Bachelorarbeit

Lars Nielsen

Realtime Radiosity mit Nvidia CUDA

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Lars Nielsen

Realtime Radiosity mit Nvidia CUDA

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Jenke
Zweitgutachter: Prof. Dr. Schmolitzky

Eingereicht am: 27. März 2018

Lars Nielsen

Thema der Arbeit
Realtime Radiosity mit Nvidia CUDA

Stichworte
Globale Beleuchtung, Radiosity, Nvidia CUDA, GPU

Kurzzusammenfassung
Diese Arbeit beschreibt die Entwicklung eines Prototyps des Radiosity-Verfahrens zur globalen Beleuchtungsberechnung in dreidimensionalen Szenen. Das Verfahren wurde zunächst als Software-Implementation umgesetzt. Später wurden ausgewählte Teile des Algorithmus mit Hardwareunterstützung umgesetzt, sodass der Algorithmus die massive, parallele Rechenkraft gängiger Grafikhardware ausnutzen kann. Dadurch sind zwei Varianten entstanden, die in Bezug auf ihre Laufzeit miteinander verglichen wurden.
Da bereits viel Zeit in die Entwicklung der Software-Implementation investiert wurde, wurde der Grad der Hardwareunterstützung auf das Geringste beschränkt. Dennoch konnte eine signifikante Beschleunigung der Laufzeit erreicht werden. Unter diesem Gesichtspunkt wird auch darauf eingegangen, welche Möglichkeiten sich weiterhin bieten die Hardwareunterstützung der entstandenen Implementation zu verbessern.
Lars Nielsen

Title of the paper
Realtime Radiosity with Nvidia CUDA

Keywords
global illumination, radiosity, Nvidia CUDA, GPU

Abstract
This work covers the development of a prototype of the radiosity algorithm for the global illumination problem in three dimensional scenes. First the algorithm was implemented as a software solution. Later chosen parts were implemented with hardware acceleration so that the algorithm can take advantage of the massive parallel computing capabilities of current graphics processing hardware. That way two variants were developed that were compared with each other regarding their runtime performance.
A lot of time has been invested into the development of the software solution. Therefore the support for hardware acceleration had to be reduced to the absolute minimum. Still

significant runtime acceleration has been achieved. In respond to the limited hardware support this work will feature opportunities to further enhanche the support for hardware acceleration.

Inhaltsverzeichnis

1 Einleitung

Die Computergrafik beschäftigt sich unter anderem mit der Darstellung von dreidimensionalen Szenen. Eine besondere Herausforderung dabei ist die Bildsynthese (auch Rendering genannt) bei der ein zweidimensionales Bild aus den dreidimensionalen Rohdaten der Szene erzeugt wird. Dafür müssen Ansichtsparameter festgelegt werden die durch eine virtuelle Kamera repräsentiert sind und anschließend der sichtbare Teil einer Umgebung auf die „Linse“ der Kamera (View-Plane) projiziert werden. Koordinaten liegen dabei in unterschiedlichen Räumen vor die während der Projektion transformiert werden müssen. Ein Betrachtungssystem transformiert die Koordinaten von Objekten dabei aus dem Model-Space über den World-Space in den View-Space der Kamera und für die Erzeugung eines Bildes in den Image-Space.

Das Radiosity-Verfahren wird in der Computergrafik eingesetzt um die Beleuchtungsrechnung in dreidimensionalen Szenen unter Berücksichtigung der globalen Wechselwirkungen von diffusen Oberflächen zu lösen. Es wurde 1984 von Goral et al. beschrieben und unterliegt in seiner Reinform einem Laufzeitaufwand von $O(n^2)$ [16]. Eine als Progressive Refinement Radiosity bekannte Erweiterung des Verfahrens ermöglicht die Erzeugung von Teillösungen die einem Anwender präsentiert werden können, während die Berechnung voranschreitet. Obgleich der Laufzeitaufwand von $O(n^2)$ beibehalten wird [9], ermöglicht die Erzeugung von Teillösungen eine Umgebung bereits zur Berechnungszeit interaktiv zu betrachten.

1.1 Relevanz der Arbeit

Das Radiosity-Verfahren arbeitet im World-Space und ist dadurch blickwinkelunabhängig, wodurch die Berechnung beim Wechseln der Perspektive nicht erneut durchgeführt werden muss. Das macht es für Anwendungen interessant in denen häufige Perspektivwechsel üblich sind. Darunter fallen zum Beispiel Architekturdesign, virtuelle Kamerafahrten, Computerspiele oder Virtual-Reality Anwendungen.

