Voxel Cone Tracing in OpenGL zur Darstellung von voll-dynamischer globaler Beleuchtung in Echtzeit

Bachelorarbeit

Roland Fischer

Matrikelnummer: 2484425

1. März 2017

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Gabriel Zachmann
2. Gutachter: Prof. Dr.-Ing. Udo Frese

Erklärung

Ich versichere, den Bachelor-Report ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 1. März 2017

.....

(Roland Fischer)

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel wird einleitend für die Bachelorarbeit auf die Motivation und die Problemstellung sowie Zielsetzung eingegangen. Daraufhin wird die weitere Struktur des Dokumentes vorgestellt.

1.1 Motivation

In der heutigen Welt haben Computer einen immer größeren Einfluss, viele Bereiche des Lebens und der Industrie sind digitalisiert oder werden von informationstechnischen Systemen oder Software unterstützt. Ein stetig wachsender Teilbereich der Informatik ist die Computergrafik, dessen Ziel es ist, mithilfe von Computern Bilder zu erzeugen und diese zu bearbeiten. Um diese Bilder zu erstellen, wird in der Regel aus digitalen Modellen von Objekten eine virtuelle Umgebung, auch Szene genannt, erstellt und darauf verschiedene Techniken zur Berechnung und Darstellung des gewünschten Bildes angewandt.

Die Beleuchtung einer solchen digitalen Szene, wie sie z.B. in Computerspielen oder dreidimensionalen Visualisierungen vorkommt, hat einen erheblichen Einfluss auf die Qualität des schließlich dargestellten Bildes. Um, je nach Anwendungsgebiet und Ziel, realistische Ergebnisse oder visuell ansprechende Effekte zu erzeugen, sind aufwendige und komplexe Algorithmen und Verfahren unerlässlich. Nicht nur sind diese sehr rechenaufwendig, was vor allem in Echtzeitanwendungen das Hauptproblem darstellt, besonders, dass mit der Zeit die Ansprüche an die visuelle Darstellung steigen, lässt die Anforderungen an die Algorithmen kontinuierlich wachsen. Verschärfend kommt hinzu, dass laut dem Gesetz des sinkenden Grenzertrags der Rechenaufwand um noch deutlich sichtbare Verbesserungen zu erzielen exponentiell steigt.

1.2 Problemstellung und Zielsetzung

Um die oben erwähnten Anforderungen an die Rechenleistung zu senken und somit echtzeitfähige Berechnungen durchführen zu können, muss auf Präzision verzichtet werden, Approximationen vorgenommen oder statische Teile der Beleuchtung vorberechnet werden. Ein aktuelles Problem, was mit dieser Arbeit untersucht werden soll, ist die dynamische Echtzeitberechnung von globaler Beleuchtung.
Das Ziel dieser Arbeit ist es somit, die aktuellsten Techniken zur Berechnung von dynamischer und globaler Beleuchtung zu betrachten und eine voxelbasierte (vgl. 2.4.2) Variante zu implementieren. Die Kategorie der auf Voxeln basierenden Algorithmen sind sehr vielversprechend, da sie einen guten Kompromiss zwischen Geschwindigkeit und Präzision und somit Realismus darstellen. Eine solche Abwägung zu treffen ist immer eine Herausforderung, aber durch die begrenzte Rechenleistung unumgänglich. Für die Implementierung wird die Computergrafik-Programmierschnittstelle OpenGL verwendet, da diese weit verbreitet und plattformunabhängig ist. Diese käme so also vielen Anwendern, in der Industrie sowie im privaten und wissenschaftlichen Umfeld, zugute. Zudem bietet OpenGL in den aktuelleren Versionen sehr hilfreiche, wenn nicht sogar zwingend erforderliche Funktionen sowie ein Maximum an Freiheit was die Implementierung betrifft.

1.3 Struktur der Arbeit

In Kapitel 2 wird auf die theoretischen Grundlagen eingegangen, die dieser und den vorherigen Arbeiten zugrunde liegen. Die wichtigsten Begriffe, Techniken und Konzepte werden definiert und erläutert. Im darauffolgenden Kapitel 3 werden die Arbeiten, auf denen diese aufbaut, thematisiert und Unterschiede untereinander aufgezeigt. Anschließend folgt das Kapitel 4, in dem die hier eingesetzten Algorithmen besprochen werden und die Entscheidungsfindung für und gegen die Übernahme von bestimmten Teilaspekten dargelegt wird. Darauf folgt die detaillierte Beschreibung der eigenen Implementierung. Im Kapitel 5 werden die Ergebnisse der Implementierung vorgestellt und evaluiert, einerseits für sich genommen und andererseits im Vergleich zu bestehenden Implementierungen zu vergleichbaren Verfahren. Zum Schluss folgt das Kapitel 6, in dem ein abschließendes Fazit gezogen wird, sowie ein Ausblick auf weiterführende Arbeit und Optimierungspotential gegeben wird.

Kapitel 2

Grundlagen

Durch dieses Kapitel werden die grundlegenden Konzepte und Techniken vermittelt, die relevant für das Verständnis der folgenden Kapitel und der Arbeit an sich sind.

2.1 Theorie des Lichttransfers

In der Natur ist das Verhalten des Lichts sehr komplex und exakte physikalische Modelle, welche dieses beschreiben, sind zu rechenintensiv, um sie in der Computergrafik einzusetzen. Meistens ist diese Präzision auch nicht erforderlich. So wird z.B. für gewöhnlich eine einfache Version des Teilchenmodells angenommen, sprich, dass Licht aus Partikeln besteht. Die Ausbreitung des Lichts wird ausschließlich als Aussenden von Lichtstrahlen modelliert.

Zwei wichtige radiometrische Größen, die zur Quantisierung von Licht verwendet werden, sind die Strahldichte (Radianz, englisch radiance) und die Bestrahlungsstärke (Irradianz, englisch irradiance).

2.1.1 Strahldichte/Radianz

Laut (Gebhardt, 2003, pp. 1-2) gibt die Strahldichte an, wie viel Lichtenergie auf eine bestimmte Fläche aus einer bestimmten Richtung auftritt, bzw. ausgesendet wird. Mit ihr lässt sich also beschreiben, wie hell eine Fläche auf einen Betrachter wirkt. Die Formel lautet:

$$L(\theta_r, \phi_r; \theta_i, \phi_i) = \frac{\delta\Phi(\theta_r, \phi_r; \theta_i, \phi_i)}{\delta A cos(\theta_r; \theta_i)\delta\omega_{r;i}} \tag{2.1}$$

Dabei beschreiben θ und ϕ den Polar- und Azimutwinkel (Vertikal- und Horizontalwinkel), $\delta\Phi(\theta_r, \phi_r, \theta_i, \phi_i)$ ist die richtungsabhängige, von der Oberfläche ausgehende bzw. eingehende Strahlungsleistung, $\delta A cos(\theta_r; \theta_i)$ das in Abstrahlrichtung projizierte Flächenelement und $\delta\omega_{r;i}$ das Raumwinkelelement.

In Abbildung 2.1 ist das differentielle Raumwinkelelement die dick umrahmte Fläche, durch das die Strahlungsleistung auf die Oberfläche trifft.

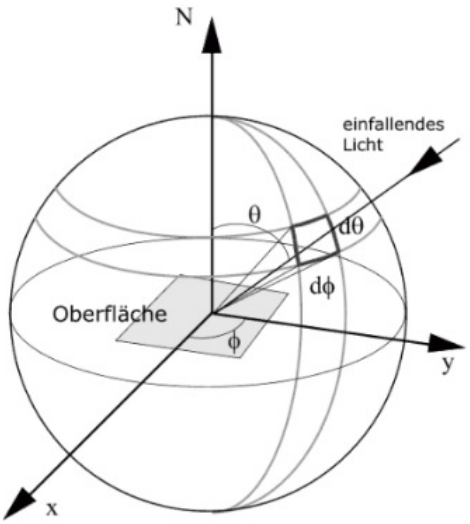

Abbildung 2.1 Strahlungsleistung trifft durch ein differentielles Raumwinkelelement auf eine Oberfläche, wo es als Strahldichte wahrgenommen wird. (Gebhardt, 2003, p. 2)

Eine wichtige Eigenschaft der Strahldichte ist, wie man an der Formel ablesen kann, dass sie abhängig von Beleuchtungs- und Betrachtungswinkel, aber nicht entfernungsabhängig ist. Die Energie ist auf allen Punkten einer Linie im Raum identisch, sofern man von einem Vakuum ausgeht.

2.1.2 Bestrahlungsstärke/Irradianz

Die Bestrahlungsstärke ist, im Gegensatz zur Strahldichte, unabhängig vom Betrachtungswinkel und beschreibt die Höhe der auf einer Fläche auftreffenden Strahlungsleistung. Sie nimmt nach dem photometrischen Entfernungsgesetz quadratisch mit der Entfernung zur Lichtquelle ab. Die Formel lautet:

$$E(\theta_i, \phi_i) = \frac{\delta\Phi_i(\theta_i, \phi_i)}{\delta A} \tag{2.2}$$

Die Bestrahlungsstärke ergibt sich demnach aus der auftreffenden Strahlungsleistung, welche abhängig von den Einfallswinkeln ist, geteilt durch die betrachtete Oberfläche.

2.1.3 Lichtreflexion und -brechung

Treffen Lichtstrahlen auf die Oberfläche eines Objektes, können sie reflektiert, absorbiert und gebrochen werden, wie in Abb. 2.2 dargestellt ist. Dabei spielt die Oberflächenbeschaffenheit des Objektes eine große Rolle dafür, ob und zu welchen Teilen Reflexion und Lichtbrechung stattfinden und wie sie sich auf die Lichtstrahlen auswirken.
Man unterscheidet zwischen diffuser und glänzender Reflexion. Im Fall der diffusen Reflexion wird eine raue Oberfläche angenommen, wodurch das Licht über die Hemisphäre über der Oberfläche reflektiert wird, wohingegen bei der glänzenden Reflexion von einer glatten Oberfläche ausgegangen wird. So wird der Lichtstrahl nur in einen kleinen Bereich um die ideale Ausgangsrichtung reflektiert.
Die Brechung oder auch Refraktion von Licht kann bei dem Übergang von einem Medium in ein anderes mit unterschiedlicher Dichte auftreten, sofern dieses eine gewisse Durchlässigkeit bzw. Transparenz aufweist.

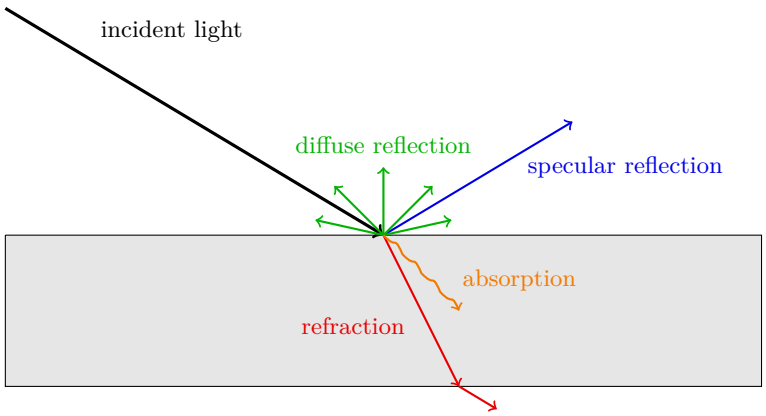

Abbildung 2.2 Diffuse und glänzende Reflexion (diffuse and specular reflection), Absorption (absorption) und Brechung (refraction) von Lichtstrahlen.

Das Oberflächenverhalten lässt sich mathematisch, mithilfe der oben definierten Größen, durch eine sogenannte **Bidirektionale Reflektanzverteilungsfunktion (BRDF)** formulieren, wie (Gebhardt, 2003, pp. 1-3) darlegt. Mit dieser kann für jeden Punkt einer Oberfläche, abhängig von Lichteinfalls- und Reflexionswinkel, ermittelt werden, wie viel Licht in diese Richtung reflektiert wird. Angegeben wird dies durch die differentielle Strahldichte geteilt durch die differentielle Bestrahlungsstärke. Die BRDF beschreibt also das Verhältnis zwischen eintreffender Bestrahlungsstärke und ausgesendeter Strahlungsdichte. Die allgemeine Formel, wie auch in (Oren and Nayar, 1995, p. 6) illustriert, lautet:

$$f_r(\theta_i, \phi_i; \theta_r, \phi_r) = \frac{\delta L(\theta_r, \phi_r; \theta_i, \phi_i)}{\delta E(\theta_i, \phi_i)} \tag{2.3}$$

In der Abbildung 2.3 sind der Aufbau und alle Komponenten der allgemeinen BRDF skizziert, wobei dort $\hat{S}$ die eingehende und $\hat{V}$ die reflektierte Strahldichte darstellt.

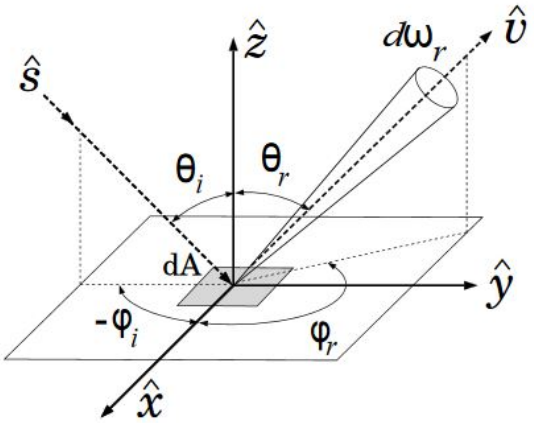

Abbildung 2.3 Geometrie und Größen einer BRDF. (Oren and Nayar, 1995, p. 6)

2.1.4 Arten von Lichtquellen

Man unterscheidet Lichtquellen in vier grundlegende Arten, wie z.B. auch in (Hughes et al., 2013, pp. 377-381) nachzulesen ist:

- Point Light
  Point Lights sind Lichtquellen, dessen Ausdehnung in Relation zur beleuchteten Szene gegen Null geht (z.B. ein Punkt im Raum ohne physisches Volumen) und die in alle Richtungen gleich verteilt Energie in Form von Licht abgeben. Die Abstrahlcharakteristik lässt sich in etwa mit der einer Glühbirne vergleichen. Die am Objekt eintreffende Strahldichte hängt von der Entfernung zur Lichtquelle ab.

- Directional Light
  Directional Lights sind Lichtquellen, die, von einer gegen unendlich gehenden Entfernung zur Szene, uniform in eine Richtung parallele Lichtstrahlen aussenden. Alle Objekte erhalten unabhängig von ihrer Entfernung und Position dieselbe Strahldichte, wie in etwa durch Sonnenlicht.

- Spot Light
  Spot Lights senden die Lichtstrahlen wie ein Scheinwerfer kegelförmig gerichtet in eine Richtung ab. Die Strahldichte am Objekt kann, je nach Modellierung, von der Entfernung zur Lichtquelle und dem Abstand zur Lichtkegelmitte abhängen.

- Area Light
  Area Lights modellieren die physische Fläche einer Lichtquelle. Sie können z.B. eine kreisförmige oder rechteckige Fläche darstellen, von der aus Lichtstrahlen ausgesendet werden. Beispielhaft könnte man einen Fernseher nennen. Sie sind berechnungstechnisch deutlich aufwendiger, bieten dafür aber auch weiche Schatten (Soft Shadows).

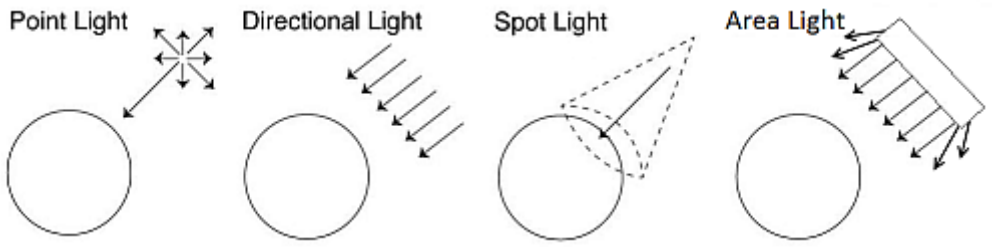

Abbildung 2.4 Verschiedene Arten von Lichtquellen. (Lig, 2016, p. 1)

In der Computergrafik werden Modelle, die das Verhalten des Lichts simulieren, unterschieden in lokale und globale Beleuchtungsmodelle. Im Folgenden werden die beiden Modellarten erläutert.

2.2 Lokale Beleuchtungsmodelle

Die Gruppe der lokalen Modelle treffen eine starke Vereinfachung und simulieren, wie der Name schon andeutet, nur lokale Effekte bezogen auf die beleuchteten Objekte, sprich den Strahlungsaustausch zwischen Lichtquelle und Objekt. Das bedeutet, dass das Verhalten der Lichtstrahlen nach der ersten Interaktion mit einer Oberfläche nicht mehr weiter betrachtet wird. In Abb. 2.5 wird diese Einschränkung verdeutlicht. Die meisten dieser Modelle lassen sich als eine BRDF ausdrücken.

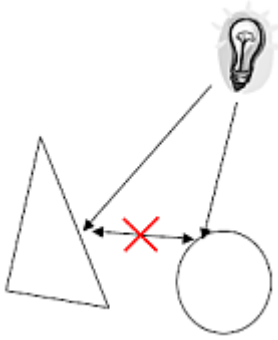

Abbildung 2.5 Lichttransfer lokaler Beleuchtungsmodelle. Zachmann (2014)

Ein in der Computergrafik sehr verbreitetes lokales Beleuchtungsmodell ist das Phong Beleuchtungsmodell, welches von (Gebhardt, 2003, p. 6) erläutert wird. Es ist relativ leicht zu berechnen, allerdings ist es auch nicht physikalisch korrekt, denn, zumindest in der ursprünglichen Variante, wird das Energieerhaltungsgesetz nicht eingehalten. Die BRDF, die vom Phong Modell definiert wird, lautet:

$$f_r(L,V) = I_A k_A + I_i n[k_d(L \cdot N) + k_s(R \cdot V)^n] \tag{2.4}$$

Eine Skizze der Komponenten des Modells ist in Abb. 2.6 dargestellt.

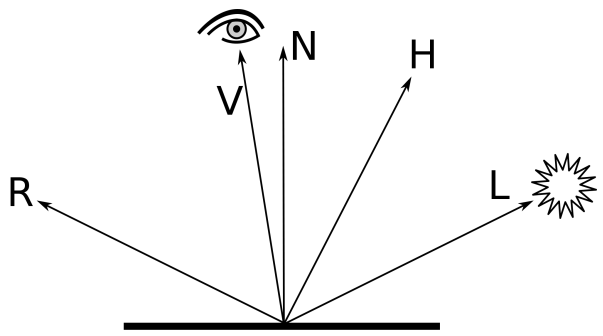

Abbildung 2.6 Skizze der Vektoren der Phong und Blinn-Phong Modelle. van Oosten (2014)

In diesem Modell wird eine Art der globalen Beleuchtung approximiert, die dafür sorgt, dass nicht alle direkt beleuchteten Objekte komplett dunkel sind. Dies wird durch eine Grundbeleuchtung I_A erreicht, die auf alle Objekte angewendet wird. Sie wird multipliziert mit einem entsprechenden ambienten Materialkoeffizienten k_A.
Hinzu addiert werden der ideal diffuse und der ideal spiegelnde Anteil des Lichts. Für ideal diffuse Oberflächen gilt nach dem Lambertschen Kosinusgesetz, dass die ausgehende Strahldichte in alle Richtungen gleich hoch ist, die Betrachtungsrichtung also keine Rolle spielt. Die Strahlungsintensität (Strahlungsleistung pro Einheitswinkel) jedoch ist abhängig von dem Einfallswinkel der Lichtquelle. Daher wird für diesen Anteil der Term $L \cdot N$ verwendet, wobei L der Einfallsvektor des Lichts und N die Oberflächennormale ist.
Der spiegelnde Anteil berechnet sich nach $(R \cdot V)^n$ mit

$$R = 2(L \cdot N)N - L \tag{2.5}$$

wobei R die ideale Reflexionsrichtung, V die Betrachtungsrichtung und n eine materialabhängige Reflexionskonstante ist, die bestimmt, wie rau oder glatt die Oberfläche ist. Der spiegelnde Anteil ist also von Lichteinfalls- sowie Betrachtungswinkel abhängig.
Der diffuse sowie der spiegelnde Anteil wird jeweils mit einem Materialkoeffizienten, k_d, respektive k_s, multipliziert.

Eine oft verwendete Verbesserung ist das Blinn-Phong Modell. Dieses verwendet statt dem Skalarprodukt zwischen idealer Reflexionsrichtung und Betrachtungsrichtung $R \cdot V$ das Skalarprodukt zwischen der Normalen und dem Halbvektor zwischen der Betrachtungs- und Lichteinfallsrichtung $N \cdot H$ mit

$$H = \frac{L + V}{||L + V||} \tag{2.6}$$

Auch hierzu siehe Abb. 2.6.

Dies führt einerseits zu einer geringen Verbesserung der Geschwindigkeit, da die Berechnung des Halbvektors günstiger ist als die des Reflexionsvektors, und andererseits zu einem optisch leicht ansprechenderen Bild. Somit ergibt sich die Formel

$$f_r(L,V) = I_A k_A I_i n[k_d(L \cdot N) + k_s(N \cdot H)^n] \tag{2.7}$$

2.3 Globale Beleuchtung

Globale Beleuchtungsmodelle simulieren das Verhalten des Lichts global, berücksichtigen also, im Gegensatz zu den lokalen Beleuchtungsmodellen, auch die Interaktion zwischen verschiedenen Objekten. Daraus folgt, dass zum einen die Beleuchtung realistischer wirkt und zum anderen, dass auch bestimmte Lichteffekte dargestellt werden können, die vorher nicht modelliert werden konnten. Solche Effekte sind z.B. indirekte Beleuchtung (Color Bleeding/indirect lighting), indirekte Schattierung (indirect shadows), fortgeschrittene, spiegelnde Reflexionen (specular/glossy reflections) und Kaustiken (caustics). In Abb. 2.7 werden einige dieser Effekte veranschaulicht.

