steigen.

Nichtsdestotrotz zeigt das Verfahren der Linespace Injection sehr viel Potential für die schnellere und varianzreduzierte Berechnung der globalen Beleuchtung mit beliebig vielen Indirektionen. Wenngleich es noch unausgereift ist, macht es dies für zukünftige Forschungen in diesem Bereich zu einem attraktiven Verfahren.

Literatur

[DF97] DeYoung, Joel ; Fournier, Alain: Properties of tabulated bidirectional reflectance distribution functions. In: *Graphics Interface* Bd. 97, 1997, S. 47–55

[DS05] Dachsbacher, Carsten ; Stamminger, Marc: Reflective shadow maps. In: *Proceedings of the 2005 symposium on Interactive 3D graphics and games* ACM, 2005, S. 203–231

[FTI86] Fujimoto, Akira ; Tanaka, Takayuki ; Iwata, Kansei: Arts: Accelerated ray-tracing system. In: *IEEE Computer Graphics and Applications* 6 (1986), Nr. 4, S. 16–26

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

[Kaj86] Kajiya, James T.: The rendering equation. In: *ACM Siggraph Computer Graphics* Bd. 20 ACM, 1986, S. 143–150

[Khr17] The Khronos Group Inc. (Hrsg.): *The OpenGL® Graphics System: A Specification (Version 4.5 (Core Profile))*. Version 2.0. The Khronos Group Inc., Juni 2017

[KKM17] Keul, Kevin ; Klee, Nicolas ; Müller, Stefan: Soft Shadow Computation using Precomputed Line Space Visibility Information. In: *25th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2017

[KLM16] Keul, Kevin ; Lemke, Paul ; Müller, Stefan: Accelerating Spatial data structures in ray tracing through precomputed line space visibility. In: *24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2016

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

[L'e96] L'ecuyer, Pierre: Maximally equidistributed combined Tausworthe generators. In: *Mathematics of Computation of the American Mathematical Society* 65 (1996), Nr. 213, S. 203–213

[Mü17] Müller, Thomas: *tev: A high dynamic range (HDR) image comparison tool for graphics people.* `https://github.com/Tom94/tev`, September 2017. – Version 1.5

[Ngu07] Nguyen, Hubert: *Gpu gems 3*. Addison-Wesley Professional, 2007

[Pho75] Phong, Bui T.: Illumination for computer generated pictures. In: *Communications of the ACM* 18 (1975), Nr. 6, S. 311–317

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

[Sch94] Schlick, Christophe: An Inexpensive BRDF Model for Physically-based Rendering. In: *Computer Graphics Forum* 13 (1994), Nr. 3, 233–246. `http://dx.doi.org/10.1111/1467-8659.1330233`. – DOI 10.1111/1467–8659.1330233. – ISSN 1467–8659

[Sch16] Schrage, Robin: *GPU basiertes Ray Tracing mit Bounding Volume Hierarchie.* 2016

[TS+05] Thrane, Niels ; Simonsen, Lars O. u. a.: A comparison of acceleration structures for GPU assisted ray tracing. 2005. – Forschungsbericht

[Whi79] Whitted, Turner: An improved illumination model for shaded display. In: *ACM SIGGRAPH Computer Graphics* Bd. 13 ACM, 1979, S. 14

[Wil78] Williams, Lance: Casting curved shadows on curved surfaces. In: *ACM Siggraph Computer Graphics* Bd. 12 ACM, 1978, S. 270–274