Für wirklich interaktive Verhältnisse war die damalige Hardware nicht leistungsstark genug. Seitdem das Verfahren 1984 erstmals beschrieben wurde, sind bis heute 34 Jahre vergangen. Insbesondere durch den Paradigmenwechsel bei den Grafikhardware-Herstellern von ausschließlich auf Rendering bezogene Spezialhardware zu leistungsfähiger general-purpose parallel-computing Hardware lohnt es sich zu untersuchen, welche Leistungssteigerung sich aus dem Einsatz aktueller Grafikhardware zur Unterstützung des Radiosity-Verfahrens ergibt.

1.2 Aufgabenstellung

Diese Arbeit beschäftigt sich mit der Implementation eines hardware-gestützten Prototyp des Radiosity-Verfahrens. Dafür muss sich zunächst in das Verfahren eingearbeitet und später Möglichkeiten für die Hardwareunterstützung untersucht werden. Auf Basis der Einarbeitung soll zunächst ein software-gestützter Prototyp entstehen. Eine Kopie des Prototyps kann dann an geeigneten Stellen durch Code mit Hardwareunterstützung ausgetauscht werden. Auf diesem Weg entstehen automatisch zwei Implementationen die in Hinblick auf ihre Laufzeit verglichen werden können. Dieser Vergleich ist ebenso Teil der Aufgabenstellung.

1.3 Abgrenzung

Die in dieser Arbeit entwickelte Software soll kein fertiges Produkt darstellen. Aufgrund einer beschränkten Arbeitszeit kann nicht sämtliche Forschung auf dem Gebiet in den Prototyp einfließen. Komfort-Funktionen die nicht in der späteren Evaluation hilfreich sind, sollen nicht entwickelt werden. Weiterhin soll der Fokus auf der Hardwareunterstützung der Radiosity-Berechnung liegen. Formfaktoren müssen nicht mit Hardwareunterstützung berechnet werden.

1.4 Struktur der Arbeit

Dieser Abschnitt gibt einen Überblick über die Struktur dieser Arbeit. Sie besteht aus sieben Teilen, wovon der erste diese Einleitung ist.

Das zweite Kapitel befasst sich mit den mathematischen und technischen Grundlagen für

diese Arbeit. Hier erhalten wir einen groben Überblick über die Beleuchtungsrechnung, wenden uns dann dem ursprünglichen Radiosity-Verfahren im Detail zu und betrachten die General-Purpose-Programmierung von Grafikhardware. Im darauf folgenden dritten Kapitel werden verwandte Arbeiten benannt, die den Stand der Technik widerspiegeln.

Im vierten und fünften Kapitel wird das Konzept und die Umsetzung dieser Arbeit vorgestellt. Hier betrachten wir was für die Implementation konzeptionell benötigt wird, schauen uns dann die Soft- und Hardware-Implementation von Radiosity an und erhalten zuletzt einen Einblick in die Arbeitsweise während der Umsetzung.

Das sechste Kapitel beschäftigt sich mit der Evaluation der entstandenen Software. Hier vergleichen wir die Soft- und Hardware-Implementation, betrachten ob das Ziel der Arbeit erreicht worden ist und stellen weiteres Potential für die Hardwareunterstützung vor. Schlussendlich wird im siebten und letzten Kapitel ein Fazit gegeben. Hier fassen wir die wichtigsten Punkte der Arbeit zusammen und erhalten einen Ausblick darüber welche Erweiterungen der entstandenen Software möglich sind.

2 Grundlagen

Dieses Kapitel legt die theoretischen Grundlagen nahe, die in dieser Arbeit Anwendung finden. Zuerst wird ein Überblick über die Beleuchtungsrechnung gegeben, danach wird das Radiosity-Verfahren vorgestellt und zuletzt auf die General-Purpose-Programmierung mit Grafikhardware eingegangen.