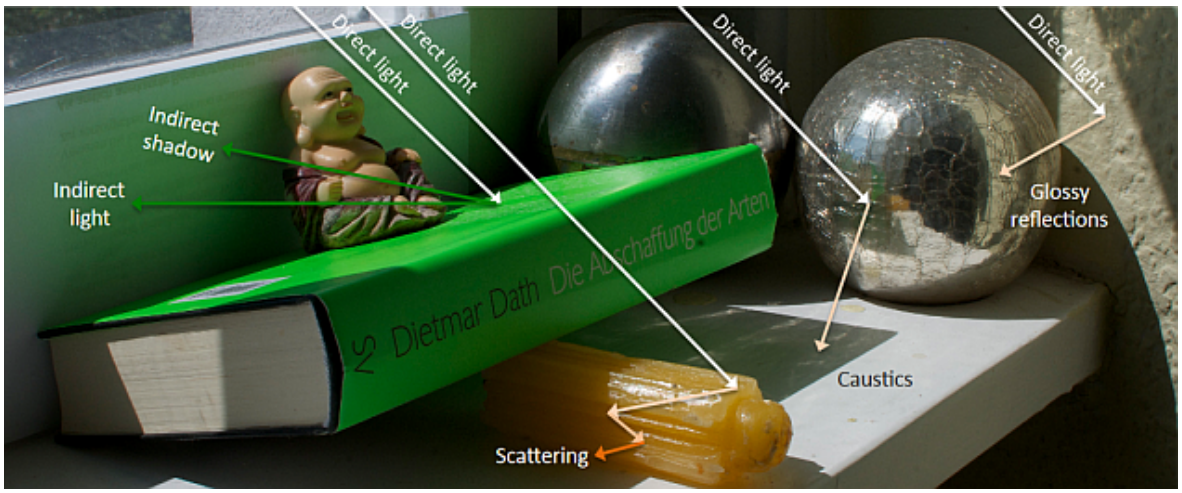

Abbildung 2.7 Global Illumination-Effekte. (Ritschel et al., 2012, p. 1)

Bei indirekter Beleuchtung handelt es sich um Lichtstrahlen, die von einem Objekt reflektiert werden und anschließend auf ein anderes auftreffen und es so beleuchten. Indirekte Verschattung ist folglich die Verdeckung von indirekter Beleuchtung. Kaustiken sind Lichteffekte, die durch eine Fokussierung bzw. Bündelung von Licht entstehen. Dies passiert meist durch Refraktion durch transparente Objekte oder Spiegelung von stark spiegelnden Objekten.

Die mathematische Basis für alle Algorithmen zur Berechnung von globaler Beleuchtung ist die 1986 von David Immel et al. und Jim Kajiya veröffentlichte Rendergleichung Kajiya (1986). Sie beschreibt das Verhalten von Licht unter Berücksichtigung von Lichtquelle,

BRDF und Sichtbarkeit und erfüllt den Energieerhaltungssatz. Die Rendergleichung ist, wie auch (Ritschel et al., 2012, pp. 2-3) darlegt, eine Integralgleichung, die beschreibt, wie viel Licht an einem Punkt (hier x) in eine Richtung (ω_o) abgestrahlt wird, sprich die ausgehende Strahldichte.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + L_r(x, \omega_o) \tag{2.8}$$

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) f_r(x, \omega_i, \omega_o)(\omega_i \cdot \mathbf{n}) d\omega_i \tag{2.9}$$

Dabei ist $L_e(x, \omega_o)$ die vom Punkt aus emittierte und $L_r(x, \omega_o)$ die reflektierte Strahldichte. Letztere berechnet sich aus dem Integral über die Hemisphäre von der aus Richtung ω_i eingehenden Strahldichte $L_i(x, \omega_i)$, der BRDF $f_r(x, \omega_i, \omega_o)$ und einem vom Lichteinfallswinkel ω_i abhängigen Faktor $(\omega_i \cdot \mathbf{n})$, der die Bestrahlungsstärke senkt.(Hughes et al., 2013, p. 374)

Die Rendergleichung zu lösen ist nicht möglich, da es eine Rekursion im Integral besitzt. Es gibt aber viele approximative Verfahren, siehe Abschnitt 2.5, die eine Lösung annähern.

Im folgenden Abschnitt sollen vorerst einige Beschleunigungstechniken vorgestellt werden.

2.4 Beschleunigungstechniken

Um den hohen Rechenaufwand in der Computergrafik allgemein und speziell bei der Berechnung von globaler Beleuchtung zu senken, werden verschiedenste Techniken eingesetzt. Einige dieser Techniken, die in dieser und vorherigen Arbeiten eingesetzt wurden, werden in diesem Abschnitt erläutert. Dazu gehören Repräsentationsformen für Daten, räumliche Datenstrukturen, die den Raum aufteilen und Objekte in Bereiche einsortieren, und Techniken, die durch Vernachlässigung von situationsabhängig unwichtigen Daten Rechenaufwand sparen.

2.4.1 Beschleunigungsdatenstrukturen

Beschleunigungsdatenstrukturen werden benutzt, um, im allgemeinsten Fall, Algorithmen durch Vereinfachen oder Einsparen von Berechnungen zu beschleunigen. In den meisten Fällen werden sie zur räumlichen Einteilung von Daten im Zuge von Raycasting eingesetzt.
Als Raycasting bezeichnet man das Aussenden eines Strahls von einem Punkt aus in eine Szene, bei dem der Schnittpunkt mit Objekten von Interesse ist. Häufige Anwendungsdomänen von Raycasting sind z.B. Kollisions- und Verdeckungsberechnung.
Das Konzept hinter den Beschleunigungsdatenstrukturen ist es, räumliche Bereiche und ihre enthaltenen Objekte zusammenzufassen und mit einem einzigen Schnitttest ggf. alle gemeinsam verwerfen zu können.

2.4.1.1 Uniform Grids

Ein Uniform Grid ist nach (Ahmed, 2009, p. 1) eine Datenstruktur, bei der eine zweidimensionale Fläche oder ein dreidimensionaler Raum gleichmäßig in Zellen aufgeteilt wird, siehe Abb. 2.8. Jede Zelle enthält in der Regel eine Liste aller Objekte, die sich völlig oder teilweise innerhalb dieser Zelle befinden. Bei einem Schnittest lassen sich so Berechnungen sparen, indem erst der Schnitt mit der Zelle und nur bei positivem Resultat mit den enthaltenen Objekten geprüft wird. Die Stärke dieser Datenstruktur sind Situationen, bei denen die Objekte homogen verteilt und gleichmäßig in ihrer Größe sind, da so eine sinnvolle Zellengröße gewählt werden kann und eine optimale Verteilung auf diese vorliegt, wie auch (Hapala et al., 2011, p. 1) anmerkt.

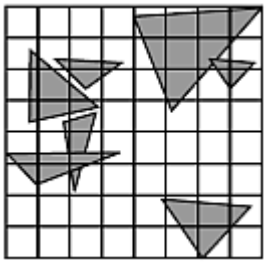

Abbildung 2.8 Einige Objekte, überlagert von einem Uniform Grid. Ertürk (2008)

2.4.1.2 Octrees

Wie z.B. (Samet and Webber, 1988, pp. 51-53) erläutern, sind Octrees Baumstrukturen, bei denen jeder Knoten, der kein Blatt ist, genau acht Kindknoten enthält. Mit ihnen lässt sich ein Raum rekursiv von der Wurzel aus immer feingranularer partitionieren, siehe Abb. 2.9. Dabei wird das Volumen, welches von einem Knoten abgedeckt wird, in der nächsten Stufe mittig entlang der Achsen aufgeteilt und jeder Oktant von einem Kindknoten repräsentiert. Ihr großer Vorteil ist der hierarchische Aufbau, so dass je nach Objektverteilung bestimmte räumliche Bereiche fein aufgelöst werden, während ausgedehnte, leere Bereiche durch wenige große Knoten abgebildet werden können. Durch die komplexere Konstruktion und Traversierung ist allerdings ein höherer Verwaltungsaufwand nötig als bei den Uniform Grids, was auch (Hapala et al., 2011, p. 1) anhand von ähnlichen hierarchischen Strukturen feststellt.

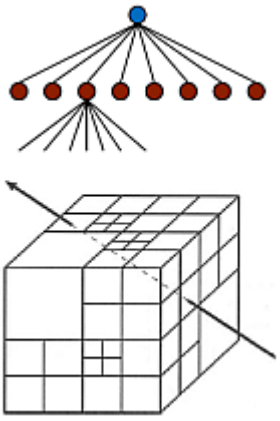

Abbildung 2.9 Oben: Die Baumstruktur eines Octrees. Unten: Ein mittels Octree partitionierter Raum. Zachmann (2015)

2.4.2 Dreidimensionale Voxel-Repräsentation

Ein Voxel repräsentiert einen diskreten Datenpunkt im dreidimensionalen Raum, wobei er ein würfelförmiges oder allgemein quaderförmiges Volumen abdeckt. Ausführlich erklärt wird diese Voxel-Repräsentation von (Kaufman et al., 1993, pp. 54-58). Eine Voxel-Datenstruktur kann sich im einfachsten Fall als ein dreidimensionales Uniform Grid betrachten lassen. Der Raum wird, über alle Achsen gleich verteilt, in einzelne Voxel gegliedert. Hierbei wird die Position der einzelnen Voxel nicht explizit gespeichert, sondern relativ zu der Datenstruktur an sich angegeben.
Voxel-Datenstrukturen werden vor allem bei der Repräsentation und Visualisierung von Volumendaten verwendet, z.B. in der Medizin oder zur Darstellung von Terrain.
Je nach Anwendungsfall können die Voxel verschiedene Daten speichern. Es kann nur ein binärer Wert gespeichert werden, z.B. um zu kodieren, ob dieses Volumen gefüllt ist oder nicht, oder auch mehrere Datensätze, wie z.B. Farbwerte. Darüber hinaus unterscheidet man, ob nur die Hülle von Objekten durch die Voxel repräsentiert werden sollen oder der komplette Körper.
Ein Vorteil der Datenrepräsentation durch Voxel, neben der offensichtlich einfachen volumetrischen Darstellung und der dynamischen Deformierbarkeit der Strukturen, ist der leicht und günstig zu berechnende Schnitttest. Der größte Nachteil ist allerdings der hohe Speicherverbrauch, denn für eine feingranulare Darstellung sind sehr viele Voxel nötig, die alle mehr oder weniger viele Daten halten müssen.

2.4.3 Level of Detail (LOD)

Level of Detail, u.a. beschrieben von (Goswami, 2012, p. 5), ist ein Begriff für Verfahren, die, in Abhängigkeit von der Betrachtungsentfernung, wenig relevante Daten vernachlässigen

oder zusammenfassen und darzustellende Objekte somit vereinfachen oder komplett verwerfen. Durch diese Reduzierung des Detailgrads kann erheblich Rechenzeit gespart werden auf Kosten von Daten, die ohnehin nicht oder kaum zum letztendlichen Bildeindruck beitragen. Um diese Abwägung der Relevanz zu treffen, wird meist die Entfernung zum Betrachter als entscheidende Metrik herangezogen und verschiedene Stufen definiert, welche mit fortschreitender Entfernung jeweils den Detailgrad senken.

2.4.3.1 Mip Mapping

Bei Mip Mapping handelt es sich nach (Tanner et al., 1998, p. 2) um ein Verfahren, welches benutzt wird, um Aliasing-Artefakte zu verhindern und das Konzept des LODs auf Texturen zu übertragen. Für eine Textur in hoher Originalauflösung wird eine Reihe kleinerer Texturen erstellt, welche jeweils die Daten der Vorgängertextur mit halber Kantenlänge auflösen, insgesamt also ein Viertel der Größe besitzen. Jede dieser Texturen bildet dann eine LOD-Stufe und alle zusammen eine Mip-Map, oder auch Bildpyramide genannt. In Abb. 2.10 ist eine solche illustriert.
Dies Verfahren kostet zwar extra Speicherplatz für die zusätzlichen Texturen, spart aber Rechenleistung beim Auslesen von Texturdaten, da in den gröber aufgelösten Texturen ein Texel mehr Raum abdeckt und somit weniger Leseoperationen ausgeführt werden müssen und diese zudem im Speicher näher beieinander liegen, was die Cache-Misses reduziert.

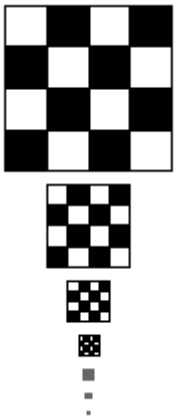

Abbildung 2.10 Eine MIP-Map-Pyramide. (Tanner et al., 1998, p. 2)

2.4.3.2 Clipmaps

Wie in (Tanner et al., 1998, pp. 1-4) erläutert, bauen Clipmaps auf MIP Mapping auf und adressieren das Problem, dass eine zu große MIP-Map nicht mehr als Ganzes in den Speicher passt, bzw. unnötig viel belegt. Clipmaps werden häufig im Zuge des Renderns von Terrain verwendet, wo Höhendaten für riesige Bereiche vorliegen, aber als Ganzes und in der Auflösung nicht in eine einzelne Textur passen. Die Kernerweiterung von Clipmaps ist, dass die

feiner aufgelösten Level der virtuellen MIP-Map beschnitten, clipped, werden. Daraus ergibt sich, dass die MIP-Map-Pyramide aufgeteilt wird in eine kleine Clipmap-Pyramide, die die gröberen, nicht beschnittenen Level enthält, und einen Clipmap-Stack, der einen Quader durch die ursprünglichen MIP-Map-Level bildet. Zur Veranschaulichung siehe Abb. 2.11.

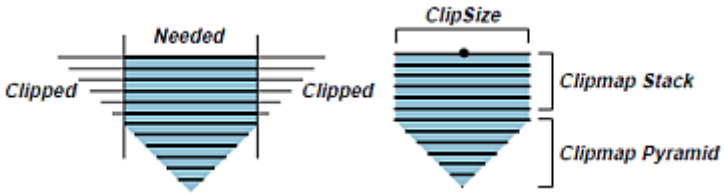

Abbildung 2.11 Eine Clipmap mit Pyramide und Stack auf Basis einer MIP-Map. (Tanner et al., 1998, p. 2)

Jedes Level des Clipmap-Stacks kann sich, unabhängig von den anderen Leveln, frei innerhalb des ursprünglichen MIP-Map-Levels bewegen, je nach Position des Betrachters, um dessen sie zentriert sind. Siehe dazu Abb. 2.12. Somit puffert sie einen relevanten, aktiven Teil des gesamten Levels.

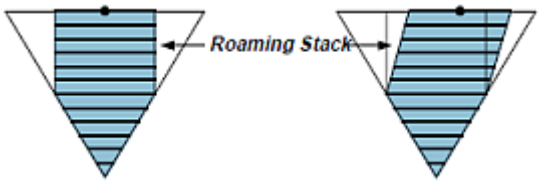

Abbildung 2.12 Bewegung der aktiven Regionen. (Tanner et al., 1998, p. 4)

Die jeweiligen Level des Clipmap-Stacks besitzen dieselbe Auflösung, repräsentieren aber je gröber das Level ist einen größeren Bereich der Szene, siehe Abb. 2.13

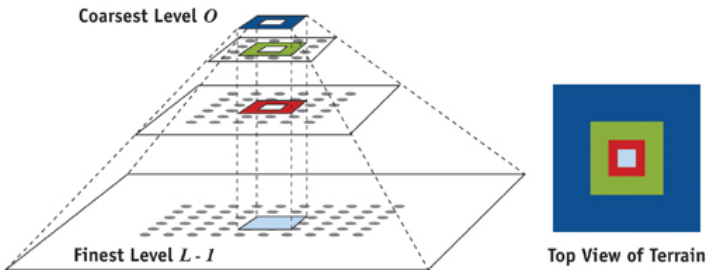

Abbildung 2.13 Die gröberen Clipmap-Level repräsentieren einen zunehmend größeren Bereich der Szene. Asirvatham and Hoppe (2005)

Um Clipmaps effizient zu aktualisieren, wird ein Verfahren benutzt, welches Toroidal Addressing genannt wird. Hierbei nutzt man die Beobachtung, dass, wenn sich die aktive Clip-Region verschiebt, sich ein quadratischer, mittiger Teil der Daten nicht verändert, einige Randbereiche aus der aktiven Region fallen und dafür gegenüber gleich große Bereiche herein

rutschen. Die somit frei werdenden Speicheradressen der nicht mehr relevanten Bereiche werden zum Speichern der neu benötigten gegenüberliegenden Bereiche genutzt. Somit können die Clipmap-Level in-place gespeichert werden. In Abb. 2.14 ist dieses Verfahren abgebildet.

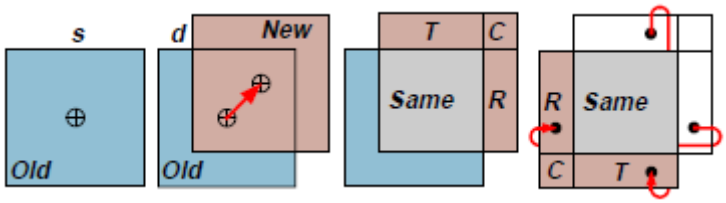

Abbildung 2.14 Update einer Clipmap mit Toroidal Adressing. (Tanner et al., 1998, p. 4)

2.5 Typische Ansätze für globale Beleuchtung

Im Folgenden werden, wie in Abschnitt 2.3 bereits angekündigt, einige typische Ansätze betrachtet, die aktuell verwendet werden, um globale Beleuchtung zu berechnen bzw. zu approximieren. Im Zuge dessen wird auch auf die jeweiligen Vor- und Nachteile eingegangen. Manche dieser Verfahren stützen sich dabei auf einige der im vorigen Abschnitt 2.4 beschriebenen Beschleunigungstechniken, jedoch sind viele nichts desto trotz nicht dynamisch echtzeitfähig.

2.5.1 Raytracing

2.5.1.1 Grundprinzip

Zur Berechnung der globalen Beleuchtung werden, wie (Whitted, 1980, pp. 1-4) beschreibt, aus Sicht der Kamera viele Strahlen (in der Regel einer durch jeden Pixel) in die Szene geschossen und jeweils der Schnittpunkt mit der Geometrie berechnet. Von diesem aus werden wieder Strahlen in die Szene ausgesandt, um zu ermitteln, von welchen Lichtquellen der Schnittpunkt direkt beleuchtet wird und um von der Oberfläche reflektiertes und gebrochenes Licht zu erhalten. Dieses Prinzip wird rekursiv fortgesetzt, indem von den Auftreffpunkten der reflektierten und gebrochenen Strahlen wiederum weitere ausgesandt werden, siehe Abb. 2.15. So bildet sich mit der Zeit ein sogenannter Strahlenbaum.

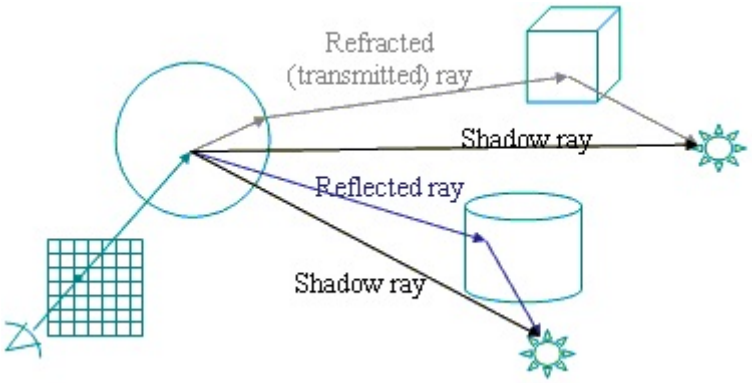

Abbildung 2.15 Ein Strahl wird von der Kamera aus in die Szene geschossen und trifft auf eine Kugel. Von dem Schnittpunkt aus werden Schatten-Strahlen zu den Lichtquellen geschossen sowie je ein Strahl gebrochen und reflektiert. Die reflektierten Strahlen treffen nach weiteren Reflexionen an anderen Objekten schließlich die Lichtquellen. Vrajitoru (2011)

Um globale Effekte wie weiche Schatten, Kaustiken und Tiefenunschärfe darzustellen, werden nach (Ritschel et al., 2012, p. 5) und (Kajiya, 1986, p. 147) jeweils mehrere Strahlen jeder Art pro Pixel und Schnittpunkt ausgesandt und die Ergebnisse gemittelt. In diesem Fall spricht man von Path Tracing. Durch diese Monte Carlo-Integration kann die Rendergleichung angenähert werden.

2.5.1.2 Varianten

Es gibt einige Varianten, die das Grundprinzip erweitern, wie z.B. Metropolis Light Transport (MLT) und Photon Mapping.
Laut (Veach and Guibas, 1997, p. 1) werden bei **MLT** Strahlen von der Lichtquelle sowie der Kamera ausgesandt und zu Pfaden kombiniert. Pfade, die einen großen Beitrag zur Beleuchtung der Szene leisten, werden durch kleine Veränderungen zu neuen Pfaden mutiert.
Bei **Photon Mapping** werden, wie (Ritschel et al., 2012, pp. 5-6) beschreibt, in einem ersten Schritt Photonen mittels Raytracing, und ausgehend von der Lichtquelle, durch die Szene geschossen. Dadurch wird eine Photon-Map der Auftreffpunkte erstellt. In einem zweiten Schritt wird für jeden Pixel die Beleuchtung entweder durch die Dichte der umliegenden Photonen oder durch Final Gathering ermittelt. Letzteres ermittelt pro Pixel die Beleuchtung, ausgehend von der sichtbaren Szene. Die Abbildung 2.16 verdeutlicht das Verfahren.