2.1 Beleuchtungsrechnung

Die Beleuchtungsrechnung in der Computergrafik ist ein integraler Bestandteil um eine Szene realistisch darzustellen. Formal handelt sie von der Bestimmung des reflektierten Lichtes an jedem beliebigen Punkt der Oberflächen einer Umgebung. Dabei wird zwischen dem spekularen und dem diffusen Anteil im reflektierten Licht unterschieden. Spekular und diffus sind dabei zwei Ausprägungen des Spektrums in der Streuung von Licht bei der Reflexion auf Oberflächen. Die Extreme dieses Spektrums bilden die ideal spekulare und die ideal diffuse Reflexion. Während der spekulare Anteil sich elliptisch um den Ausfallwinkel herum ausdehnt, streut der diffuse Anteil gleichmäßig in alle Richtungen [34]. Zur Berechnung dieser Phänomene haben sich zwei unterschiedliche Herangehensweisen entwickelt; lokale und globale Modelle. Beide Modelle können sowohl den spekularen als auch diffusen Anteil berücksichtigen.

2.1.1 Lokale Reflexionsmodelle

Lokale Reflexionsmodelle sind das Ergebnis eines von Schnelligkeit dominierten Zweiges in der Computergrafik. Sie berücksichtigen keine Wechselwirkungen zwischen Oberflächen und betrachten nur das von einer Lichtquelle eintreffende Licht auf eine Oberfläche [34]. Das heißt aber auch, dass sie eine Reihe von Effekten nicht darstellen können die aus der Wechselwirkung des Lichts zwischen Oberflächen entsteht. Dazu gehören zum Beispiel Schatten, Umgebungsreflexion oder Farbbluten.

Oft werden diese Effekte nachträglich durch zusätzliche Algorithmen oder Tricks hinzugefügt [34]. So können Schatten zum Beispiel eingebracht werden indem ausgehend von Lichtquellen Schattenvolumen berechnet werden [34]. Umgebungsreflexion kann mit sogenannten Environment-Maps simuliert werden, einer Textur die einem Bild der Umgebung entspricht [34].

Der Schwerpunkt von lokalen Reflexionsmodellen liegt darin die Interaktion von Licht mit Oberflächen aus unterschiedlichen Materialien zu simulieren. Dafür gibt es zwei verschiedene Ansätze. Der empirische Ansatz basiert auf der Konstruktion einer leicht zu berechnenden Formel welche die Reflexion einer realen Oberfläche imitiert [34]. Bei dem physikalischen Ansatz wird die Mikrogeometrie von Oberflächen modelliert um noch genauere Reflexionen zu ermöglichen [34].

Das wohl bekannteste und am weitesten verbreitetste lokale Reflexionsmodell ist das 1975 entwickelte Reflexionsmodell von Phong und basiert auf dem empirischen Ansatz [26, 34]. 1977 entwickelte Blinn ein lokales Reflexionsmodell auf Basis des Phong Modells und setzte zur Bestimmung der spekularen Komponenten einen physikalischen Ansatz ein [6, 34]. Cook und Torrance erweiterten dieses Modell 1982 um eine genauere Beschreibung der spektralen Zusammensetzung von Glanzlichtern [13, 34]. Trotz der Forschung auf dem Bereich bleibt das Phong Modell das meistgenutzte lokale Reflexionsmodell [34].

2.1.2 Globale Beleuchtungsrechnung

In der globalen Beleuchtungsrechnung wird für jeden Punkt in der Umgebung sämtliche Beleuchtung berücksichtigt, die diesen Punkt erreicht. Das bedeutet sowohl das direkt von Lichtquellen erhaltene Licht, als auch das indirekt über andere Oberflächen reflektierte Licht wird in die Berechnung mit einbezogen [34]. Oberflächen reflektieren das auf sie treffende Licht und geben einen Anteil davon wieder in die Umgebung ab. Da hierbei die globale Wechselwirkung des Lichts berücksichtigt wird, werden auch keine zusätzlichen Algorithmen für den Schattenwurf benötigt [34]. Schatten befinden sich einfach an den Stellen, die von weniger Licht erreicht werden. Auch Umgebungsreflexion oder Farbbluten sind ein natürliches Ergebnis der Berechnung.

Rendering Gleichung

1986 hat Kajiya die Rendering-Gleichung vorgestellt. Als mathematisches Modell dient sie dem Vergleich von unterschiedlichen Verfahren der globalen Beleuchtungsrechnung [20, 34]. Ein Verfahren der globalen Beleuchtungsrechnung kann dabei als eine Näherung an die Rendering-Gleichung gemessen werden [20, 34]. Die Gleichung bestimmt die von einem Punkt x' zu einem Punkt x übertragene Intensität und kann Formel 2.1 entnommen werden [20, 34].