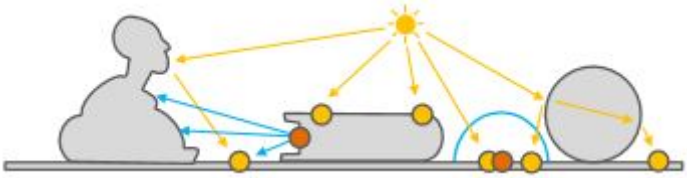

Abbildung 2.16 Photonen werden von der Lichtquelle aus in die Szene entsandt und nach Reflexion gespeichert (gelbe Punkte). Indirekte Beleuchtung wird mittels Final Gathering (blaue Pfeile) oder einer Abschätzung der Photonendichte (blauer Kreis) berechnet. (Hier an den orangen Punkten) (Ritschel et al., 2012, pp. 6)

Für auf Raytracing basierenden Verfahren gilt, dass sie sehr präzise und arm an Artefakten globale Beleuchtung darstellen können, aber gleichzeitig dabei auch sehr langsam sind. (Křivánek et al., 2010, p. 1)

2.5.2 Radiosity

Die Grundidee hinter Radiosity ist nach (Ritschel et al., 2012, pp. 4-5), die Szenengeometrie in sogenannte Patches zu diskretisieren und eine Verbindungsstruktur zwischen den Patches zu erzeugen, die angibt, welche Patches sich gegenseitig beleuchten. Schließlich wird der Lichttransport selbst berechnet, vgl. Abb. 2.17.

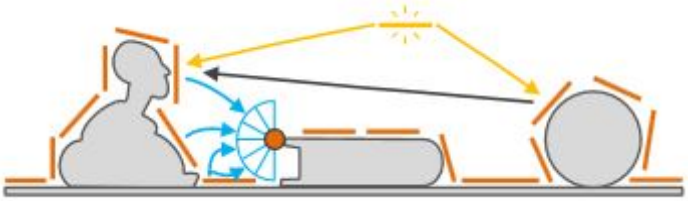

Abbildung 2.17 Die Geometrie wird in Patches (orange) diskretisiert und die Beleuchtung untereinander berechnet (Pfeile) (Ritschel et al., 2012, p. 5)

Die meisten Implementierungen beschränken sich zwecks Geschwindigkeitsgewinn allerdings auf diffuse Oberflächen, wie u.a. (Křivánek et al., 2010, p. 1) erwähnt. Ein weiteres Problem ist außerdem dynamische, deformierbare Geometrie, da hierfür zusätzlich zu den Patches auch die Verbindungsstruktur neu berechnet werden muss.

2.5.3 Screen Space-Techniken

Wie (Ritschel et al., 2012, pp. 5,11) schreibt, ist die Nutzung von Screen Space-Techniken zur annäherungsweisen Berechnung von Umgebungsverdeckung sehr verbreitet. Die einfachste Variante ist Screen Space Ambient Occlusion (SSAO), bei der für jeden Pixel ein durchschnittlicher Verdeckungswert aus der Z-Buffer-Position seiner Nachbarpixel berechnet wird.

Komplexere Varianten, wie Screen Space Directional Oclusion (SSDO) und Horizon Based Ambient Occlusion (HBAO), erreichen bessere Ergebnisse.
Prinzip bedingt sind Sceen Space-Techniken sehr schnell zu berechnen und unabhängig von der Komplexität der Geometrie, allerdings auch mangels Informationen vergleichsweise unpräzise und eingeschränkt, was die darstellbaren Effekte angeht. (Mavridis et al., 2010, p. 2) bestätigt dies.

2.5.4 Instant Radiosity (Virtual Point Lights)

Instant Radiosity funktioniert ähnlich wie Photon Mapping. (Ritschel et al., 2012, pp. 6-7) beschreibt es wie folgt. In einem ersten Schritt wird die Photon-Map erstellt. Im zweiten Schritt wird hier allerdings jedes Photon als Virtual Point Light (VPL) betrachtet, welches Licht in die Szene abgibt. Die Pixel nehmen dieses Licht ggf. auf. Zur Veranschaulichung siehe Abb. 2.18.

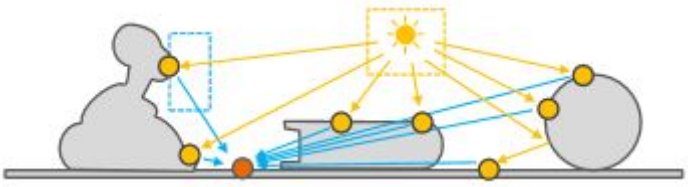

Abbildung 2.18 Jedes Photon agiert nach Reflexion durch die Szene (gelbe Punkte) als Virtual Point Light und beleuchtet (blaue Pfeile) die Szene (hier beispielhaft den orangen Punkt) (Ritschel et al., 2012, p. 7)

Mit zunehmender Anzahl der VPLs wird dieses Verfahren durch die jeweilige teure Schattenberechnung langsam.

2.5.5 Voxel-Based Cone Tracing

Wie von Crassin et al. (2011), Panteleev (2014) und McLaren (2015) beschrieben, wird bei Verfahren dieser Art eine auf Voxeln basierende Datenstruktur genutzt, um Geometrie- sowie Beleuchtungsinformationen der Szene räumlich abzuspeichern. Durch Anwendung von approximativen Cone Tracing können die Daten relativ schnell ausgelesen und Abschätzungen über Sichtbarkeit und Energietransfer gemacht werden. Die Cone Tracing-Phase fungiert dabei als Final Gathering und ermittelt die indirekte Beleuchtung. Dabei setzt sich die indirekte Beleuchtung aus der eingehenden Strahlungsleistung über die komplette Hemisphäre zusammen. Die Cones partitionieren die Hemisphäre und durch eine gewichtete Summierung ihrer Resultate wird eine Approximation der indirekten Beleuchtung berechnet. Die Zusammenfassung von vielen Strahlen zu weniger Cones erhöht die Performance deutlich und ist, bezogen auf die Präzision der Ergebnisse, eine vertretbare Approximation, da die einzelnen Strahlen räumlich sowie direktional kohärent sind. Die Voxel-Datenstruktur ist meist hierarchisch aufgebaut und liegt auf dem GPU-Speicher. Üblicherweise wird somit der komplette

Algorithmus auf der Grafikkarte ausgeführt, was der Geschwindigkeit zugute kommt.
In einem ersten Schritt wird die polygonale Geometrie mittels des Rasterisierers in die Voxel-Datenstruktur überführt, danach Informationen über die direkte Beleuchtung der Szene in den Voxeln gespeichert. In einem letzten Schritt wird die Szene aus Sicht der Kamera gerendert, aber zudem für jeden Pixel durch approximatives Cone Tracing durch die Voxel-Datenstruktur die indirekte Beleuchtung berechnet. Die endgültige Beleuchtung besteht schließlich aus dem mittels Rasterisierer gerenderten direkten Beleuchtungsanteil sowie dem über das approximative Cone Tracing erhaltenen indirekten Beleuchtungsanteil.
Voxel Cone Tracing-Vefahren sind in der Lage, für statische sowie dynamische Szenen glaubhafte Annäherungen von globaler Beleuchtung in Echtzeit zu berechnen. Dies sowohl für diffuse als auch spiegelnde Reflexionen. Ein Nachteil dieser Verfahren ist der mit der Granularität der Voxel-Datenstruktur stark steigende Speicherbedarf, der bereits in Abschnitt 2.4.2 erwähnt wurde. Dieser steigt kubisch mit der Auflösung der Voxel-Datenstruktur.

Kapitel 3

Vorherige Arbeit

In diesem Kapitel werden drei Implementierungen von Voxel Cone Tracing zur angenäherten Echtzeit-Berechnung von globaler Beleuchtung vorgestellt, die als Basis bzw. Inspiration für die in Kapitel 4 dargelegte eigene Implementierung dienen. Das Grundprinzip, erläutert in Abschnitt 2.5.5, haben alle gemein.
Ich habe mich für Voxel Cone Tracing und gegen die anderen in Abschnitt 2.5 diskutierten Verfahren zur Darstellung der globalen Beleuchtung entschieden, weil es nach Abwägung der genannten Vor- und Nachteile am vielversprechendsten erschien. Speziell die voraussichtlich vergleichsweise hohe Geschwindigkeit bei voll-dynamischen Szenen war ausschlaggebend.

3.1 Voxel Cone Tracing von Crassin et al.

In der von Crassin et al. (2011) vorgestellten Variante, welche als Vorreiter für diese Art von Algorithmen gilt, wird ein Sparse-Voxel-Octree (SVO) als Datenstruktur verwendet und dynamisch aktualisiert. Eine Erläuterung von Octrees im Allgemeinen ist in Abschnitt 2.4.1.2 nachzulesen. Der größte Unterschied des SVO zu einem regulären Octree ist, dass die Knoten dynamisch zur Laufzeit mittels der GPU erzeugt werden und nicht anfangs für alle möglichen Knoten der Speicher reserviert wird. Ein Knoten repräsentiert genau einen Voxel und alle Knoten werden linear im GPU-Speicher gespeichert. Zudem werden je 2x2x2 Knoten in Kacheln gruppiert, so dass ein Pointer pro Knoten ausreicht, um alle seine Kindknoten zu referenzieren. Das Befüllen mit den Geometriedaten wird laut dem Paper mittels des Rasterisierers der Grafikkarte in drei Schritten, jeweils einmal pro Hauptachse der Szene, durchgeführt. In einer späteren Präsentation Crassin (2012) wird eine verbesserte Variante des Voxelisierens vorgestellt, die mit nur einem Schritt auskommt. Dabei wird für jedes Dreieck die Hauptachse der Normalen ermittelt und das Dreieck auf die Ebene in World Space projiziert, die möglichst orthogonal zur Hauptachse liegt. Um die Abdeckungsrate weiter zu erhöhen, wird Conservative Rasterization angewandt. Dazu werden die Kanten des Dreiecks nach außen verschoben und anschließend an den Fragment-Kanten beschnitten. So wird jedes

Fragment gerendert, was von dem Dreieck geschnitten wird, unabhängig vom Sampling-Punkt innerhalb des Fragments.

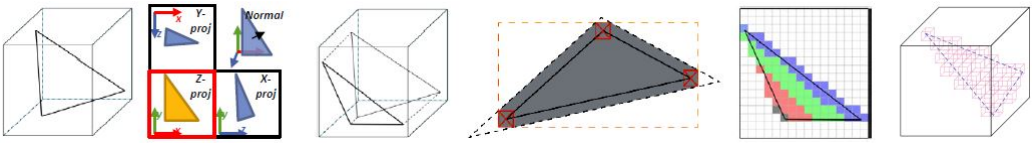

Abbildung 3.1 Voxelization mittels Wahl der Hauptachse, Projektion auf die Ebene mit größter Fläche und Conservative Rasterization. Crassin (2012)

Nach dem Einfügen der Geometriedaten wird nach dem ursprünglichen Paper einmal aus Sicht jeder Lichtquelle gerendert und die Radianz sowie Richtung der Lichtstrahlen abgespeichert. Daten niedriger Level im SVO werden in einem weiteren Schritt iterativ in die höheren Level propagiert, indem die Daten der Voxel der tieferen Ebene gemittelt werden. Dabei werden die Daten pro Seite eines Voxels gemittelt, gespeichert und später wieder ausgelesen. Das bietet den Vorteil, dass, falls mehrere Seiten eines Objektes, z.B. Vorder- und Rückseite, in denselben Voxel fallen, ihre Farbwerte nicht zu einer einzigen Farbe zusammengemischt werden, was potentiell schlechte Resultate liefern würde. Durch die anisotropische Speicherung der Daten kann ein Voxel verschiedene Farben in die einzelnen Richtungen reflektieren.
Schließlich wird mittels approximativem Cone Tracing die indirekte Beleuchtung pro Pixel ermittelt. Dazu werden mehrere Cones über die Hemisphäre der Geometrie ausgesandt, vgl. Abb. 3.2, und jeweils Samples entlang der Cones ausgelesen. Für den diffusen Anteil des indirekten Lichts werden mehrere Cones mit größerem Öffnungswinkel ausgesandt, während für den glänzenden Anteil nur ein schmalerer Cone in die Reflexionsrichtung ausgesandt wird.

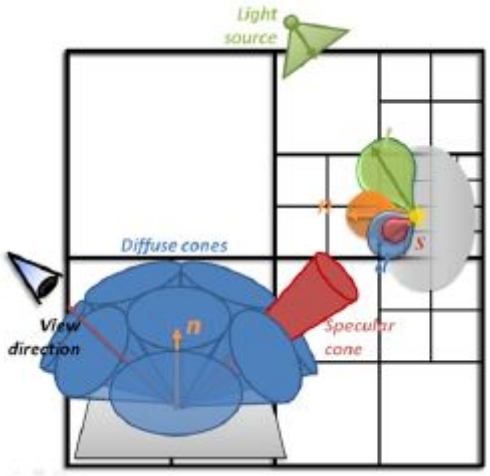

Abbildung 3.2 Cones für den diffusen sowie glänzenden Anteil des Lichtes werden von der Pixel-Position in World Space durch den SVO geschossen.(Crassin et al., 2011, p. 3)

Dabei entspricht der Cone-Radius an der Sample-Position immer dem Level der Datenstruktur und die Schrittweite ist unabhängig von der Größe der Voxel. Quadrilineare Interpolation mindert Aliasing-Artefakte und sorgt für weiche Übergänge. Abbildung 3.3 verdeutlicht den Samplingprozess. Die Farbe und der Verdeckungsgrad der Samples werden nach dem klassischen Volumen-Raymarching-Verfahren entlang der Cones akkumuliert.

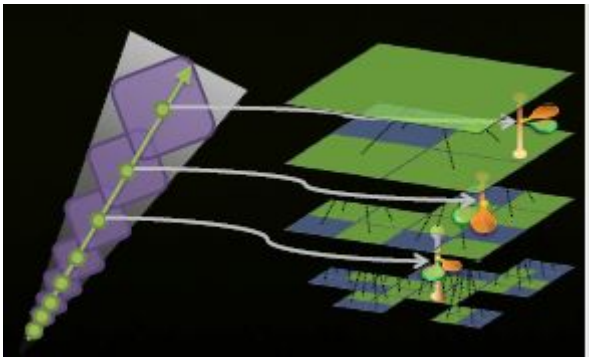

Abbildung 3.3 Die Samples eines Cones werden abhängig vom Cone-Radius aus den Leveln des SVO geladen und interpoliert. Crassin (2012)

3.2 Voxel Cone Tracing von NVIDIA

Die von NVIDIA entwickelte Voxel Cone Tracing-Variante (VXGI) nutzt nach Panteleev (2014) als Voxel-Datenstruktur statt dem SVO eine Clipmap mit drei bis fünf LODs. Diese Datenstruktur wird dynamisch aktualisiert. Das Einfügen der Geometrie in die Clipmap geschieht hier wie bei Crassin et al. mit dem Rasterisierer der Grafikkarte, hier auch in nur einem Schritt. Dazu wird jedes Dreieck auf die Ebene projiziert, auf der es die größte Fläche einnimmt, und auf diese Ebene mit Multisampling Antialiasing (MSAA) gerendert. Anschließend werden die Subsamples auf die anderen beiden Seiten zurück projiziert. Veranschaulicht wird dies in Abb. 3.4.

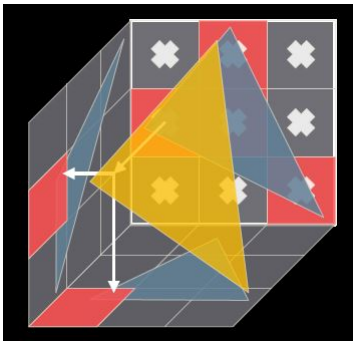

Abbildung 3.4 Ein Dreieck wird auf die Seite des Voxels gerendert, auf der es die größte Fläche einnimmt (hier die hintere) und anschließend die überdeckten Fragmente auf die anderen Seiten projiziert.Panteleev (2014)

Die Beleuchtungsdaten werden mittels Reflective Shadow Maps (RSM) berechnet und in die einzelnen Seiten der Voxel abgespeichert. Beim RSM wird wie bei regulärem Shadow Mapping aus Sicht der Lichtquelle gerendert, allerdings hier Position, Normale und Farbe für alle sichtbaren Texel abgespeichert, vgl. Abb. 3.5.

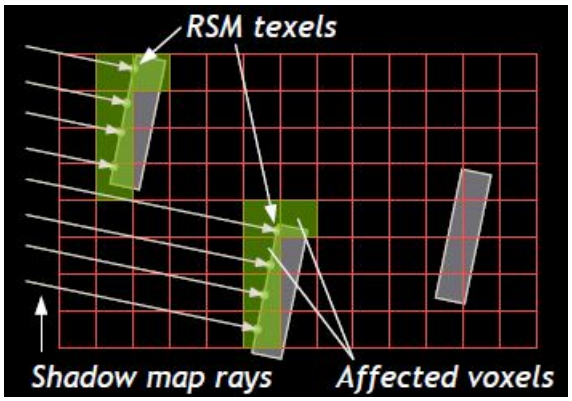

Abbildung 3.5 Einfügen der Beleuchtungsdaten in die Voxel mittels RSM.Panteleev (2014)

Die berechneten Daten werden anschließend in die höheren LODs propagiert. Im Cone Tracing-Schritt wird für jedes Sample iterativ ein gröberes LOD gewählt. Zudem wird nicht für jeden Pixel die indirekte Beleuchtung berechnet, sondern in einer geringeren Auflösung gerendert und anschließend interpoliert, da die Lichtverteilung im Screen Space nicht so hochfrequent ist, dass eine etwas gröbere Abtastung ins Gewicht fallen würde. Trotz der zusätzlichen Interpolation spart man so deutlich an Berechnungszeit ein.

3.3 Voxel Cone Tracing von Q-Games

Die von Q-Games im Spiel „The Tomorrow Children“ verwendete Implementierung nutzt laut McLaren (2015) als Datenstruktur eine sogenannte Voxel Texture Cascade. Diese besteht aus mehreren identisch aufgelösten und sich überlappenden Texturen, bei denen sich die räumliche Ausdehnung von Level zu Level verdoppelt, siehe Abb. 3.6.

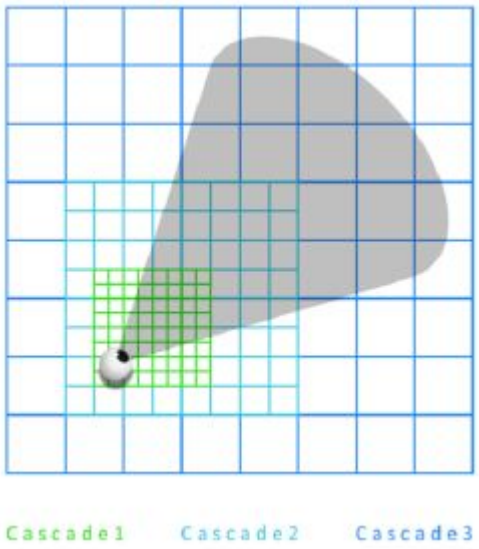

Abbildung 3.6 Voxel Texture Cascades, bestehend aus 3 überlappenden Texturen.McLaren (2015)

Damit gleicht sie vom Prinzip her einem Clipmap-Stack. Auch hier wird die Datenstruktur wie bei den vorher vorgestellten Implementierungen dynamisch aktualisiert und die Daten wieder pro Face eines Voxels gespeichert. Für das Einfügen der Geometriedaten in die Voxel wird wie üblich der Rasterisierer der Grafikkarte verwendet, hier allerdings mit Supersampling. Zusätzlich wird mittels Compute-Shader jeweils ermittelt, welche Fragmente das vorderste und das hinterste des Voxels sind und nur diese gespeichert. Anschließend wird die direkte Beleuchtung in die Datenstruktur gespeichert, was teilweise mittels Cone Tracing geschieht. Ein großer Unterschied zu den beiden zuvor präsentierten Varianten von Voxel Cone Tracing besteht hier darin, dass nicht nur die erste indirekte Reflexion, sondern zwei bis drei berechnet werden, was vor allem zusätzlichen Speicher kostet, aber auch bessere Ergebnisse liefert. Die indirekte sowie die finale Beleuchtung im Screen Space werden beide ebenfalls mittels Cone Tracing ermittelt, letzteres dabei wie bei NVIDIA in einer niedrigeren Auflösung.

Kapitel 4

Algorithmen und Implementierung

In diesem Kapitel wird von mir die eigene Implementierung eines Voxel Cone Tracing Verfahrens detailliert vorgestellt, welche auf den in Kapitel 3 beschriebenen bestehenden Arbeiten basiert. Zuerst wird ein grober Überblick gegeben, darauf folgend die verwendeten Algorithmen und Datenstrukturen veranschaulicht sowie begründet, warum diese gewählt wurden. Schließlich endet dieses Kapitel mit konkreten Implementierungsdetails und einer Komplexitätsanalyse.

4.1 Überblick

Die hier vorgestellte, selbstgeschriebene Implementierung und ihre zugrunde liegenden Algorithmen folgen dem für Voxel Cone Tracing üblichen Schema, welches in 2.5.5 beschrieben ist. Es wird eine mehrstufige Voxel-basierte Datenstruktur verwendet, in die, unter Zuhilfenahme des Rasterisierers der Grafikkarte, die Geometrie- und Beleuchtungsdaten der Szene eingefügt werden. Auf diese Datenstruktur wird anschließend approximatives Cone Tracing angewendet, um die indirekte Beleuchtung zu berechnen. Schließlich werden direkte und indirekte Beleuchtung kombiniert, wobei die direkte Beleuchtung mittels Forward-Rendering und Blinn-Phong-Beleuchtungsmodell berechnet wird. In Algorithmus 1 wird der grobe Ablauf abstrahiert dargestellt, er gleicht auf dieser Ebene denen in Kapitel 3.

Algorithmus 1 Grober Ablauf der Voxel Cone Tracing Implementierung

function VoxelizePolygonalGeometry
store material data in corresponding voxel of voxel-datastructure
function InjectLighting
calculate and store irradiance in corresponding voxel of voxel-datastructure
function PropagadeData
propagade averaged data across LODs
function ConeTracing
trace cones and calculate indirect lighting
function FinalLighting
calculate direct lighting and combine with indirect lighting

In dieser Implementierung wird, im Gegensatz zu den im vorigen Kapitel 3 vorgestellten, jeder Schritt des Algorithmus in jedem Frame ausgeführt, ohne Daten aus vorigen Frames zu übernehmen. Insbesondere wird auch die Voxel-Datenstruktur immer komplett neu befüllt und nicht dynamisch aktualisiert. Dem zugrunde liegt die Annahme, dass es für sehr dynamische Szenen sehr aufwendig und damit auch teuer wäre, zu ermitteln, welche Bereiche vom letzten Frame übernommen werden können und welche neu berechnet werden müssen. Durch das hier verwendete Vorgehen kann auch bei voll-dynamischen Szenen die Darstellung von globaler Beleuchtung bei gleichbleibender Geschwindigkeit gewährleistet werden. Für überwiegend statische Szenen wäre ein dynamisches Aktualisieren der Datenstruktur selbstverständlich performanter.
In Anbetracht des gegebenen Zeitrahmens und dem Umfang der Arbeit beschränkt sich diese Implementierung nur auf die diffuse indirekte Beleuchtung und die erste indirekte Reflexion. Prinzipiell wären jedoch auch beliebig viele Reflexionen und auch andere Phänomene, wie z.B. glänzende Reflexion, möglich, mehr dazu in Kapitel 6, Absatz 2.
Die Implementierung erfolgt in OpenGL 4.5, da so prinzipiell Plattform-Unabhängigkeit gewährleistet werden kann. Ältere Versionen von OpenGL wie 3.x bieten nach Evaluation leider nicht alle benötigten Funktionen, um die Algorithmen mit der nötigen Performance umzusetzen. Rendering-Engines wie Ogre3D sind leider ebenfalls deutlich eingeschränkt was die Funktionen bzw. die verwendete OpenGL Version betrifft. Speziell in neueren Versionen von Ogre3D ist zudem das verwendete Material-System bezogen auf die Anforderungen von Voxel Cone Tracing ungeeignet sowie die Dokumentation allgemein spärlich.

4.2 Algorithmen und Datenstrukturen

4.2.1 Datenstruktur

Bei der im Überblick angesprochenen Datenstruktur handelt es sich um mehrere dreidimensionale Uniform Grids (Erläuterung siehe 2.4.1.1) gleicher Auflösung, die sich gegenseitig überlappen und jeweils das achtfache Volumen im Raum repräsentieren, sprich ein Volumen

doppelter Seitenlänge. Jedes Grid entspricht so einem Level Of Detail. Das Volumen, welches von der Datenstruktur abgedeckt wird, umschließt die Kamera und verschiebt sich mit ihr zur Laufzeit, so dass zu jedem Zeitpunkt der Bereich um die Kamera abgedeckt ist. Abb. 4.1 visualisiert den Aufbau der Übersichtlichkeit halber in 2D.

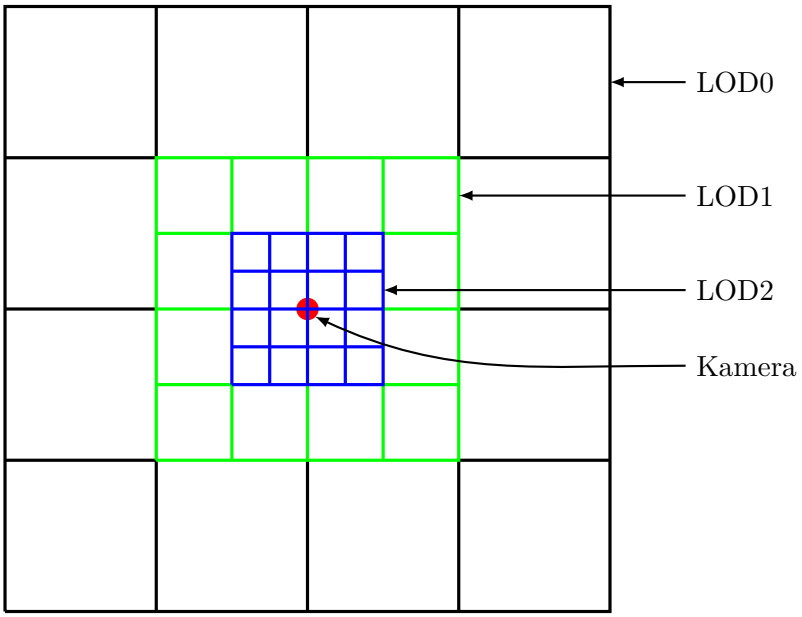

Abbildung 4.1 Aufbau der Datenstruktur des Voxel-Volumens in 2D.

Diese Datenstruktur entspricht im Prinzip der sogenannten Voxel Texture Cascade, welche in der Implementierung von Q-Games, siehe Abschnitt 3.3, verwendet wurde. Der interne Aufbau wird später in Abschnitt 4.3.2 erläutert.
Entschieden wurde sich für diese Datenstruktur, weil sie im Vergleich zu z.B. Clipmaps 2.4.3.2 und besonders SVOs 3.1 deutlich einfacher zu implementieren und dabei doch relativ flexibel ist. Dies hält den Umfang der Arbeit in einem vertretbaren Rahmen. Zudem sind Kosten des (Neu-)Erstellens und Befüllens durch die Einfachheit der Datenstruktur vergleichsweise gering, was der Geschwindigkeit im Hinblick auf das gewählte und im vorigen Abschnitt beschriebene Update-Verfahren zugutekommt.

4.2.2 Voxelisierung der Geometrie

Um die Geometriedaten in die Voxel-Datenstruktur einzufügen wird, wie bei den anderen vorgestellten Implementierungen, auf den Rasterisierer der Grafikkarte zurückgegriffen und, ähnlich wie in der Implementierung von NVIDIA, in einem einzigen Pass jedes Dreieck auf die Ebene des Koordinatensystems projiziert, auf dem es die größte Fläche abdeckt. Anschließend wird aus dieser Sicht mit orthogonaler Projektion gerendert und die Daten in den entsprechenden Voxel gespeichert (siehe Algorithmus 2). Es wird nur in das feinste verfügbare LOD geschrieben.

Die Auflösung, in der in diesem Schritt gerendert wird, kann über einen Parameter eingestellt werden. Zwecks höherer Geschwindigkeit und Reduktion von Arbeitsaufwand wird auf weitere Schritte, wie Antialiasing, Conservative Rasterization oder Rückprojektion auf die anderen Ebenen, verzichtet, was zu einem etwas schlechteren, aber vertretbar hohen Erfassungsgrad der Geometrie führt. Durch Conservative Rasterization würden die Dreiecke temporär leicht vergrößert, so dass sie bei der Rasterisierung auf jeden Fall von einem Fragment erfasst werden würden, Antialiasing würde durch die größere Anzahl an Samples pro Fragment die Chance der Erfassung zumindest erhöhen.

Algorithmus 2 Grober Ablauf der Voxelisation

```
function VoxelizePolygonalGeometry
    disable depth test and culling
    for all triangles do
        calculate best suiting main axis from normal
        do corresponding orthogonal projection
        rasterize from corresponding viewport
        for all fragments do
            store material data in corresponding voxel of voxel-datastructure
```

4.2.3 Berechnen und Speichern der Beleuchtungsdaten

Das Einfügen der Beleuchtungsdaten in die Voxel Datenstruktur geschieht hier ähnlich wie in den Implementierungen von Crassin et al. und NVIDIA. Es wird aus Sicht jeder Lichtquelle gerendert (Point Lights sechsmal, je Richtung einmal) und für jedes sichtbare Fragment die entsprechenden Daten in den dazugehörigen Voxel abgespeichert. Directional Lights werden dabei orthogonal gerendert, Point- und Spot Lights perspektivisch. Auch hier ist die Rendering-Auflösung einstellbar und es wird nur in das feinste verfügbare LOD geschrieben. In diesem Fall wird direkt die Irradianz berechnet und gespeichert, was die Menge an benötigtem Speicher minimiert. Zudem werden in dieser eigenen Implementierung jeweils mehrere Spot-, Point-, und Directional Lights unterstützt, was nicht selbstverständlich ist. In den in Kapitel 3 vorgestellten Implementierungen wird nicht explizit auf eine Unterstützung mehrerer Lichtquellen der verschiedenen Typen hingewiesen. Die finale Irradianz eines Voxels ergibt sich nach der Multiplikation des über alle Lichtquellen und Fragmente gemittelten Wertes mit der Anzahl der relevanten Lichtquellen. Die verwendete Art des Einfügens der direkten Beleuchtung bietet allgemein den Vorteil, dass sie leicht mit Shadow Mapping kombiniert werden kann, bei dem ebenfalls aus Sicht der Lichtquellen gerendert werden muss. In Anbetracht des Umfangs dieser Arbeit wurde hierauf allerdings verzichtet. Nachfolgend wird der Ablauf in Algorithmus 3 verdeutlicht.

Algorithmus 3 Grober Ablauf des Einfügens der Beleuchtungsdaten

```
function InjectLighting
    enable early depth test
    for all lightsources do
        transform to lightsources view
        for all triangles do
            for all fragments do
                calculate averaged irradiance and increase lights-counter
                store irradiance and lights-counter in corresponding voxel
        clear z-buffer
    for all voxels do
        calculate final irradiance by multiplying with lights-counter
```

4.2.4 Propagieren der Daten

Anschließend werden die in den Voxeln gespeicherten Daten durch die Hierarchie der LODs propagiert, da beim Einfügen der Daten nur in dem jeweiligen Level gespeichert wurde. Dies geschieht iterativ vom feinsten bis zum gröbsten Level, wobei jeweils die Daten von 8 Voxeln des feineren Levels zu einem des gröberen Levels zusammengefasst werden. Algorithmus 4 zeigt den allgemeinen Ablauf. Das Vorgehen anfangs nur in je ein LOD zu schreiben und anschließend alle Voxel-Daten zu propagieren bietet den Vorteil, die hohe Parallelisierbarkeit des Problems auszunutzen und ohne explizite Synchronisation auszukommen, da gleichzeitig auf allen Voxeln eines Levels gearbeitet werden kann.

Algorithmus 4 Grober Ablauf des Propagierens der Daten

```
function PropagadeData
    for all LODs − 1 do
        dispatch compute call
        for all voxels of LOD do
            average material and irradiance values from corresponding finer voxels
            store averaged values in voxel of coarser current LOD
```

4.2.5 Cone Tracing

In einem weiteren Arbeitsschritt wird die Szene aus Sicht der Kamera gerendert und approximatives Cone Tracing auf die Voxel-Datenstruktur angewandt. Dazu werden für $n \cdot n$ Pixel, jeweils von dessen Position in Weltkoordinaten aus, eine Reihe von Cones ausgesandt, die durch das Voxel-Volumen laufen und Daten von Sample-Positionen akkumulieren. Dabei ist n ein Parameter, der die Cone Tracing-Auflösung in Abhängigkeit der finalen Auflösung bestimmt. Die indirekte Beleuchtung wird anschließend über eine gewichtete Summe über die Resultate der Cones berechnet. Das grundlegende Prinzip entspricht hier dem der zuvor vorgestellten Implementierungen, jedoch gibt es auch einige Unterschiede in Detailaspekten,

wie z.B. der Anzahl der Cones. Auf diese Aspekte wird in Abschnitt 4.3.3.4 eingegangen. Der grobe Ablauf wird in Algorithmus 5 veranschaulicht.

Algorithmus 5 Grober Ablauf des Cone Tracings

```
function ConeTracing
    use main viewport
    for all triangles do
        for all fragments do
            calculate cone directions
            for all cones do
                while dist < max and opacity < 1 do
                    calculate sample position and increase dist
                    get color and irradiance data from sample
                    accumulate cone color and opacity
            calculate final color by weighted sum of cone results
            write final color as indirect lighting to texture
```

Nur für jeden n-ten Pixel die indirekte Beleuchtung zu ermitteln geschieht aus demselben Grund wie bei NVIDIA und der Implementierung von Q-Games, im Schnitt ist die Änderung der Farb- und Helligkeitswerte von Pixel zu Pixel nicht so stark, dass eine höhere Abtastung einen deutlichen Mehrwert bringen würde. So sinkt die Zahl der nötigen Berechnungen für das Cone Tracing quadratisch mit n.

4.2.6 Finale Beleuchtung

Im letzten Schritt wird die direkte Beleuchtung der Szene über gewöhnliches Rendern aus Sicht der Kamera und Anwenden eines lokalen Beleuchtungsmodells, vgl. 2.2, berechnet. In diesem Fall wird das Blinn-Phong-Beleuchtungsmodell genutzt. Dabei muss die indirekte Beleuchtung, die zuvor ggf. in einer niedrigeren Auflösung berechnet wurde, bilinear interpoliert und zur direkten Beleuchtung addiert werden, siehe Algorithmus 6. Das Blinn-Phong-Beleuchtungsmodell wird verwendet, weil es einen guten Kompromiss aus Darstellungsqualität und Berechnungskosten darstellt.

Algorithmus 6 Grober Ablauf der Berechnung der finalen Beleuchtung

```
function FinalLighting
    use main viewport
    for all triangles do
        for all fragments do
            calculate direct lighting using blinn-phong
            interpolate indictect lighting data
            combine direct and indirect lighting
```

Wie im Überblick angesprochen, wird Forward-Rendering genutzt. Dies hat den Grund, dass es leicht zu implementieren ist und bei einer niedrigen Anzahl von Lichtquellen sich die Extrakosten, die durch das wiederholte Rendern der Geometrie für jede Lichtquelle anfallen, in Grenzen halten. Prinzipiell wäre auch Deferred Rendering möglich, falls viele Lichtquellen gleichzeitig dargestellt werden sollen.

4.3 Implementierungsdetails

Hier wird die konkrete, eigene Implementierung der im vorigen Abschnitt beschriebenen Algorithmen und Datenstrukturen vorgestellt. Dabei werden zuerst Aufbau und Ablauf des Programms veranschaulicht und anschließend auf für das Voxel Cone Tracing relevante Teile und besonders interessante Details eingegangen. Dieser Abschnitt endet mit einer Komplexitätsanalyse.

4.3.1 Allgemeiner Programmaufbau und -ablauf

In Abb. 4.2 ist ein Klassendiagramm zu sehen, welches einen Überblick über das Programm und dessen Aufbau gibt. Der Übersichtlichkeit halber werden nebensächliche Funktionen und Attribute nicht explizit dargestellt.

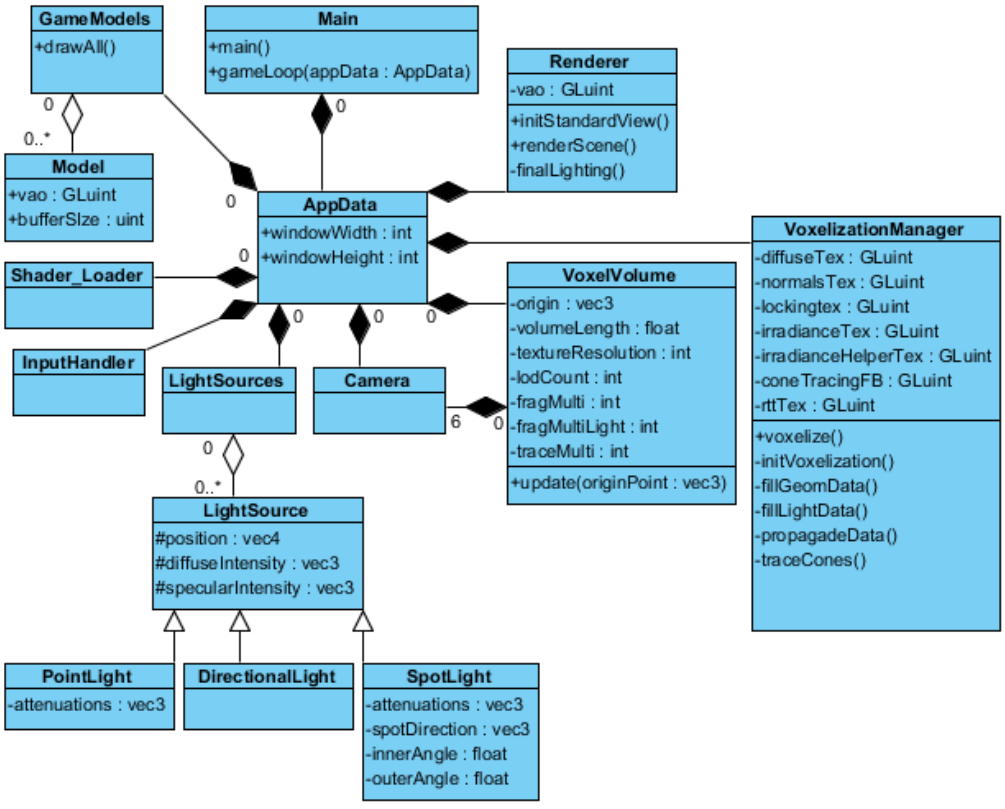

Abbildung 4.2 Aufbau des Programms mit den wichtigsten Funktionen und Attributen.

Nachfolgend eine kurze Beschreibung der Klassen:

- **Main**
  Hier werden OpenGL-Kontext, Fenster und alle weiteren Objekte erzeugt sowie die Game-Loop ausgeführt.

- **AppData**
  Diese Klasse dient zur programmweiten Datenhaltung. Hier befinden sich die meisten Daten, die klassenübergreifend benötigt werden, so dass meist nur eine Referenz oder ein Pointer auf die zu Beginn erstellte Instanz als Funktionsargument übergeben werden muss. Auch Pointer zu den Instanzen der anderen Klassen befinden sich hier.

- **Camera**
  Diese Klasse enthält die Funktionalität einer typischen First-Person-Kamera. Sie lässt sich über diverse Parameter einstellen. Zudem kann eine orthographische oder perspektivische Kamera erstellt werden.

- **Shader_Loader**
  Diese Klasse kompiliert Shaderdateien aus entsprechenden Textdateien und linkt diese zu einem Shaderprogramm.

- **LightSource**
  LightSource ist die Basisklasse für alle Lichtquellen. Sie wird erweitert von den Klassen *DiretionalLight*, *SpotLight* und *PointLight*. Grundlagen der Arten von Lichtquellen siehe Kapitel 2. Alle in der Szene vorkommenden Lichtquellen werden in *AppData* von *LightSources* gehalten.

- **VoxelVolume**
  Diese Klasse dient zur Speicherung von Daten bezüglich des Voxel-Volumens und von Parametern zur Voxelisierung. Über eine Update-Methode werden die Daten des Volumens aktuell gehalten, wenn sich die Kamera des Nutzers und mit ihr das Volumen in der Szene bewegt.

- **InputHandler**
  Diese Klasse verarbeitet Tastatur- und Mausevents. Tastendrucke werden in einer Klasse *KeyState*, welches von *AppData* gehalten wird, gespeichert. Auf Basis des KeyStates wird die Bewegungsfunktion der Kamera ausgeführt. Mausbewegungen werden zur weiteren Verarbeitung an das Kameraobjekt des Nutzers weitergeleitet.

- **GameModels**
  GameModels bietet die Funktionalität, Modelle aus Dateien zu laden und in ingame-Modelle zu übersetzen, die gerendert werden können.

- **VoxelizationManager**
  Diese Klasse beinhaltet die Logik des eigentlichen Voxel Cone Tracing-Algorithmus. Angefangen mit dem Voxelisieren der Geometrie, über das Einfügen der Beleuchtung in die Voxel und dem Propagieren der Daten über die LODs, bis hin zum Cone Tracing selbst. Ausführliche Beschreibung siehe Abschnitt 4.3.3.

- **Renderer**
  Diese Klasse ist für das allgemeine Rendering und den Ablauf dessen zuständig. Unter anderem werden hier alle nötigen Buffer erstellt sowie die finale Beleuchtung berechnet und angezeigt, mehr dazu in Abschnitt 4.3.4.

In Abb. 4.3 ist ein Sequenzdiagramm dargestellt, welches den Ablauf der wichtigsten Funktionen der Implementierung von Voxel Cone Tracing visualisiert.

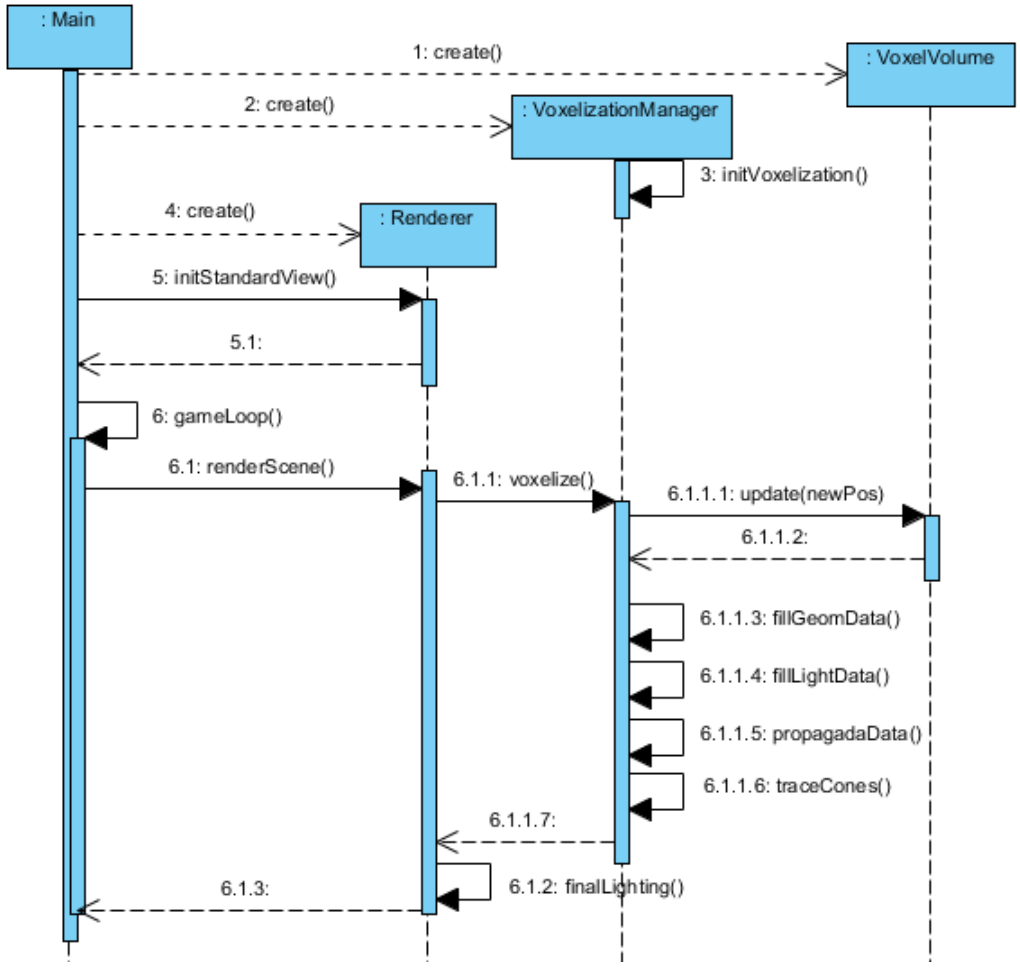

Abbildung 4.3 Ablauf der Kernfunktionen der Voxel Cone Tracing-Implementierung.