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_s \rho(x, x', x'') I(x', x'') dx'' \right] \tag{2.1}$$

Hierbei gilt:

$I(x, x')$ ist die Intensität des Lichts die von Punkt x' auf Punkt x trifft.

$g(x, x')$ ist ein Geometrieterm. Wenn sich die Punkte x und x' nicht sehen können ist er 0. Ansonsten entspricht er $1/r^2$ wobei r die Entfernung zwischen den Punkten ist.

$\epsilon(x, x')$ ist der Abstrahlungsterm. Er entspricht dem von Punkt x' abgestrahlten Licht das x erreicht.

$\rho(x, x', x'')$ ist der Streuungsterm. Er entspricht dem Licht das von einem Punkt x'' ausgestrahlt, an einem Punkt x' gestreut wird und auf einen Punkt x trifft.

$\int_s$ entspricht dem Integral über alle Punkte der Umgebung.

Die Rendering-Gleichung besagt also, dass die von einem Punkt x' zu einem Punkt x übertragene Intensität dem von x' zu x ausgestrahlten Licht plus dem Licht das durch alle anderen Punkte x'' über x' zu x gestreut wird entspricht [34].

Lichtpfadnotation

Zur Kategorisierung von Verfahren der globalen Beleuchtungsrechnung kann die 1990 von Heckbert vorgestellte Lichtpfadnotation[1] verwendet werden. Sie beschreibt die Interaktion von Licht entlang des Pfades von einer Lichtquelle (L) zu einem Betrachter (E) mit diffusen (D) oder spekularen (S) Oberflächen [17]. Ein Pfad kann also durch eine Zeichenfolge beschrieben werden, die dem regulären Ausdruck L(D|S)*E entspricht [17]. Ein Pfad LDDSE bedeutet zum Beispiel, dass das Licht zweimal von einer diffusen Oberfläche und einmal von einer spekularen Oberfläche reflektiert wurde, bevor es den Betrachter erreicht hat. Anhand dieser Notation können für jedes Verfahren der globalen Beleuchtungsrechnung die möglichen Lichtpfade beschrieben werden. Für eine

[1] `https://de.wikipedia.org/wiki/Lichtpfadnotation` - Abgerufen: 27. März 2018

ganzheitliche Lösung der globalen Beleuchtungsrechnung müsste ein Verfahren alle Pfade L(D|S)*E berechnen können.

Verfahren der globalen Beleuchtungsrechnung

Als die am längsten verbreiteten Verfahren in der globalen Beleuchtungsrechnung haben sich das Ray Tracing von Whitted und Radiosity von Goral et al. etabliert. Beide Verfahren fokussieren sich in ihren Grundformen nur auf einen Teil des Lichts. Ray Tracing beschränkt sich dabei auf den ideal spekularen Anteil [35, 34], während sich Radiosity dem ideal diffusen Anteil widmet [16, 34].

Bei dem Ray Tracing von Whitted werden vom Betrachter Strahlen in die Umgebung geschossen [35]. Trifft ein solcher Strahl auf eine spekulare Oberfläche wird er gebrochen, reflektiert oder beides und die neuen Strahlen rekursiv weiter verfolgt [34]. Um mit diesem Verfahren ein Bild zu erzeugen, kann für jeden Pixel ein Strahl geschossen werden durch den die Farbe des Pixels ermittelt wird. Da die Strahlen vom Betrachter geschossen werden ist das Verfahren blickwinkelabhängig, bei einem Wechsel der Perspektive muss das Verfahren also erneut angewandt werden um ein Bild zu erzeugen. Das klassische Ray Tracing kann nur Pfade der Form LD?S*E berechnen [17].

In seiner Abhandlung über die Rendering-Gleichung hat Kajiya ein als Path Tracing bekanntes Verfahren entwickelt [20]. Es basiert auf dem Ray Tracing Verfahren und kombiniert es mit statistischen Verfahren aus der Mathematik. Im Gegensatz zum Ray Tracing endet ein Strahl nicht wenn er auf eine diffuse Oberfläche trifft. Stattdessen wird der Strahl zufällig reflektiert. Die Genauigkeit des erzeugten Bildes hängt dann von der Anzahl der geschossenen Strahlen ab. Wie das Ray Tracing ist auch das Path Tracing blickwinkelabhängig. Es berechnet allerdings Pfade der Form L(D|S)*E und stellt damit eine umfassende Lösung der globalen Beleuchtungsrechnung dar.