Eine detaillierte Beschreibung der Datenstruktur, der Voxel Cone Tracing-Schritte und des finalen Renderings folgt in den Abschnitten 4.3.2, 4.3.3 und 4.3.4.

4.3.2 Details der Datenstruktur

Durch die Funktion *initVoxelization()* werden die 3D-Texturen angelegt, die zusammen die Voxel-Datenstruktur bilden. Für die diffuse Farbe, Irradianz, Hilfsvariablen für die Irradianz sowie Normale wird jeweils eine 3D-Textur des Formats GL_RGBA16F und für eine zur Synchronisation verwendete Textur GL_R32F genutzt. Andere Formate werden leider nicht von den nötigen Befehlen unterstützt, daher gibt es ungenutzte Kanäle in der Irradianz-Hilfsvariablen-Textur. Alle Texturen haben die Auflösung $(n, n, n \cdot l)$, wobei n die Auflösung des Voxel-Volumens und l die Zahl der LODs ist. Ein Voxel besteht demnach aus jeweils einem Texel dieser fünf Texturen. Die Daten aller LODs werden hintereinander und blockweise in derselben Textur gespeichert, angefangen mit LOD0. Abb. 4.4 zeigt diesen Aufbau.

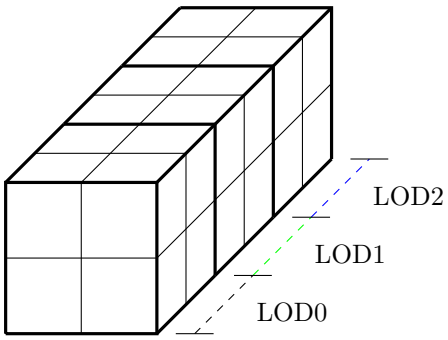

Abbildung 4.4 Datenlayout aller 3D-Texturen, hier am Beispiel der diffusen Farbe.

Welchen Raum in der Szene die einzelnen LODs repräsentieren, zeigt Abb. 4.1. Zudem wird für den Cone Tracing-Schritt ein separater Framebuffer mit Farb- und Tiefenkomponente angelegt, in dem später per Render-To-Texture die indirekte Beleuchtung gespeichert wird.

4.3.3 Details der Voxel Cone Tracing-Schritte

Die Funktion *voxelize()* der Klasse *VoxelizationManager* dient als Schnittstelle nach außen und führt das Voxel Cone Tracing aus, wobei die einzelnen Schritte des Algorithmus in private Unterfunktionen ausgelagert sind.
Zu Beginn wird die Update-Funktion des Voxel-Volumens ausgeführt, anschließend alle Texturen zurückgesetzt. So ist das Volumen wieder um die ggf. bewegte Kamera zentriert und alle vorherigen, potentiell veralteten Daten gelöscht. Für die Ausführung der Funktionen *fillGeomData()*, *fillLightData()* und *propagadeData()* wird das Schreiben des Color-Buffers deaktiviert, denn dies wäre unnötig und würde Rechenzeit kosten. Schließlich wird der Color-Buffer vor Ausführung von *traceCones()* wieder aktiviert.

4.3.3.1 Details der Voxelisierung der Geometrie

fillGeomData() voxelisiert die polygonale Geometrie und speichert diffuse Oberflächenfarbe in den Voxeln bzw. den zugehörigen 3D-Texturen. Die Auflösung, in welcher die Voxelisierung geschieht, wird über einen Parameter bestimmt und ist durch diesen ein Vielfaches von der Voxel-Auflösung. Somit kann auch bei geringeren Voxel-Auflösungen eine genauere Voxelisierung durchgeführt werden. Um die komplette Geometrie, unabhängig ihrer Sichtbarkeit, zu berücksichtigen, wird für diese Funktion der Z-Test deaktiviert. So kann auch verdeckte Geometrie sichtbare Bereiche indirekt beleuchten. Backface Culling wird auch deaktiviert.
Der Vertex-Shader reicht die Farbe, Position und Normale der Vertices in Weltkoordinaten an den Geometrie-Shader weiter. Dieser berechnet die Normale des Dreiecks und zu welcher der Weltkoordinatensystem-Hauptachsen die Winkel-Differenz am geringsten ist, siehe Listing 4.1. Anschließend wird das Dreieck mit der zugehörigen View-Projektionsmatrix orthogonal auf die entsprechende Ebene projiziert. Diese drei Projektionsmatrizen werden per uniform übergeben und gehören zu zuvor einmalig erstellten Kameras des Voxel-Volumens. Diese View-Projektion verbessert die Abdeckung beim Rasterisieren, da jedes Dreieck so auf der Ebene gerastert wird, auf dem es die größte Fläche einnimmt.

```
1  vec3 edge01 = (gl_in[1].gl_Position - gl_in[0].gl_Position).xyz;
2  vec3 edge02 = (gl_in[2].gl_Position - gl_in[0].gl_Position).xyz;
3  vec3 triangleNormal = normalize(cross(edge01, edge02));
4
5  float axisX = abs(dot(triangleNormal, vec3(1, 0, 0)));
6  float axisY = abs(dot(triangleNormal, vec3(0, 1, 0)));
7  float axisZ = abs(dot(triangleNormal, vec3(0, 0, 1)));
```

Listing 4.1 Berechnung der Normale eines Dreiecks und der Hauptachse mit geringster Differenz.

Der Fragment-Shader berechnet nachfolgend mittels zusätzlich in World Space weitergereichten Koordinaten, in welchem LOD sich das jeweilige Fragment befindet und verwirft die, die sich außerhalb des Voxel-Volumens befinden. Anschließend werden die Koordinaten im Voxel-Volumen berechnet und über eine Busy-Waiting-Synchronisation die Farb- und Normaleninformationen, gemittelt mit den bereits darin befindlichen, in den zugehörigen Voxel gespeichert. Diese Synchronisation ist nötig, da, wie am Anfang dieses Abschnitts erläutert, je nach Auflösungsparameter mehrere Fragmente denselben Voxel und damit Texel in den Texturen abdecken. Ein paralleles Schreiben würde zu Fehlern führen. Für das Locking werden eine 3D-Textur mit Integerformat sowie atomic-Befehle verwendet, so dass jeweils nur eine Shaderinstanz zurzeit auf derselben Speicheradresse arbeiten kann. Jedes Texel der Locking-Textur dient als Spinlock. Das Lesen und Schreiben der Texturen geschieht über imageLoad und -Store-Befehle, welche ermöglichen, dass aus den Shadern heraus beliebige Texel gelesen und geschrieben werden können, vgl. Listing 4.2. Zudem müssen die Texturen, hier als image-Variable gebunden, mit dem coherent-Qualifizierer versehen werden, um die Kohärenz über mehrere Shaderinstanzen hinweg gewährleisten zu können.

```
1  bool waiting = true;
2  while(waiting)
3  {
4    if(imageAtomicCompSwap(lockingTex, voxelPosLod, 0, 1) == 0)
5    {
6      vec4 normalOld = imageLoad(normalsTex, voxelPosLod);
7      vec4 diffuseOld = imageLoad(diffuseTex, voxelPosLod);
8      //averaging and counter increase (normal.w)
9      vec4 diffuseNew = normalOld.w / (normalOld.w + 1) * diffuseOld + 1 / (normalOld.
         w + 1) * vec4(fragIn.colorInterp, 1);
10     vec4 normalNew = normalOld.w / (normalOld.w + 1) * normalOld + 1 / (normalOld.w
         + 1) * vec4(fragIn.normalInterp, 1);
11     normalNew.w = normalOld.w + 1;
12     imageStore(diffuseTex, voxelPosLod, diffuseNew);
13     imageStore(normalsTex, voxelPosLod, normalNew);
14     memoryBarrier();
15     waiting=false;
16    imageAtomicExchange(lockingTex, voxelPosLod, 0);
17   }
18 }
```

Listing 4.2 Synchronisiertes Mitteln und Schreiben der Geometriedaten in die Voxel.

Abschließend wird ein glMemoryBarrier-Befehl ausgeführt, um sicherzustellen, dass alle Schreiboperationen auf dem Speicher abgeschlossen wurden, bevor der nächste Schritt des Algorithmus beginnt.

4.3.3.2 Details des Berechnens und Speicherns der Beleuchtungsdaten

Nach dem Voxelisieren wird durch *fillLightData()* aus Sicht jeder Lichtquelle gerendert, um die Irradianz für jeden Voxel durch den Fragment-Shader zu berechnen und aufzusummieren. Dazu wird iterativ für jeden Lichtquellen-Typ und dann jede derartige Lichtquelle der Renderbefehl vorbereitet und ausgeführt. Für jede Art Lichtquelle steht eine vorkonfigurierte Kamera zur Verfügung, so muss nur noch jeweils die Blickrichtung und Position aktualisiert werden. Für Point Lights werden 6 Renderbefehle durchgeführt, je einer pro Grundrichtung mit 90 Grad Field Of View (FOV). So wird die komplette Sphäre um die Lichtquelle erfasst. Nach jedem Renderbefehl muss per glMemoryBarrier-Befehl die Beendigung der Schreiboperationen sichergestellt werden und die Locking-Textur sowie der Z-Buffer zurückgesetzt werden, um Fehler auszuschließen. Auch muss explizit der Early-Z-Test aktiviert werden, da der Z-Test sonst erst nach dem Fragment-Shader und den imageStore-Befehlen erfolgen würde. Dadurch würden fälschlicherweise auch Beleuchtungsdaten in von der Lichtquelle aus nicht sichtbaren Voxeln gespeichert werden.
Vertex- und Geometrie-Shader reichen Farb-, Positions- und Normalendaten im View Space weiter, die Position zusätzlich in Weltkoordinaten. Der Fragment-Shader ist im Prinzip ähnlich wie der der Funktion *voxelize()* aufgebaut. Hier wird jedoch die Irradianz statt der Geometriedaten berechnet und gemittelt. Da pro Lichtquelle mehrere Fragmente in denselben

Voxel fallen können und gemittelt werden müssen, aber die Ergebnisse der verschiedenen Lichtquellen addiert werden müssen, wird zuerst der Durchschnitt über alle Lichtquellen und Fragmente berechnet. Um durch Multiplikation des Durchschnittswerts mit der jeweiligen Anzahl der Lichtquellen das Endergebnis zu berechnen, wird pro Voxel zudem gespeichert, wie viele Lichtquellen in das Durchschnittsergebnis eines Voxels einflossen. Dazu dient die Irradianz-Hilfsvariablen-Textur. Die Synchronisation ist analog zu der bei dem Voxelisieren. Die Berechnung des Irradianzanteils jeder Lichtquelle geschieht über das Lambertsche Kosinusgesetz, je nach Lichtquelle mit oder ohne Attenuation-Berechnung. Liegt der Beleuchtungsanteil unter einem Schwellwert und wäre so nicht relevant, wird die Shaderinstanz verworfen, um Rechenleistung zu sparen.
Um pro Voxel die endgültige Irradianz aus dem Durchschnittswert zu berechnen, wird über einen Compute-Shader jeweils der Irradianzwert mit der Anzahl der beteiligten Lichtquellen multipliziert.

4.3.3.3 Details des Propagierens der Daten

Da bei dem Einfügen der Geometrie- und Beleuchtungsdaten in die Voxel nur in die jeweils feinste vorhandene LOD-Stufe geschrieben wird, müssen die Daten noch über die gröberen LODs propagiert werden. Dies geschieht iterativ, begonnen mit der feinsten LOD-Stufe i. In jeder Iteration wird ein Compute-Shader mit $(\frac{x}{16}, \frac{x}{16}, \frac{x}{16})$ Work Groups ausgeführt, wobei x die Voxel-Auflösung ist. Jede Work Group hat die Größe $(8, 8, 8)$. Diese Aufteilung auf $8^3 = 512$ Shaderinstanzen pro Work Group führt nach empirischer Messung über verschiedene Voxel-Auflösungen zu der besten gemittelten Geschwindigkeit und liegt noch unter dem von OpenGL definierten Maximalwert von 1024. Speziell bei hohen Auflösungen wie 64^3 oder 128^3 ist der Geschwindigkeitsgewinn im Vergleich zu deutlich weniger Shaderinstanzen pro Work Group beachtlich. Die GlobalInvocationID dient als Koordinate zum Auslesen der 8 Voxel des aktuellen LODs, wobei diese noch verdoppelt und mit einem Offset addiert werden muss, vgl. Listing 4.3, Zeilen 2 und 3.

```
1 ivec3 coords = ivec3(gl_GlobalInvocationID);
2 ivec3 readCoords=coords * 2;
3 vec3 lodOffset = vec3(0, 0, (lod - 1) * voxelResolution);
4 vec3 lodOffsetStore = vec3(0, 0, (lod - 2) * voxelResolution);
5 diffuse[0] = imageLoad(diffuseTex, ivec3(readCoords + lodOffset));
6 diffuse[1] = imageLoad(diffuseTex, ivec3((readCoords + vec3(0, 0, 1)) + lodOffset));
7   //...
8 irradiance[0] = imageLoad(irradianceTex, ivec3(readCoords + lodOffset));
9   //...
```

Listing 4.3 Berechnung der Koordinaten und Auslesen der Daten der Voxel bei dem Propagieren.

Die Farb- sowie Irradianzdaten der 8 ausgelesenen Voxel werden anschließend gemittelt, wobei die RGB-Komponenten der Farbe direkt mit der Alphakomponente multipliziert und später

durch den Gesamtwert der Alphakomponente dividiert werden, siehe Listing 4.4. Dadurch lässt sich ein undurchsichtiger, schwarzer Voxel unterscheiden von einem leeren und korrekt berücksichtigen.

```
1  for(int j = 0; j < 8; j++)
2  {
3    newDiffuse.xyz += diffuse[j].xyz * diffuse[j].w;
4    newDiffuse.w += diffuse[j].w;
5    newIrradiance.xyz += irradiance[j].xyz;
6  }
7  if(newDiffuse.w >= 0.01f)
8  {
9    newDiffuse.xyz /= newDiffuse.w;
10   newDiffuse.w /= 8;
11 }
12 newIrradiance.xyz /= 8;
```

Listing 4.4 Mitteln der Daten bei dem Propagieren.

4.3.3.4 Details des Cone Tracings

Die Funktion *traceCones()* berechnet schließlich die indirekte Beleuchtung, genau genommen den diffusen Anteil, anhand der Daten in der Voxel-Datenstruktur. Um die indirekte Beleuchtung später mit der direkten zu kombinieren, wird ein separater Framebuffer gebunden, in den per Render-To-Texture geschrieben wird, was durch Hardwareoptimierungen schneller sein sollte, als dies über imageLoad/Store-Befehle zu tun. Zudem wird auch hier mit einer separaten (niedrigeren) Auflösung gerendert, was einen Geschwindigkeitsgewinn einbringt. In dieser Implementierung werden pro Fragment neun Cones mit einem Öffnungswinkel von 45 Grad ausgesandt, einer in Richtung der Normalen, die anderen acht mit einem Winkel von 50 Grad um diese rotiert. Prinzipiell wären auch andere Werte möglich, wie z.B. 5 Cones wie in 3.1, diese hier gewählten stellen allerdings einen vernünftigen Kompromiss aus Granularität und Geschwindigkeit dar.
Der Fragment-Shader verwirft zunächst alle Fragmente, die sich nicht im Voxel-Volumen befinden. Anschließend werden nacheinander die einzelnen Cones berechnet und ausgesandt. Gewichtet werden ihre Ergebnisse wie der diffuse Anteil im Blinn-Phong-Beleuchtungsmodell, vgl. 2.2, über das Gaußsche-Kosinusgesetz, da sie als separate Lichtquelle betrachtet werden können, welche dieses Fragment beleuchten. Zudem wird der über die Cone-Ergebnisse akkumulierte RGB-Anteil wie im Schritt des Propagierens zwecks Normalisierung durch die Alpha-Komponente geteilt und diese durch einen separaten Parameter.
Jeder Cone führt eine Art Volumen-Raymarching durch, d.h. liest zwischen Minimal- und Maximaldistanz in bestimmten Abständen die Daten im Voxel-Volumen aus und akkumuliert diese. Abbruchkriterien der Schleife sind einerseits die erwähnte Maximaldistanz und andererseits eine Undurchsichtigkeit von 100%. Die Wahl der Sample-Positionen wird über den Cone-Radius und die vorherige Position getroffen, so dass mit steigender Distanz die

Schrittgröße steigt und sich die Samples möglichst wenig überlappen. Für jedes Sample wird zuerst das LOD berechnet, welches an der Position verfügbar ist und dessen Voxel-Länge möglichst ähnlich des Sample-Durchmessers ist. Dies sorgt für eine gute Approximation der Volumendaten durch den Cone. Danach werden die Daten des entsprechenden Voxels ausgelesen. Anschließend wird das Sample verworfen, wenn dessen zugrunde liegende Geometrie dem Cone abgewandt ist oder der Voxel bereits von diesem Cone ausgelesen wurde. Ansonsten ist das Sample valide und diffuse Farbe und Irradianz werden multipliziert und Farbe und Undurchsichtigkeit über die im Volumen-Rendering üblichen Formeln akkumuliert, wobei eine quadratische Attenuation über die bisherige Distanz hinzukommt. Listing 4.5 zeigt diese Akkumulation.

```
1 coneColor += (1.0 - coneOpacity) * vec4(sampleIrradiance.xyz, 1) * sampleDiffuse * (1
    / (1 + dist * dist));
2 coneOpacity += (1.0 - coneOpacity) * sampleIrradiance.w * sampleDiffuse.w;
```

Listing 4.5 Akkumulation der Samples eines Cones.

Erweiterbar wäre das Cone Tracing durch einen in Reflexionsrichtung ausgesandten weiteren Cone mit geringerem Öffnungswinkel, um den glänzenden Anteil der indirekten Beleuchtung einzufangen. Aus Zeitgründen wird darauf hier verzichtet. Zudem ließe sich der Cone Tracing Schritt iterativ auf zwischengespeicherten Daten wiederholen, um mehrere Reflexionen des indirekten Lichtes zu berechnen. Dies würde allerdings die Kosten und den Speicherbedarf stark in die Höhe treiben, zumal jede weitere Reflexion anteilig weniger zum Endergebnis beiträgt.

4.3.4 Details des Renderings und der finalen Beleuchtung

Die Klasse *Renderer* ist hauptverantwortlich für den Ablauf des Renderings und die Initialisierung aller nötigen Buffer und Rendertargets. Die finale Beleuchtung wird hier direkt berechnet, die anderen Schritte werden über *voxelize()* von *VoxelizationManager* ausgeführt.

Mit der Funktion *initStandardView()* wird das zum Rendering nötige VertexArrayObject und Buffer für Vertexpositionen, -normalen, -farben sowie -indices erstellt und konfiguriert. Befüllt werden die Buffer mit fest kodierten, statischen Daten.

Die Funktion *renderScene()* ist Ausgangspunkt des Renderings. Als erstes werden Farb- und Z-Buffer zurückgesetzt und das entsprechende VertexArrayObject gebunden. Anschließend wird die *voxelize()* Funktion des *VoxelizatitonManagers* ausgeführt, welche die Hauptfunktionen des Voxel Cone Tracings umsetzt. Danach wird mit der endgültige Renderauflösung *finalLighting()* aufgerufen.

Diese Methode führt schließlich einen Shader aus, der die direkte Beleuchtung in der finalen Renderauflösung und aus Sicht der Hauptkamera, welche vom Nutzer kontrolliert wird, berechnet. Dazu wird nacheinander für alle Lichtquellen jeden Typs mittels Blinn-Phong-Beleuchtungsmodell, vgl. 2.2, der jeweilige Beleuchtungsanteil berechnet und aufsummiert. Anschließend wird die bereits berechnete Irradianz der indirekten Beleuchtung aus einem per uniform übergebenen sampler-Objekt ausgelesen und bilinear interpoliert. Dies ist nötig, da die indirekte Beleuchtung mit einer niedrigeren Auflösung als der finalen Fensterauflösung berechnet werden kann, um Rechenleistung zu sparen. Die indirekte Beleuchtung wird mit der Oberflächenfarbe der Geometrie an diesem Fragment multipliziert und mit der direkten Beleuchtung kombiniert. Das Resultat ergibt die endgültige Pixelfarbe.
Obwohl eine zusätzliche Filterung der indirekten Beleuchtung, z.B. mit einem Gauß-Filter oder einem bilateralen Filter, das Ergebnis deutlich aufwerten würde, indem die harten Übergänge in dem indirekten Beleuchtungsanteil aufgeweicht werden würden, muss bedauerlicherweise aus Zeitgründen auf eine solche verzichtet werden.

4.3.5 Komplexitätsanalyse

In diesem Abschnitt soll die Komplexität des Algorithmus bezüglich Speicher- und Zeitbedarf analysiert werden, dazu werden folgende Variablen definiert:

- C_{proj} = Berechnen der Hauptachse eines Dreiecks und Projektion
- C_{mat} = Speichern der Materialdaten pro Pixel
- C_{avgIrr} = Berechnen, Mitteln und Speichern der Irradianz pro Pixel
- C_{finIrr} = Berechnen und Speichern der finalen Irradianz pro Voxel
- $C_{avgData}$ = Mitteln und Speichern der Voxel-Daten
- C_{cones} = Berechnen der Cones
- C_{sample} = Berechnen der Sample-Position und -Daten
- C_{interp} = Bilineare Interpolation eines Wertes
- C_{blinn} = Berechnung von Blinn-Phong-Lighting pro Lichtquelle und Pixel
- t =# Dreiecke
- l =# LODs
- i =# Lichtquellen
- s =# Samples pro Cone
- c =# Cones pro Fragment

- n =# Pixel pro Dimension bei der finalen Beleuchtung
- n_{vox} =# Pixel pro Dimension bei der Voxelisierung
- n_{inj} =# Pixel pro Dimension bei dem Light Injecting
- n_{tra} =# Pixel pro Dimension bei dem Cone Tracing
- v =# Voxel pro Dimension pro LOD

Durch die verschiedenen Auflösungen in den Schritten des Algorithmus gibt es auch verschiedene Pixel-Variablen. Die Dimension aller Pixel-Variablen ist zwei, die der Voxel drei.

Die Komplexität bezüglich des Speicherbedarfs beträgt nach 4.3.2 :

$$5 \cdot l \cdot v^3 \Rightarrow O(v^3)$$

Der benötigte Speicherplatz steigt also erwartungsgemäß linear mit der Anzahl der LODs und kubisch mit der Anzahl der Voxel. Die Anzahl der Voxel ist also der maßgebende Faktor.

Die Zeitkomplexität setzt sich wie folgt aus den einzelnen Komplexitäten der Schritte des Algorithmus zusammen:
Voxelisierung der Geometrie(4.2.2):

$$t \cdot (C_{proj} + n_{vox}^2 \cdot C_{mat})$$

Einfügen der Beleuchtung: (4.2.3)

$$i \cdot t \cdot (n_{inj}^2 \cdot C_{avgIrr}) + v^3 \cdot l \cdot C_{finIrr}$$

Propagieren der Daten: (4.2.4)

$$(l-1) \cdot ((v^3/4) \cdot C_{avgData})$$

Cone Tracing: (4.2.5)

$$t \cdot (n_{tra}^2 \cdot (C_{cones}(c \cdot (s \cdot C_{sample}))))$$

Finale Beleuchtung: (4.2.6)

$$t \cdot (n^2 \cdot (C_{interp} + i \cdot C_{blinn}))$$

Die Gesamt-Zeitkomplexität ist schließlich die Summe der Teile.

$$O(i \cdot t \cdot n_{inj}^2 + t \cdot (n^2 + n_{vox}^2 + n_{tra}^2) + v^3) \Rightarrow O(v^3)$$

Hier ist ersichtlich, dass die Anzahl der Voxel durch ihr kubisches Wachstum auch den größten Einfluss auf die Zeitkomplexität hat. Die Menge der Dreiecke und die einzelnen Mengen der Pixel kommen zwar häufig als Faktor vor, aber nur mit linearem bzw. quadratischem Wachstum.

Wie sich die Performance in der Praxis in Abhängigkeit der Variablen verhält, wird in Abschnitt 5.1 untersucht.

Kapitel 5

Ergebnisse und Evaluation

5.1 Auswertung

Im Folgenden werden die Ergebnisse der selbstgeschriebenen Voxel Cone Tracing Implementierung betrachtet.
Getestet wurde mit einer GeForce GTX 1060 von NVIDIA auf Windows 7 und einer Programmauflösung von 1280 mal 720 Pixeln sowie vierfachem Multisample-Antialiasing. Zudem ist anzumerken, dass alle Ergebnisse in einer Debug-Konfiguration von Microsoft Visual Studio und mithilfe von dem Debugger und Profiler NVIDIA Nsight entstanden sind und die Geschwindigkeit in einer Produktivumgebung eventuell höher ausfallen könnte. Alle Daten bezüglich der Geschwindigkeit wurden aus jeweils 3 Messungen gemittelt.

Die erste, sehr simple Testszene besteht aus einem grauen Boden, einer roten und einer gelb-braunen Wand und einem verschiedenfarbigen Kubus, die alle von einem Point Light beleuchtet werden. Abb. 5.1 zeigt die Szene mit direkter Beleuchtung. Für den Test wurde eine Voxel-Auflösung von 32 und 3 LODs gewählt.

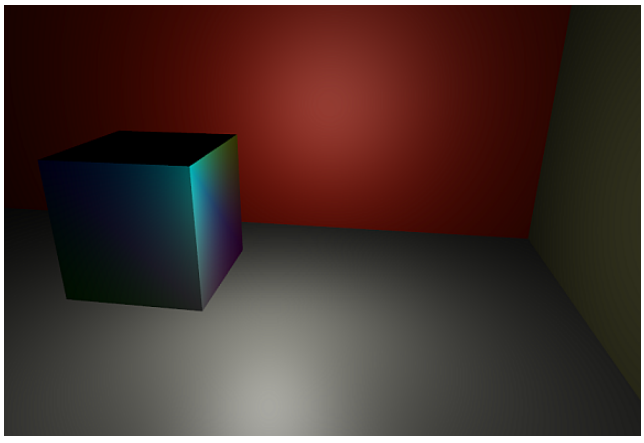

Abbildung 5.1 Eigene Testszene mit direkter Beleuchtung gerendert.

Abb. 5.2 zeigt auf der linken Seite die Szene mit den in der Voxel-Datenstruktur gespeicherten, berechneten Irradianzwerten, auf der rechten Seite einen Ausschnitt mit den in den Voxeln eingefügten Materialfarben. Die punktförmigen, schwarzen Artefakte im linken Bild sind nur durch temporäre Code-Änderungen für dieses Testbild entstanden, wahrscheinlich durch Z-Fighting, und entstehen nicht in der Produktivversion. Man kann anhand beider Bilder sehr gut die Größen der Voxel des jeweiligen LOD-Level und deren Übergang sehen.

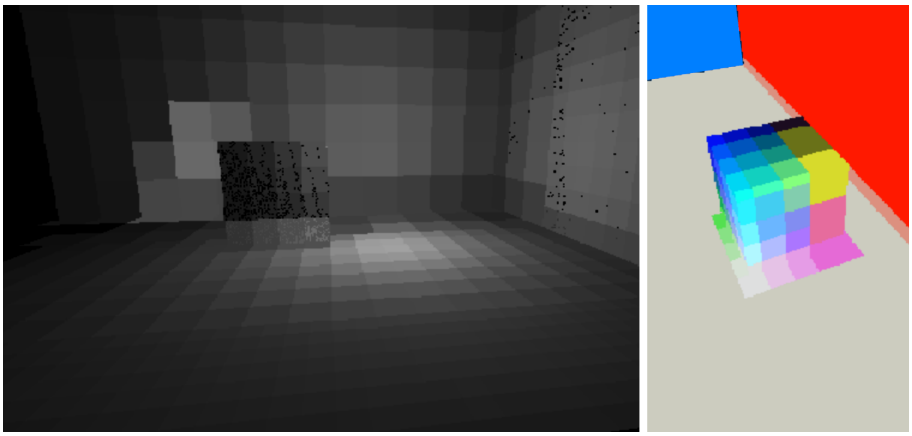

Abbildung 5.2 Linke Seite: Szene mit Irradianzdaten der Voxel-Datenstruktur gerendert. Rechte Seite: Ausschnitt der Szene mit Materialfarben der Voxel gerendert.

In Abb. 5.3 ist das Endergebnis, eine Kombination aus direkter und indirekter Beleuchtung, zu sehen. Man kann vor allem am Boden deutlich erkennen, dass physikalisch plausibel die Farbe von Wänden und Kubus als reflektierte indirekte Beleuchtung von den Oberflächen aufgenommen wird. Mit steigender Entfernung zwischen den Objekten sinkt die Intensität durch die simulierte Absorption des Mediums zwischen Ausgangs- und Endpunkt der (reflektierten) Lichtstrahlen.

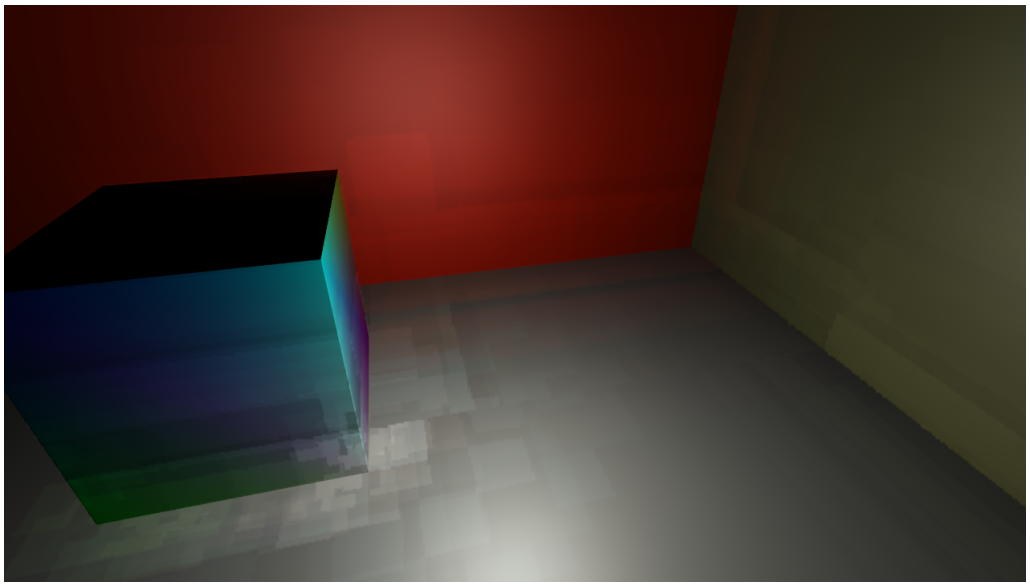

Abbildung 5.3 Endgültiges Bild der Szene mit direkter und indirekter Beleuchtung.

Auf der linken Seite von Abb. 5.4 ist ein Ausschnitt der Szene dargestellt, bei dem nur die indirekte Beleuchtung angezeigt wird, wie sie durch das Voxel Cone Tracing berechnet wird. Hier lässt sich noch deutlicher sehen, wie die Objekte das Licht reflektieren und dieses die Farbe des reflektierenden Objektes annimmt. Auf der rechten Seite ist die Szene wieder mit direkter und indirekter Beleuchtung abgebildet, diesmal aber mit 2 Lichtquellen. Man erkennt den Schein des hinzugekommenen Spot Lights, was ebenfalls für diffuse Reflexionen sorgt.

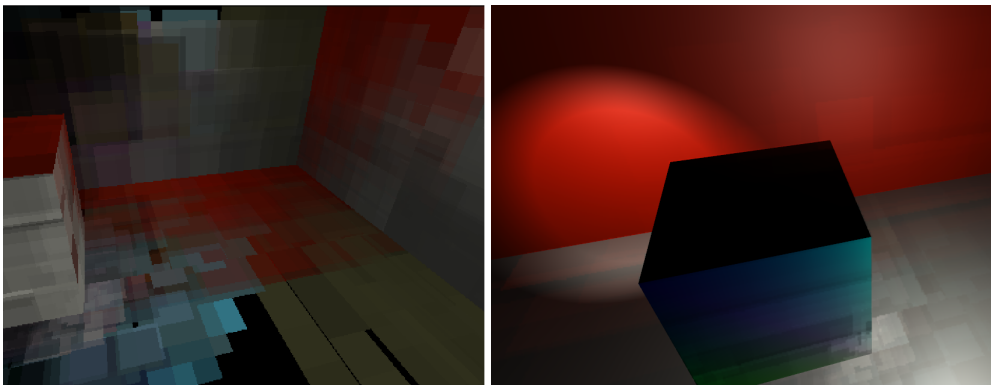

Abbildung 5.4 Linke Seite: Nur der indirekte Anteil der Szene gerendert. Rechte Seite: Wieder finales Bild der Szene, aber mit zusätzlichem Spot Light.

Für diese Szene wurden, wie anfangs erwähnt, 32 Voxel pro Achse und 3 LODs eingestellt. Zudem Betrug die Seitenlänge des Voxel-Volumens 32 Einheiten und die Voxelisierungs-Auflösung pro Achse das Achtfache der Voxel-Auflösung. Die Light Injection-Auflösung glich der Voxel-Auflösung und die Cone Tracing-Auflösung entsprach der halben Programmauflösung pro Achse. Die Programmauflösung ist die, in der die direkte Beleuchtung gerendert und das endgültige Fenster dargestellt wird. Wie viel Zeit die einzelnen Schritte der Implementierung mit diesen Einstellungen in Anspruch nahmen, ist Tabelle 5.1 zu entnehmen.

Zeit in ms				
Voxelisierung	Light Injection	Propagation	Cone Tracing	Finale Beleuchtung
0,071	0,345	0,026	1,380	0,141

Tabelle 5.1 Dauer der einzelnen Schritte der Implementierung in der selbst erstellten Testszene.

Man kann gut erkennen, dass in dieser Szene fast alle Schritte extrem schnell sind, nur das Cone Tracing sticht etwas heraus, obwohl es an sich auch nicht viel Zeit braucht. Dass dieser Schritt hier im Vergleich nicht ganz so schnell ist, liegt daran, dass er nahezu unabhängig von der Anzahl der Polygone ist und daher nicht von der simplen Szene profitiert. Alle weiteren Messergebnisse dieser Szene verwenden das hier dargestellte Resultat als Referenz.

Durch das Hinzufügen der 2. Lichtquelle, vgl. Abb. 5.4 rechts, steigt die Berechnungsdauer der finalen Beleuchtung, wobei sich dies durch die geringe Anzahl der Polygone in dieser Szene in Grenzen hält, und vor allem der des Light Injection-Schritts deutlich. Letzteres kann in

Tabelle 5.2 nachvollzogen werden. Auch konnte festgestellt werden, dass eine Verdoppelung der Light Injection-Auflösung pro Achse zu ungefähr einer Vervierfachung der entsprechenden Berechnungszeit führt, was durch die quadratische Abhängigkeit in Bezug auf die Auflösung, vgl. Komplexitätsanalyse in Abschnitt 4.3.5, zu erwarten war.

Light Inejction		
Veränderung	Zeit (ms)	Zuwachsfaktor
Doppelte Auflösung pro Achse	1,254	4,11
2. Lichtquelle	0,817	2,36

Tabelle 5.2 Dauer und Geschwindigkeitsverlust im Light Injection-Schritt durch eine feinere Abtastung bzw. eine zusätzliche Lichtquelle in der selbst erstellten Testszene.

Für die Auflösungsparameter der Light Injection- und Voxelisierungs-Schritte ist zu beachten, dass sie möglichst niedrig gewählt werden sollten, die jeweilige Renderauflösung also möglichst wenig größer als die Voxel-Auflösung selbst sein sollte. Nicht nur steigen die Berechnungskosten quadratisch mit ihr, wie in Tabelle 5.2 am Beispiel der Light Injection-Auflösung gezeigt, auch bricht die Geschwindigkeit bei hohen Werten szenenabhängig extrem ein. Ursache dafür ist, dass in diesem Fall viele Fragmente in dieselben Voxel, und somit auch dieselben Speicheradressen, schreiben müssen und durch die Synchronisation dann die Parallelität auf der Grafikkarte beeinträchtigt wird. Zumal ein höherer Parameterwert, sprich eine höhere Renderauflösung, in diesen Algorithmus-Schritten keinen Qualitätsgewinn bringt, sobald die Abtastung fein genug ist, damit alle Voxel im relevanten Bereich erfasst werden.

Verdoppelt man die Anzahl der Voxel pro Achse, führt das durch die unveränderte Größe des Voxel-Volumens zu feineren Voxeln, wie in Abb. 5.5 auf der linken Seite ersichtlich. Auch erkennt man, dass die Distanz, bis zu welcher die indirekte Beleuchtung erfasst wird, sinkt, was daran liegt, dass die Maximaldistanz während des Cone Tracings von der maximalen Voxel-Größe abhängt. Erhöht man die Anzahl der LODs auf 4 und vergrößert man zusätzlich zu der Anzahl der Voxel auch die Größe des Voxel-Volumens, so erhöht sich mit der Voxel-Größe auch wieder die Distanz der Darstellung der indirekten Beleuchtung, wie auf der rechten Seite von Abb. 5.5 illustriert wird.

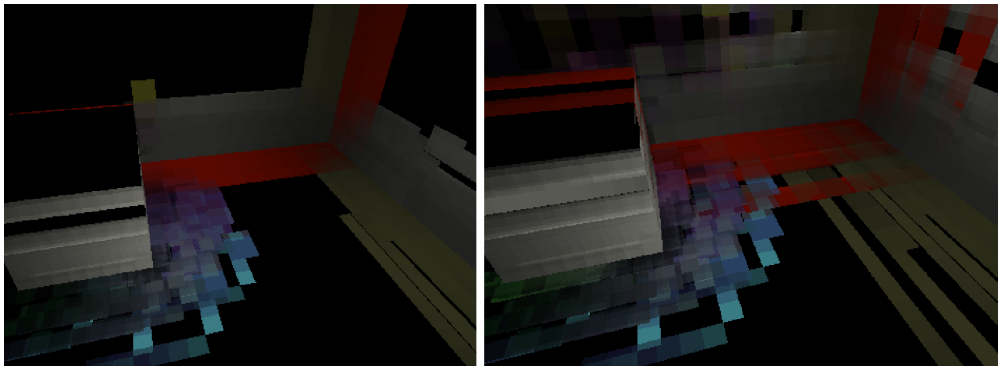

Abbildung 5.5 Linke Seite: Verdoppelte Voxel-Auflösung (64 pro Achse). Rechte Seite: Ebenfalls 64 Voxel pro Achse, aber zusätzlich verdoppelte Seitenlänge des Voxel-Volumens und 4 statt 3 LODs.

Eine Messung der Auswirkungen auf die Geschwindigkeit (vgl. Tabelle 5.3) zeigt, dass, wie laut Komplexitätsanalyse in Abschnitt 4.3.5 anzunehmen war, der Propagations-Schritt in etwa linear mit der Verachtfachung der Voxel skaliert und durch ein zusätzliches LOD die Zeit nochmals um ungefähr um 50% steigt. Vorher waren 2 Propagations-Iterationen nötig, nun 3. Da die Light Injection- und Voxelisations-Auflösungen von der Voxel-Auflösung abhängig sind, verwundert auch hier ein moderater Anstieg der Berechnungszeit nicht. Dass der Voxelisierungs-Schritt relativ gesehen stärker steigt, könnte daran liegen, dass auch sein Voxel-Auflösungs-Faktor von 8 deutlich höher ist. Dass jedoch die Dauer der Voxelisierung mit zusätzlichem LOD und größeren Voxeln in den groben LODs deutlich sinkt, ist verwunderlich. Der Anstieg der Cone Tracing-Zeit in diesem Fall dagegen ist leicht zu erklären. Durch das größere Voxel-Volumen und damit auch den größeren Voxeln steigt die Maximaldistanz des Cone Tracings, was die höheren Kosten verursacht.

	64 Voxel pro Achse		64 Voxel pro Achse, 4 LODs, doppelte Volumen-Seitenlänge	
Schritte	Zeit (ms)	Zuwachsfaktor	Zeit (ms)	Zuwachsfaktor
Voxelisierung	0,147	2,07	0,042	0,59
Light Injection	0,444	1,28	0,453	1,31
Propagation	0,253	9,73	0,381	14,65
Cone Tracing	1,506	1,09	3,652	2,64

Tabelle 5.3 Geschwindigkeitsverluste durch mehr Voxel und LODs in der selbst erstellten Testszene.

Leider zeigen die Abbildungen 5.3 und 5.4, dass auch nach Kombination von direkter und indirekter Beleuchtung zum finalen Bild deutliche Artefakte im Bereich der indirekten Beleuchtung ersichtlich sind, welche von der Voxel-Struktur herrühren. Wie bereits im letzten Absatz in Abschnitt 4.3.4 angemerkt, müsste der Algorithmus bzw. die Implementierung durch v.a. bessere Interpolation und Filterung während und nach des Cone Tracing-Schrittes erweitert werden, um diese Voxel-Muster zu beseitigen und saubere Beleuchtungsergebnisse zu liefern.

Als zweite Testszene dient die bekannte Cornell-Box (McGuire (2011)) mit ansonsten unveränderten Testbedingungen und Parametern (dazu siehe Absatz 1 und 5 des Kapitels). Abb. 5.6 illustriert das endgültige Bild, Abb. 5.7 nur den indirekten Anteil. Auch hier sind die indirekte diffuse Beleuchtung (man beachte den grünen und roten Schein auf Wänden, Quader und Boden), aber auch die zuvor erwähnten Artefakte erkennbar. Zudem fällt speziell in dieser Szene auf, dass hier anscheinend Probleme mit dem Cone Tracing bestehen, da v.a. Boden und Rückwand kaum diffuse Reflexionen der roten, linken Wand aufnehmen. Die Daten in der Voxel-Datenstruktur sind jedoch nach Begutachtung korrekt eingefügt worden, die vorigen Schritte des Algorithmus zeigen demnach keine Auffälligkeiten. Da dieses Verhalten ausschließlich in dieser Testszene auftritt, liegt der Verdacht nahe, dass die Ursache auf die Daten der Testszene zurückzuführen ist, welche für diesen Test zum Programmstart aus einer Datei geladen wurden.