Das Radiosity-Verfahren betrachtet die Wechselwirkung von diffusen Oberflächen [16, 34]. Licht wird hier zwischen den Oberflächen reflektiert und als Intensitäten in der Geometrie der Umgebung gespeichert [16]. Dadurch ist das Verfahren blickwinkelunabhängig [16, 34] und muss nach einem Wechsel der Perspektive nicht erneut durchgeführt werden. Das klassische Radiosity-Verfahren berechnet ausschließlich Pfade nach dem Muster LD*E [17]. Da das Radiosity-Verfahren Bestandteil dieser Arbeit ist, wird es im nächsten Abschnitt noch detaillierter vorgestellt.

2.2 Radiosity

Radiosity ist ein Verfahren das die globale Beleuchtungsrechnung in dreidimensionalen Umgebungen löst und wurde erstmals 1984 von Goral et al. beschrieben [16]. Das Verfahren basiert auf dem Prinzip der Wärmeübertragung [16]. Licht wird hier als Energie verstanden, die von Oberflächen emittiert, absorbiert und reflektiert wird. Emittierende Oberflächen dienen als Lichtquellen einer Umgebung [16]. Das abgegebene Licht wird in der Umgebung solange zwischen den Oberflächen reflektiert, bis es absorbiert wurde. Das von jeder Oberfläche absorbierte Licht wird abschließend in Form von (z.B. RGB-) Intensitäten gespeichert und angezeigt. Durch das Speichern der Intensitäten in den Oberflächen, ist das Verfahren blickwinkelunabhängig [16]. Die Beleuchtungsrechnung muss beim Wechseln der Perspektive also nicht erneut durchgeführt werden. Die berechneten Intensitäten können einfach als Farbwerte verwendet und angezeigt werden.

Das Radiosity-Verfahren setzt voraus, dass alle Oberflächen in der Umgebung ideal diffus reflektieren [16]. Das von ideal diffus reflektierenden Oberflächen reflektierte Licht wird gleichmäßig in alle Richtungen gestreut. Aufgrund dieser Vereinfachung braucht in der Radiosity-Berechnung nicht berücksichtigt werden, dass die Lichtausdehnung auch unregelmäßig sein kann. Die Beschränkung auf ideal diffuse Oberflächen bedeutet aber auch, dass in der Reinform des Verfahrens zum Beispiel keine spiegelnden oder durchsichtigen Oberflächen unterstützt werden.

Das Radiosity-Verfahren berechnet für jede Oberfläche einen Lichtausstoß. Dieser wird über die gesamte Oberfläche als konstant angenommen. Deshalb werden Oberflächen in sogenannte Patches aufgeteilt, die als Ausgangsbasis des Verfahrens dienen [16]. Der Lichtausstoß eines Patches setzt sich dann zusammen aus der eigenen Leuchtkraft plus der Summe des Lichts aus der Umgebung das auf dieses Patch fällt. Um diesen Zusammenhang zu formalisieren, werden zunächst einige Kenngrößen vorgestellt.

Der Lichtausstoß eines Patches wird als B_i notiert. Er berechnet sich aus der Leuchtkraft des Patches E_i und dem akkumulierten Lichtausstoß $\sum_{j=1}^{N} B_j F_{ji}$ aller anderen Patches der auf das Patch i fällt. Letzterer muss noch mit der Reflektivität ρ_i des Patches multipliziert werden. Sie gibt an welche Farbanteile von dem Patch reflektiert werden und entspricht der unbeleuchteten Farbe des Patches. Somit legt sie fest welche Farbe das Patch annimmt, wenn es mit weißem Licht bestrahlt wird. Bei F_{ji} handelt es sich

um einen Formfaktor. Er gibt an welcher Anteil des von einem Patch j abgegebenen Lichtausstoß Patch i trifft. Weiteres zu Formfaktoren kann in Abschnitt 2.2.1 nachgelesen werden. Formel 2.2 fasst diese Zusammenhänge noch einmal zusammen [16].

$$B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ji} \tag{2.2}$$

B_i = Lichtausstoß des Patches i
E_i = Leuchtkraft des Patches i (Nur bei Lichtquellen $\neq 0$)
ρ_i = Reflektivität des Patches i (Farbe)
F_{ji} = Formfaktor von Patch j zu Patch i

Während die oben genannte Formel den Lichtausstoß eines Patches beschreibt, so muss sie für jedes Patch der Umgebung gelöst werden. Daraus wird ersichtlich, dass das Radiosity-Verfahren einen Laufzeitaufwand von $O(n^2)$ (mit n = Anzahl Patches) hat [9].