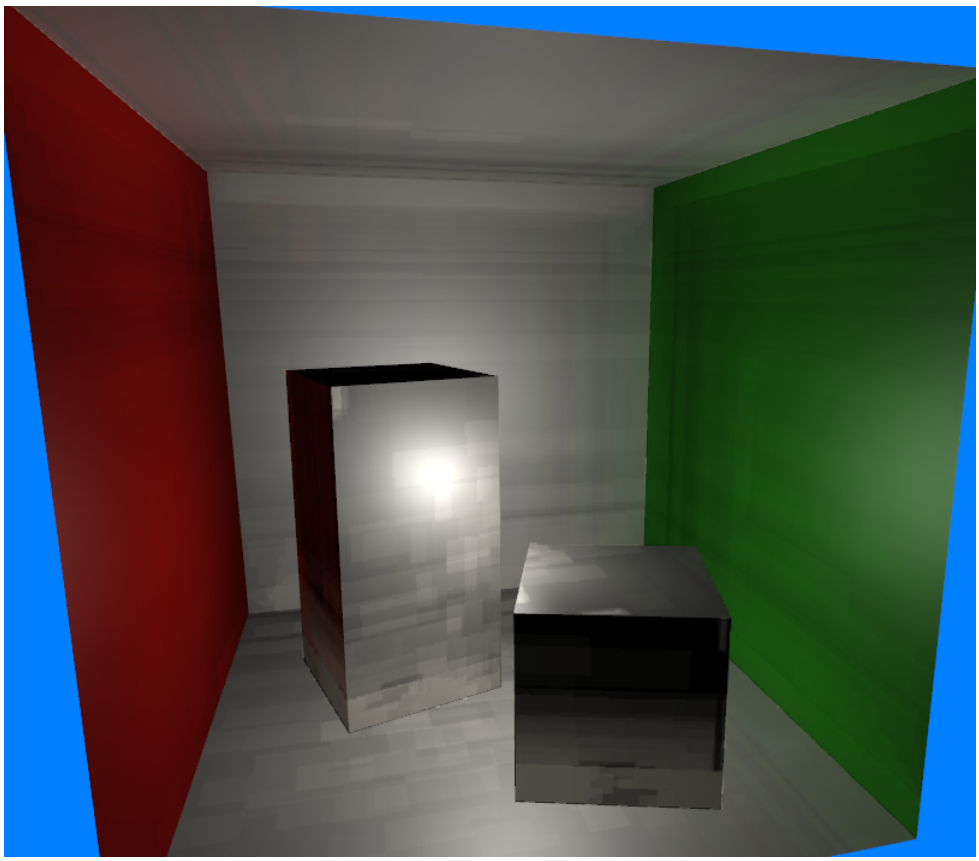

Abbildung 5.6 Cornell-Box mit direkter und indirekter Beleuchtung.

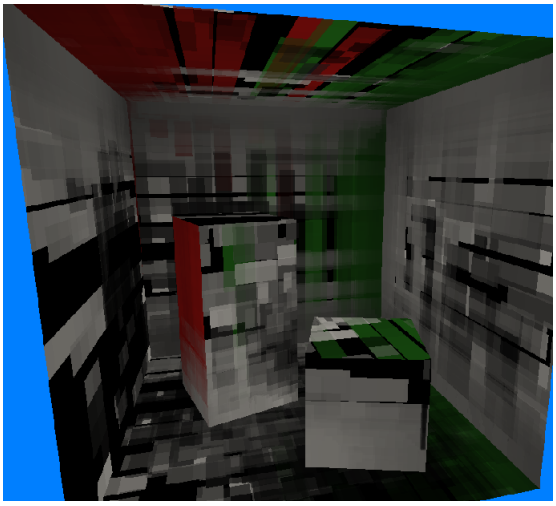

Abbildung 5.7 Cornell-Box mit indirekter Beleuchtung.

Tabelle 5.4 zeigt wieder die Berechnungszeit der einzelnen Schritte. Im Vergleich zu den Daten der ersten Szene, siehe Tabelle 5.1, fallen die deutlich höheren Zeiten der Schritte Light Injection und Cone Tracing auf. Ersteres liegt wahrscheinlich daran, dass hier mehrere große Flächen sehr nah an dem Point Light liegen und so für mehr Fragmente die Berechnung und v.a. synchronisierte Speicherung der Irradianz nötig ist.

Zeit in ms				
Voxelisierung	Light Inejction	Propagation	Cone Tracing	Finale Beleuchtung
0,019	2,438	0,026	3,116	0,104

Tabelle 5.4 Dauer der einzelnen Schritte der Implementierung in der Cornell-Box-Testszene.

Um zu vergleichen, wie sich verschiedene Cone Tracing Auflösungen auf das visuelle Ergebnis und die Geschwindigkeit auswirken, wurde die Szene je einmal mit Cone Tracing in voller Programmauflösung (1280 mal 720), der halben und der geviertelten Auflösung pro Achse gerendert. Abb. 5.8 zeigt, wie mit sinkender Auflösung (von oben nach unten) Aliasing-Artefakte an dem indirekten Beleuchtungsanteil steigen. Auch diese Artefakte ließen sich mit nachträglicher Filterung sowie Antialiasing auf das Zwischenergebnis und einer cleveren Interpolation der Cone Tracing- auf die Programmauflösung mindern. Dies würde deutliche Qualitätsgewinne bringen.

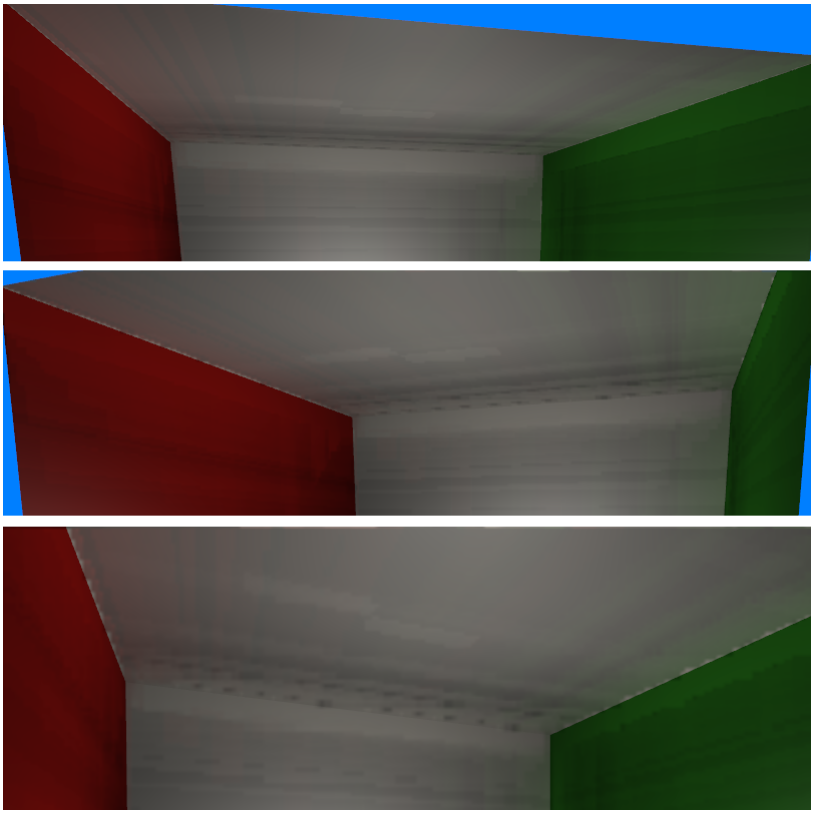

Abbildung 5.8 Ausschnitt der Cornell-Box mit verschiedenen Cone Tracing-Auflösungen. Oben: Volle CT-Auflösung. Mitte: Halbe CT-Auflösung pro Achse. Unten: Viertel CT-Auflösung pro Achse.

In Tabelle 5.5 werden die Geschwindigkeitsmessungen des Vergleichs illustriert. Im Idealfall würde man durch die quadratische Abhängigkeit zu der Auflösung pro Achse eine Reduktion auf ein Viertel und Sechszehntel sehen. Praktisch konnte eine Reduktion auf 35% und 10% nachgewiesen werden, was immer noch einen erheblichen Geschwindigkeitsgewinn bedeutet.

Cone Tracing		
Auflösung pro Achse	Zeit (ms)	Zuwachsfaktor
Voll	8,987	-
Halb	3,116	0,346
Viertel	0,922	0,102

Tabelle 5.5 Geschwindigkeitsvergleich des Cone Tracing-Schritts in der Cornell-Box-Testszene mit verschiedenen Cone Tracing-Auflösungen.

Schließlich wurde noch eine dritte komplexere Testszene mit deutlich mehr Polygonen (262.267) getestet. Dabei handelt es sich um die sogenannte Sponza-Szene (McGuire (2011)) ohne Texturen. Die Oberflächenfarben wurden aus der Material-Datei übernommen. Abb. 5.9 stellt das endgültige Bild der Szene dar, Abb. 5.10 nur den indirekten Teil der Beleuchtung. Der Eindruck der vorigen Tests setzt sich auch in dieser Szene fort, die Approximation der indirekten Beleuchtung ist klar ersichtlich und ihre Darstellung folgt augenscheinlich den physikalischen Gesetzen. Auch die Voxel-Artefakte des Cone Tracings und deren mangelnde Filterung fallen wieder auf.

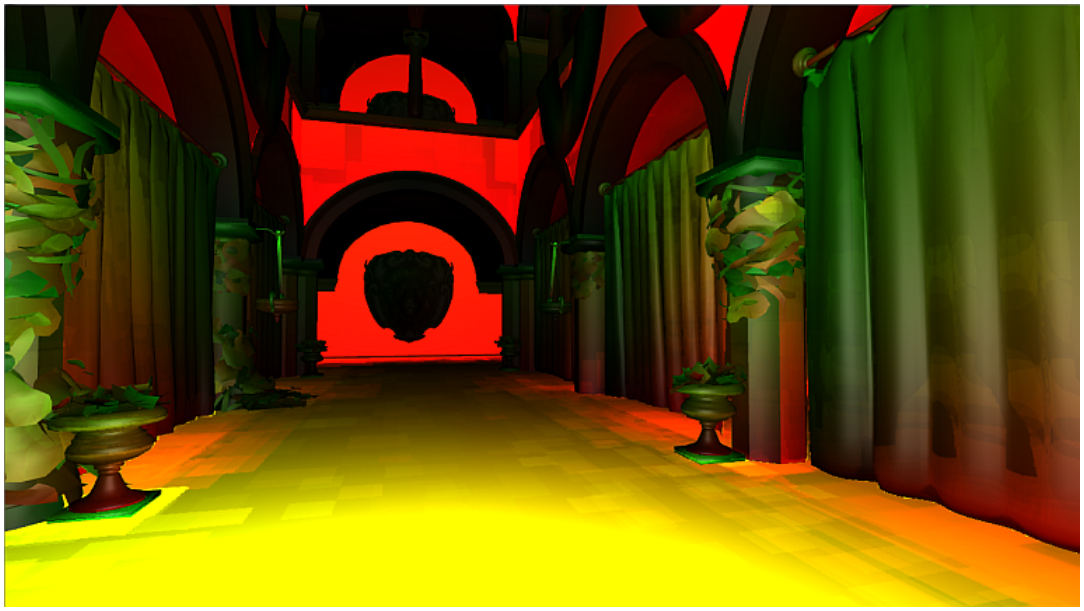

Abbildung 5.9 Sponza-Szene mit direkter und indirekter Beleuchtung gerendert.

Für diese Szene konnte der Zeitbedarf teilweise nur angenähert werden, was begründet wird durch die Kombination der Menge der Polygone und dem verwendeten Messverfahren. Erwartungsgemäß fallen nach Tabelle 5.6 die Berechnungskosten durch die Bank weg höher aus. Ausgenommen der Propagations-Schritt, der durch seine ausschließliche Abhängigkeit bezüglich der Zahl der Voxel wie in den anderen Szenen vernachlässigbar ist. Der Schritt der finalen Beleuchtung dauert relativ gesehen zwar deutlich länger als in den simpleren Szenen, aber ist insgesamt dennoch vergleichsweise schnell. Das Einfügen der Beleuchtung in die Voxel ist hier sogar etwas schneller als in der Cornell-Box-Szene, was dadurch begründet ist, dass hier die Distanz von der Lichtquelle zu den meisten Flächen etwas höher ist. Der Schritt der Voxelisierung ist in dieser Szene erstmals vergleichsweise teuer, was in Anbetracht der vielen Polygone abzusehen war.

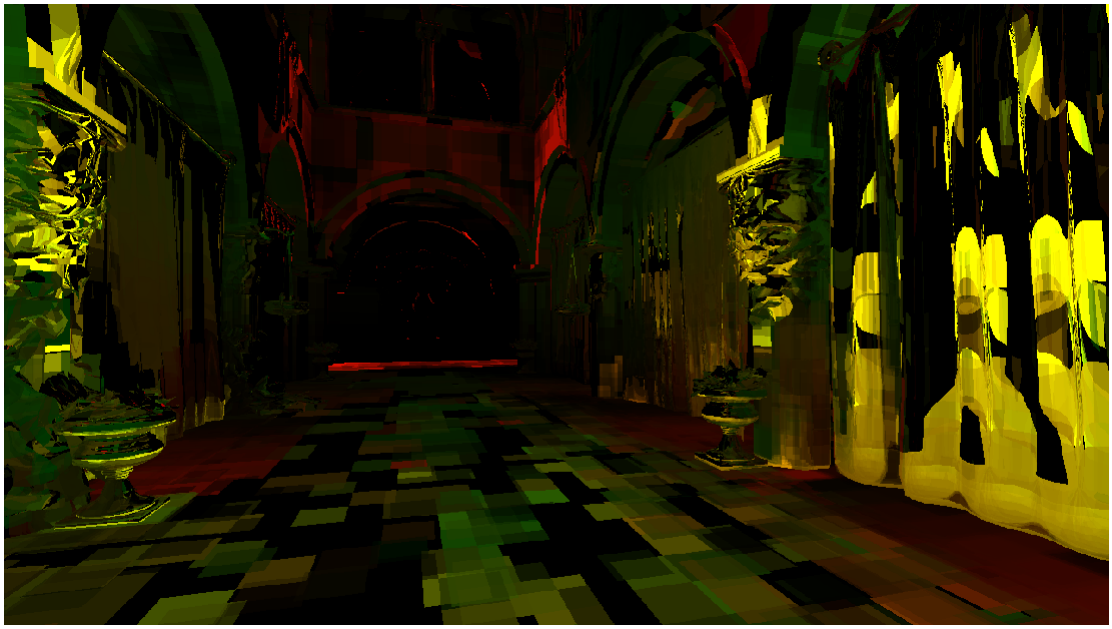

Abbildung 5.10 Sponza-Szene ausschließlich mit indirekter Beleuchtung gerendert.

Zeit in ms				
Voxelisierung	Light Inejction	Propagation	Cone Tracing	Finale Beleuchtung
≈ 2,200	≈ 2	0,03	≈ 4,5	≈ 0,5

Tabelle 5.6 Dauer der einzelnen Schritte der Implementierung in der Sponza-Testszene.

Abb. 5.11 stellt wie schon bei der Cornell-Box-Szene einen Vergleich über verschiedene Cone Tracing-Auflösungen dar, links mit direkter und indirekter, rechts nur mit indirekter Beleuchtung. Man erkennt auf den unteren Ausschnitten die geringere Auflösung (v.a. rechts) und wie sich diese mit der verwendeten simplen bilinearen Interpolation durch eine unschärfere indirekte Beleuchtung im endgültigen Bild (links) bemerkbar macht.

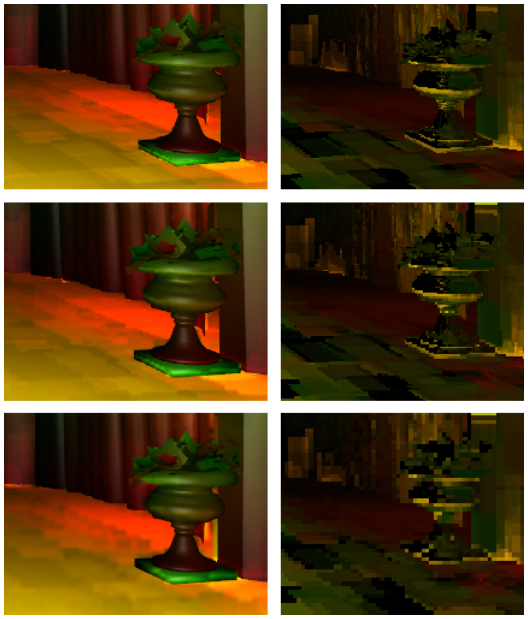

Abbildung 5.11 Ausschnitt der Sponza-Szene rechts mit indirekter, links mit direkter und indirekter Beleuchtung. Obere Reihe mit voller CT-Auflösung. Mittlere Reihe mit halber CT-Auflösung pro Achse. Untere Reihe mit viertel CT-Auflösung pro Achse.

Auch in dieser Szene wird durch die Reduktion der Cone Tracing-Auflösung zwar nicht der rechnerisch optimale Geschwindigkeitszuwachs erzielt, aber mit 31% und 13% der Ursprungszeit wieder ein sehr hoher, siehe Tabelle 5.7.

Cone Tracing		
Auflösung pro Achse	Zeit (ms)	Zuwachsfaktor
Voll	≈ 14,9	-
Halb	≈ 4,5	0,307
Viertel	≈1,9	0,127

Tabelle 5.7 Geschwindigkeitsvergleich des Cone Tracing-Schritts in der Sponza-Testszene mit verschiedenen Cone Tracing-Auflösungen.

Abschließend zeigt Tabelle 5.8 noch einmal eine Verdopplung der Anzahl der Voxel pro Achse. Der Zuwachs der jeweiligen Rechenzeit ist grob vergleichbar mit denen im analogen Test in der ersten Testszene, vgl. Tabelle 5.3. Entsprechend gelten dieselben Erläuterungen.

	64 Voxel pro Achse	
Schritte	Zeit (ms)	Zuwachsfaktor
Voxelisierung	≈ 4,3	≈ 1,95
Light Injection	≈ 3,1	≈ 1,55
Propagation	0,264	8,80

Tabelle 5.8 Geschwindigkeitsverlust durch mehr Voxel in der Sponza-Testszene.

Nach Begutachtung aller Messergebnisse in den drei Testszenen ist eindeutig, dass die Geschwindigkeit dieser Implementierung von Voxel Cone Tracing mühelos im Echtzeit-Bereich vorzufinden ist. Das Diagramm in Abb. 5.12 fasst noch einmal die Rechenzeiten der einzelnen Schritte des Algorithmus über alle Testszenen hinweg zusammen.

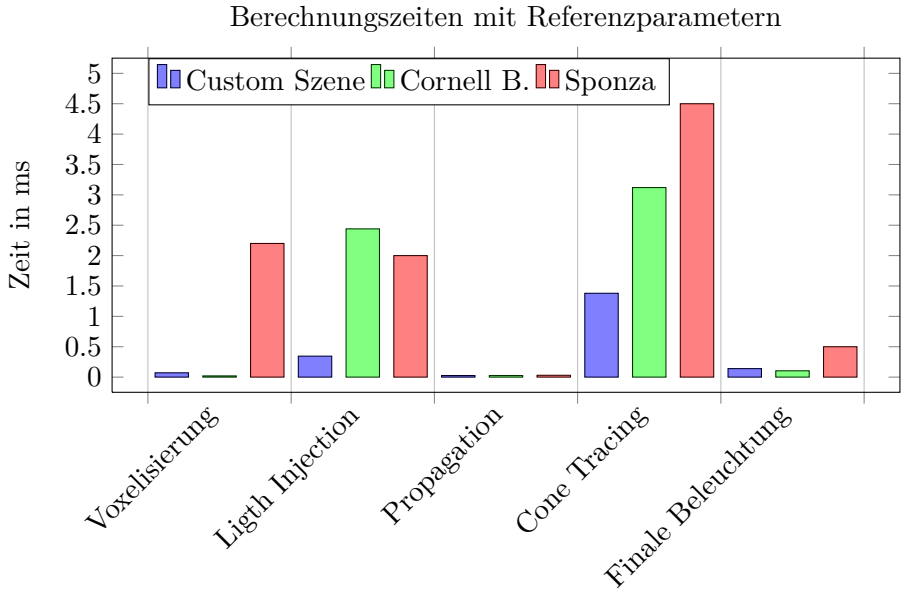

Abbildung 5.12 Vergleich der Berechnungszeiten der Algorithmus-Schritte in den drei Testszenen. Jeweils in den Referenzeinstellungen, Werte siehe Tabellen 5.1, 5.4, 5.6.

Die letztendliche Geschwindigkeit hängt maßgeblich von den Parametern ab, so kann z.B. durch Reduktion der Cone Tracing-Auflösung deutlich Rechenzeit eingespart werden, wie das Diagramm in Abb. 5.13 illustriert. Dies geht zwar mit einer Reduktion der Präzision und in diesem Fall der Bildqualität einher, jedoch ließe sich dies durch Erweiterungen des Algorithmus bzw. der Implementierung um bessere Interpolation und Filterung maskieren, wie bereits am Ende des Abschnitts 4.3.4 und weiter oben bei dem Cone Tracing-Auflösungsvergleich anhand der Cornell-Box-Szene geschrieben wurde.

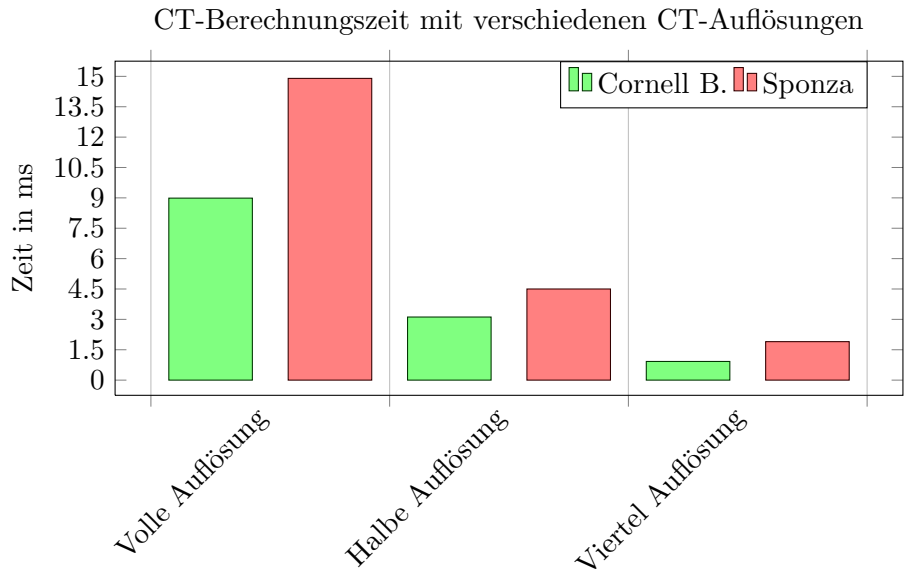

Abbildung 5.13 Vergleich der Berechnungszeiten des Cone Tracing-Schritts mit unterschiedlichen Auflösungen. Werte siehe Tabellen 5.5, 5.7.

In dem Diagramm in Abb. 5.14 sind die Auswirkungen eines weiteren wichtigen Parameters auf die Geschwindigkeit abgebildet. Eine doppelt so hohe Voxel-Auflösung pro Achse führt zu moderaten Geschwindigkeitsverlusten. Der Propagations-Schritt ist um einen beachtlichen Faktor 10 langsamer, jedoch ist der Schritt absolut gesehen extrem schnell, weswegen der Verlust insgesamt nicht groß ins Gewicht fällt.

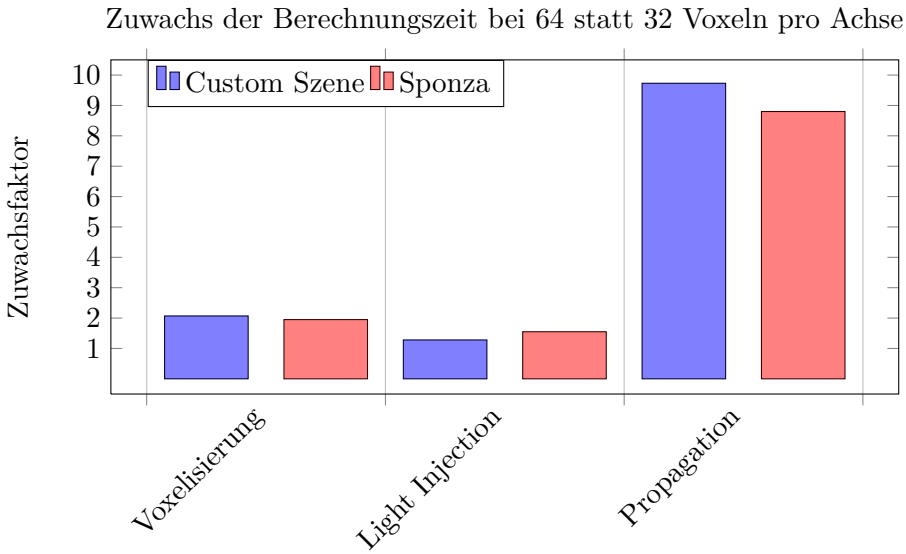

Abbildung 5.14 Übersicht über den Anstieg der Berechnungszeiten einiger Algorithmus-Schritte bei Erhöhung der Voxel-Auflösung von 32 auf 64 (pro Achse). Werte siehe Tabellen 5.3, 5.8.

Der Speicherbedarf hängt auch maßgeblich von der Voxel-Auflösung ab. Die endgültigen Kosten bezüglich des GPU-Speichers betragen bei einer Voxel-Auflösung von 32 pro Achse und 3 LODs lediglich 3,46 MB. Nach der Komplexitätsanalyse in Abschnitt 4.3.5 steigen sie jedoch kubisch mit der Voxel-Anzahl und linear mit den LODs. Eine Messung bestätigt dies, so hat man mit einer Voxel-Auflösung von 64 bereits einen Verbrauch von 27 MB und mit 128 Voxeln 216 MB. Eine weitere LOD-Stufe bei einer Voxel-Auflösung von 64 lässt den Speicherbedarf lediglich von 27 auf 36 MB steigen.

Ein wichtiger, bisher unerwähnter Punkt ist, dass in den Sponza und Cornell-Box-Szenen jedes Mesh einzeln durch den Algorithmus gelaufen ist, also alle Meshes eigene Drawcalls hatten. Dadurch wird sichergestellt dass auch, wenn sich alle Meshes dynamisch bewegen würden, kein Geschwindigkeitsverlust im Vergleich zu den hier dargelegten Messungen auftreten würde. Somit gelten die Ergebnisse auch für voll-dynamische Szenen.

Bei Bewegung der Kamera und aktivierter Aktualisierung des Voxel-Volumens treten leider zusätzliche Artefakte auf, die durch die bereits im letzten Absatz von Abschnitt 4.3.4 erwähnten Verbesserungsmöglichkeiten des Algorithmus und der Implementierung reduziert werden könnten. Auch temporales Antialiasing wäre eine große Hilfe, um die Artefakte zu bekämpfen.

5.2 Vergleich mit anderen Implementierungen

Nach der Auswertung der selbstgeschriebenen Implementierung und ihrer Parameter im letzten Abschnitt soll in diesem ein kurzer Vergleich zu den in Kapitel 3 vorgestellten, bereits existierenden Implementierungen von Crassin et al. und NVIDIA gezogen werden. Mangels verfügbarem Quellcode und den ohnehin unterschiedlichen Grafik-APIs und Rendering-Engines ist ein direkter, aussagekräftiger Vergleich nur schwer möglich. Der hier gezogene Vergleich ist demnach nur als grober Anhaltspunkt zu betrachten.

Abb. 5.15 zeigt die Sponza Szene, hier wohlgemerkt mit Texturen, beleuchtet durch die Voxel Cone Tracing-Implementierung von Crassin et al. Die diffuse, indirekt Beleuchtung ist gut an den Bögen oberhalb der Vorhänge zu erkennen. Im Vergleich zu den Ergebnissen der selbstgeschriebenen Implementierung ist die indirekte Beleuchtung deutlich besser, da augenscheinlich frei von Artefakten. Was die Geschwindigkeit angeht, so spricht Crassin et al. (2011) bei einer Auflösung von 512 mal 512 von 30 FPS auf einer NVIDIA GeForce GTX 480. Bei 1024 mal 768 seien es 11 FPS. Eingestellt wurden drei diffuse Cones und ein 9-Level Octree der Auflösung 512^3. Bei der Auflösung von 512 mal 512 soll der Light Injection-Schritt 16 ms dauern und nur bei Bedarf ausgeführt werden. Das dynamische Update der Voxelisierung hängt von der Dynamik der Szene ab, beträgt aber mehrere Millisekunden. Die Berechnung der indirekt diffusen Beleuchtung dauert 7,8 ms pro Frame. Ein Vergleich mit der

von mir geschriebenen Implementierung gestaltet sich aufgrund der komplett verschiedenen Bedingungen schwierig. Crassins Version ist absolut gesehen deutlich langsamer, die dort verwendete Grafikkarte aber auch deutlich älter und schwächer.

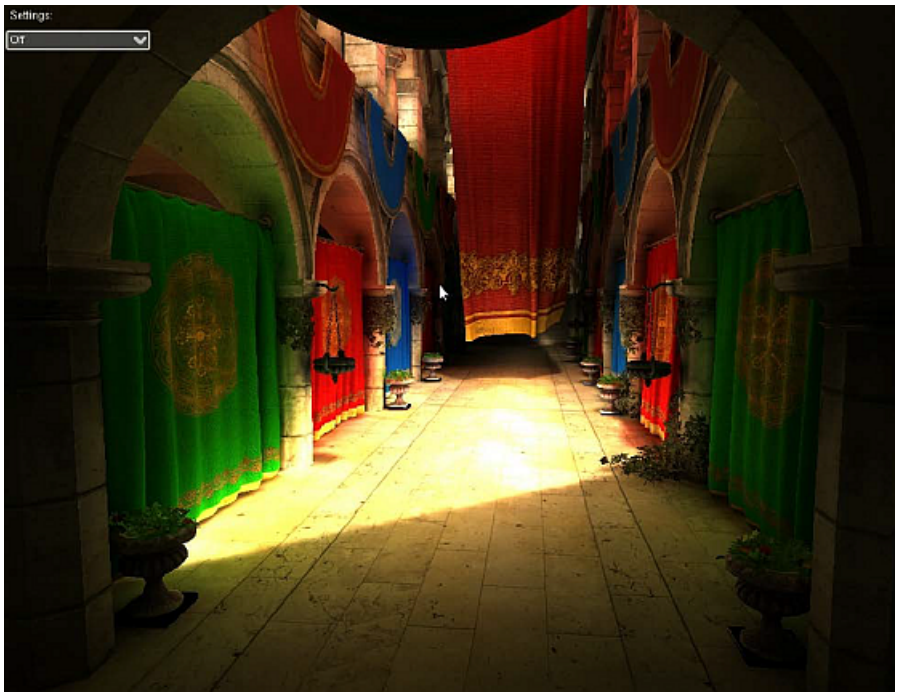

Abbildung 5.15 Sponza Szene mit Texturen mittels Voxel Cone Tracing von Crassin et al. Crassin (2012)

Wie NVIDIAs Voxel Cone Tracing-Implementierung die Sponza Szene darstellt, ist in Abb. 5.16 dargestellt. Auch hier erkennt man die qualitativ bessere indirekte Beleuchtung gut, v.a. wieder an den Bögen und Säulen in der oberen Etage. Im Vergleich zu der Version von Crassin et al. lässt sich jedoch augenscheinlich kein gravierender Unterschied feststellen. Bezüglich der Geschwindigkeit wird von NVIDIA mit einer NVIDIA GeForce GTX 770 und einer Auflösung von 1920 mal 1080 eine Berechnungszeit für die globale Beleuchtung von 7,4 ms für mittlere und 12,9 ms für hohe Qualitätseinstellungen angegeben. Für die hohen Einstellungen würden 17 Cones verwendet, aber die Cone Tracing-Auflösung nur $\frac{1}{9}$ der Programmauflösung pro Achse betragen. Dabei nimmt der Cone Tracing-Anteil 59% der Gesamtzeit in Anspruch. Im Vergleich wirkt NVIDIAs Version trotz der stärkeren Grafikkarte merklich performanter als die Version von Crassin et al. Ein Vergleich mit meiner Implementierung ist wieder schwierig, jedoch könnte man sagen, dass die beiden Versionen grob in derselben Geschwindigkeitsklasse liegen.

Abbildung 5.16 Sponza Szene mit Texturen mittels Voxel Cone Tracing von NVIDIA. Panteleev (2014)

In Tabelle 5.9 werden noch einmal die Implementierungen mit den genannten Parametern und Berechnungszeiten gegenübergestellt.

Sponza Szene			
Implementierung	Crassin	NVIDIA	Eigene
Grafikkarte	GTX 480	GTX 770	GTX 1060
Auflösung	512x512	1920x1080	1280x720
Cones pro Pixel	3	17	9
CT-Auflösung pro Achse	$\frac{1}{1}$	$\frac{1}{9}$	$\frac{1}{4}$
Zeit in ms	≈ 33	12.9	$\approx 6{,}8$

Tabelle 5.9 Vergleich der Implementierungen.

Abschließend lässt sich sagen, dass die bestehenden Implementierungen eine deutlich höhere Qualität erreichen, da die selbstgeschriebene Implementierung, wie in Kapitel 5.1 betrachtet, an deutlicher Artefaktbildung leidet. Bezüglich der Geschwindigkeit muss sie sich, soweit man das sagen kann, nicht verstecken, es bleibt sogar Spielraum für Qualitätsverbesserungen, zumal sie bei hoher Dynamik in den Szenen nicht zusätzlich einbrechen würde.

Kapitel 6

Fazit und Ausblick

Ziel dieser Arbeit war es, aktuelle Verfahren zur Berechnung von globaler Beleuchtung in Echtzeit zu betrachten und ein auf Voxeln basierendes - Voxel Cone Tracing - durch eine eigene Implementierung genauer zu untersuchen. Es sollte geprüft werden, inwiefern mit, bzw. trotz einiger Approximationen ein plausibel aussehendes Ergebnis in Echtzeit erreicht werden kann und ob hierfür, auch speziell in voll-dynamischen Szenen, Voxel Cone Tracing gut geeignet ist. Es konnte im Laufe der Arbeit gezeigt werden, dass es für dieses Problem viele mögliche Verfahren gibt, die erwartungsgemäß alle Vor- und Nachteile besitzen und leider auch diversen Einschränkungen unterliegen. Auch wurde durch die eigene Implementierung bestätigt, dass Voxel Cone Tracing im Allgemeinen den Anforderungen - qualitativ ansprechende globale Beleuchtung, Berechnung in Echtzeit-Geschwindigkeit und potentiell voll-dynamische Szenen - absolut gerecht wird und somit eine gute Lösung für Echtzeitanwendungen mit entsprechendem Ansprüchen ist. Vor allem durch die vielen einstellbaren Parameter, wie z.B. der Voxel- oder Cone Tracing-Auflösung und der Anzahl der LODs, gibt es effektive Stellschrauben, die es ermöglichen, einen optimalen Kompromiss aus Geschwindigkeit und Qualität bzw. Präzision zu erreichen. Durch meine Implementierung in OpenGL ist zudem sichergestellt, dass eine breite Nutzerbasis, über verschiedene Plattformen verteilt, die Möglichkeit erhält, diese und auch allgemein vergleichbare Implementierungen von Voxel Cone Tracing zu nutzen. Im Vergleich mit den in dieser Arbeit ebenfalls vorgestellten, bestehenden Implementierungen von Voxel Cone Tracing muss sich diese Implementierung nach einem Vergleich leider geschlagen geben, was die Qualität der globalen Beleuchtung bei der gegebenen Geschwindigkeit angeht. In Anbetracht der eingesetzten Zeit und Ressourcen war dies allerdings zu erwarten. Anzumerken ist jedoch auch, dass diese Implementierung im Gegensatz zu den anderen durch OpenGL, wie oben erwähnt, plattformunabhängig ist und keinen Geschwindigkeitseinbruch bei voll-dynamischen Szenen erleidet.

Die im Zuge dieser Arbeit entstandene Implementierung ist mangels des knappen Zeitrahmens deutlich ausbaufähig, es gibt viel Verbesserungspotential und Raum für Ergänzungen. So dient sie mehr als eine Art Basisversion, als eines vollständigen Systems.

Die vielversprechenden Verbesserungen, die es bedauerlicherweise nicht mehr in diese Arbeit geschafft haben, wären eine clevere Interpolation und Filterung der indirekten Beleuchtung, um die Artefakte und Muster, bedingt durch die Voxel-Struktur, zu maskieren und eine anisotropische Voxelisierung um das Mitteln von Geometriedaten unterschiedlicher Seiten zu verhindern und dadurch die Präzision zu erhöhen. Beides wäre mit relativ wenig Mehraufwand direkt in den Algorithmus integrierbar. Eine naheliegende Erweiterung der Implementierung wäre ein zusätzlicher Cone-Trace, um glänzende Reflexion darzustellen, wie es die bestehenden Implementierungen tun. Die Zusatzkosten würden sich in Grenzen halten. Weitere potentielle Verbesserungen, die es zu prüfen lohnen würde, wären z.B. Conservative Rasterization oder Antialiasing zwecks präziserer Voxelisierung, wie es Crassin et al. und NVIDIA umgesetzt haben, oder das Berechnen von mehr als einer indirekten Reflexion, ähnlich der Implementierung von Q-Games.

Literaturverzeichnis

(2016). oflight - openframework dokumentation. http://openframeworks.cc/documentation/gl/ofLight/, Abgerufen am: 02.06.2016.

Ahmed, S. (2009). Ray tracing acceleration data structures. Lecture Notes Computer Graphics, Indian Institute of Technology Bombay. https://www.cse.iitb.ac.in/~paragc/teaching/2009/cs475/notes/accelerating_raytracing_sumair.pdf, Abgerufen am: 06.12.2016.

Asirvatham, A. and Hoppe, H. (2005). Terrain rendering using gpu-based geometry clipmaps. In Pharr, M., editor, *GPU Gems 2*, pages 27–45. Addison-Wesley. http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html, Abgerufen am: 06.12.2016.

Crassin, C. (2012). Octree-based sparse voxelization for real-time global illumination. GPU Technology Conference. http://www.icare3d.org/research/GTC2012_Voxelization_public.pdf, Abgerufen am: 02.06.2016.

Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. (2011). Interactive indirect illumination using voxel-based cone tracing: An insight. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, pages 20:1–20:1. ACM. http://maverick.inria.fr/Publications/2011/CNSGE11b/GIVoxels-pg2011-authors.pdf, Abgerufen am: 06.12.2016.

Ertürk, U. R. (2008). The clipmap: A virtual mipmap. Master's thesis, University of Abertay Dundee. http://www.hevi.info/dissertation-ray-tracing/MSc_Dissertation_Umut_Riza_ERTURK_Ray_Tracing_On_Cell_by_Using_KD_Tree.html, Abgerufen am: 05.06.2016.

Gebhardt, N. (2003). Einige brdf modelle. http://www.irrlicht3d.org/papers/BrdfModelle.pdf, Abgerufen am: 02.06.2016.

Goswami, P. (2012). *Level-of-Detail and Parallel Solutions in Computer Graphics*. PhD thesis, University of Zurich. http://www.zora.uzh.ch/62018/1/Goswami.pdf, Abgerufen am: 06.12.2016.

Hapala, M., Karlik, O., and Havran, V. (2011). When it makes sense to use uniform grids for ray tracing. WSCG '11, pages 193–200. Vaclav Skala - UNION Agency. http://dcgi.felk.cvut.cz/home/havran/ARTICLES/HapalaKarlikHavran2011wscg.pdf, Abgerufen am: 15.02.2017.

Hughes, J. F. et al. (2013). *Computer Graphics: Principles and Practice*. Addison-Wesley Professional, 3 edition.

Kajiya, J. T. (1986). The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150. http://www.cse.chalmers.se/edu/year/2011/course/TDA361/2007/rend_eq.pdf, Abgerufen am: 06.12.2016.

Kaufman, A., Cohen, D., and Yagel, R. (1993). Volume graphics. *Computer*, 26(7):51–64. https://cvc.cs.stonybrook.edu/Publications/1993/KCY93/file.pdf, Abgerufen am: 06.12.2016.

Křivánek, J., Ferwerda, J. A., and Bala, K. (2010). Effects of global illumination approximations on material appearance. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, pages 112:1–112:10. ACM. https://www.cs.cornell.edu/~kb/publications/SIG10VPLPerception.pdf, Abgerufen am: 06.12.2016.

Mavridis, P., Gaitatzes, A., and Papaioannou, G. (2010). Volume-based diffuse glocal illumination. In *Proc. of Computer Graphics, Visualization, Computer Vision and Image Processing 2010*, pages XX–XX. http://www.pmavridis.com/data/VolumeBasedDiffuseGlobalIllumination_final.pdf, Abgerufen am: 06.12.2016.

McGuire (2011). Computer graphics archive. http://graphics.cs.williams.edu/data, Abgerufen am: 20.02.2017.

McLaren, J. (2015). The technology of the tomorrow children. Game Developer Conference. http://fumufumu.q-games.com/archives/TheTechnologyOfTomorrowsChildrenFinal.pdf, Abgerufen am: 06.12.2016.

Oren, M. and Nayar, S. K. (1995). Generalization of the lambertian model and implications for machine vision. *International Journal of Computer Vision*, 14(3). http://www1.cs.columbia.edu/CAVE/publications/pdfs/Nayar_IJCV95.pdf, Abgerufen am: 02.06.2016.

Panteleev, A. (2014). Practical real-time voxel-based global illumiantion for current gpus. GPU Technology Conference. http://on-demand.gputechconf.com/siggraph/2014/presentation/SG4114-Practical-Real-Time-Voxel-Based-Global-Illumination-Current-GPUs.pdf, http://on-demand.gputechconf.com/gtc/2014/presentations/S4552-rt-voxel-based-global-illumination-gpus.pdf, Abgerufen am: 06.12.2016.

Ritschel, T., Dachsbacher, C., Grosch, T., and Kautz, J. (2012). The state of the art in interactive global illumination. *Computer Graphics Forum*, 31(1):160–188. https://people.mpi-inf.mpg.de/~ritschel/Papers/GISTAR.pdf, Abgerufen am: 05.06.2016.

Samet, H. and Webber, R. E. (1988). Hierarchical data structures and algorithms for computer graphics. part i. *IEEE Comput. Graph. Appl.*, 8(3):48–68. http://www.cs.umd.edu/~hjs/pubs/SameCGA88b.pdf, Abgerufen am: 06.12.2016.

Tanner, C. C., Migdal, C. J., and Jones, M. T. (1998). The clipmap: A virtual mipmap. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 151–158. ACM. http://www.cs.virginia.edu/~gfx/courses/2002/BigData/papers/Texturing/Clipmap.pdf, Abgerufen am: 02.06.2016.

van Oosten, J. (2014). Texturing and lighting with opengl and glsl. http://www.3dgep.com/texturing-and-lighting-with-opengl-and-glsl/#The_Phong_and_Blinn-Phong_Lighting_Models, Abgerufen am: 05.06.2016.

Veach, E. and Guibas, L. J. (1997). Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76. ACM Press/Addison-Wesley Publishing Co. https://graphics.stanford.edu/papers/metro/metro.pdf, Abgerufen am: 06.12.2016.

Vrajitoru, D. (2011). Computer graphics. Lecture at Indiana University South Bend. http://www.cs.iusb.edu/~danav/teach/c481/c481_15_raytrace.html, Abgerufen am: 06.12.2016.

Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349. https://pdfs.semanticscholar.org/b1d7/6a254801a09916479659160fd839c905ae87.pdf, Abgerufen am: 06.12.2016.

Zachmann, G. (2014). Computergraphik i. Vorlesungsfolien, Uni Bremen.

Zachmann, G. (2015). Advanced computer graphics. Vorlesungsfolien, Uni Bremen.