Die einzigen Unbekannten in dieser Gleichung sind die Lichtausstöße der Patches B_i. Die Leuchtkraft E_i und Reflektivität ρ_i der Patches sind Werte die vor der Berechnung bekannt sind. Ebenso sind die Formfaktoren F_{ji} bekannt, da sie sich aus der Geometrie der Umgebung berechnen [16]. Die Definition der Formfaktoren wird im nächsten Abschnitt behandelt.

2.2.1 Formfaktoren

Ein Formfaktor F_{ij} gibt an, welchen anteiligen Einfluss der Lichtausstoß eines gegebenen Patches i auf ein anderes Patch j hat. Er berechnet sich hierbei aus der Geometrie der Umgebung und ist abhängig von der Größe, Entfernung, Position und Orientierung der Oberflächen zueinander [16]. Da Formfaktoren ein Ergebnis der Geometrie sind, brauchen sie nur bei Veränderungen der Umgebung neu berechnet werden [16].

Die Berechnung eines Formfaktors F_{ij} geschieht wie in Formel 2.3 zu sehen ist durch ein zweifaches Integral über die Fläche der Patches i und j [16, 34]. A_i und A_j geben hierbei den Flächeninhalt der Patches an. ϕ_i und ϕ_j bezeichnen den Winkel zwischen

den Oberflächennormalen und einer Linie zwischen den beiden Patches. r ist die Distanz zwischen den Patches.

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\phi_i cos\phi_j}{\pi r^2} dA_j dA_i \tag{2.3}$$

Diese Formel berücksichtigt nicht, dass ein Patch für ein anderes Patch teilweise oder komplett unsichtbar sein kann. Dies wird als das Verdeckungs- oder Sichtbarkeitsproblem bezeichnet. Durch die Einführung eines Verdeckungsfaktors innerhalb der Integrale kann dieses Problem jedoch erfasst werden [10].

Für Formfaktoren gelten einige Regeln, anhand derer auch Plausibilitätsprüfungen formuliert werden können [16]. So gilt zwischen den beiden Formfaktoren zweier Patches eine Wechselseitigkeit:

$$A_i F_{ij} = A_j F_{ji} \tag{2.4}$$

Außerdem muss die Summe aller Formfaktoren eines Patches in einer geschlossenen Umgebung 1 ergeben:

$$\sum_{j=1}^{N} F_{ij} = 1 \tag{2.5}$$

Ein Patch das sich nicht selber sehen kann hat zu sich selbst den Formfaktor 0:

$$F_{ii} = 0 \tag{2.6}$$

In der Radiosity-Berechnung wird für jedes Patch-Paar ein Formfaktor benötigt. Daraus ergibt sich eine $n \times n$-Matrix wobei n die Anzahl der Patches bezeichnet und entspricht somit einem Speicherbedarf von $O(n^2)$.

2.2.2 Nusselt Analogie

Ein als Nusselt Analogie bekanntes Theorem besagt, dass zur Bestimmung eines Formfaktors die Projektion einer Oberfläche j auf eine Halbkugel über der Oberfläche i betrachtet werden kann [34] [2]. Die Projektion wird wiederum auf die Bodenfläche der Halbkugel projiziert und die von ihr eingenommene Fläche geteilt durch die Gesamtfläche ergibt den Formfaktor [10, 1]. Eine weitere nützliche Eigenschaft ist, dass alle Oberflächen welche dieselbe Projektion auf diese Halbkugel erzeugen auch denselben Formfaktor erhalten [10, 34]. Die Nusselt Analogie gilt als Grundlage für viele Verfahren zur Bestimmung von Formfaktoren.

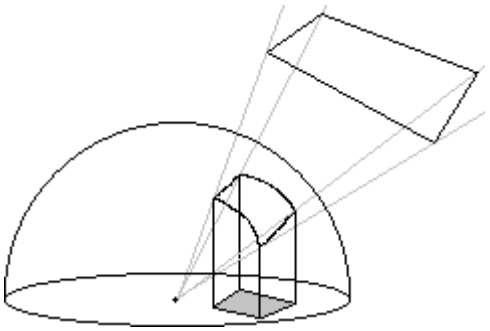

Abbildung 2.1: Projektion einer Oberfläche auf eine Halbkugel gemäß der Nusselt Analogie

2.2.3 Gathering und Shooting

Nach der Definition der Radiosity-Gleichung (siehe Formel 2.2) sammelt jedes Patch das einfallende Licht aus der Umgebung und nimmt es auf, was als „Gathering“ bezeichnet wird. Das bedeutet mit jeder Berechnung der Radiosity-Gleichung wird der Lichtausstoß von nur einem Patch berechnet. Dieser Prozess kann auch umgedreht werden, sodass jedes Patch seinen Lichtausstoß in die Umgebung abgibt, was „Shooting“ genannt wird [9]. Das hat zur Konsequenz, dass mit jedem Schritt der Lichtausstoß aller anderen Patches erhöht wird. Abbildung 2.2 veranschaulicht diesen Sachverhalt.

Nachdem nun die Grundlagen des Radiosity-Verfahrens vorgestellt wurden, widmet sich der nächste Abschnitt der General-Purpose-Programmierung von Grafikhardware.

[2] `https://en.wikipedia.org/wiki/View_factor#Nusselt_analog` - Abgerufen: 27. März 2018

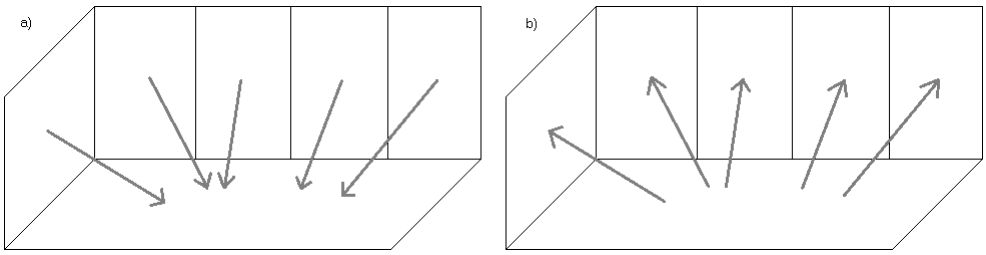

Abbildung 2.2: Schematische Darstellung des a) Gatherings und b) Shootings nach Cohen et al. [9]

2.3 General-Purpose Graphics-Processing-Unit Programmierung

Die Prozessoren aktueller Grafikhardware haben sich aufgrund der Nachfrage nach Echtzeit-Grafikanwendungen in hochgradig parallele Multikern-Prozessoren entwickelt [25]. Sie sind darauf ausgelegt dieselben Berechnungen parallel auf vielen Datenelementen durchzuführen [25]. Diese Art der Parallelisierung wird datenparallel[3] genannt und bietet sich besonders bei Grafikanwendungen an. So wird zum Beispiel bei der Bildsynthese die Farbe jedes Pixels parallel durch ein Shaderprogramm berechnet.

Da Grafikhardware früher überwiegend zur Darstellung von Grafikanwendungen eingesetzt wurde, überrascht es nicht, dass die ersten Schnittstellen zur Programmierung der Grafikhardware auf eben jene Grafikanwendungen ausgelegt waren. 1992 veröffentlichte Silicon Graphics die Grafikschnittstelle Open Graphics Library (OpenGL), die als standardisierte, plattformunabhängige Schnittstelle zur Entwicklung von Grafikanwendungen vorgesehen war [28]. Microsoft entwickelte mit Direct3D ebenfalls eine Grafikschnittstelle, die seit 2000 mit Version 8.0 das Programmieren von Shadern ermöglichte [28]. Damit hatten Entwickler zum ersten Mal die Möglichkeit Einfluss auf die von der Grafikhardware durchgeführte Berechnung zu nehmen [28].

Zur Berechnung von beliebigen Problemen eigneten sich die Grafikschnittstellen dennoch weniger. Um beliebige Berechnungen mit den parallelen Fähigkeiten der Grafikprozessoren auszuführen, mussten die Probleme als Renderingaufgaben modelliert werden [28]. Eingabedaten mussten so als Farben, Texturkoordinaten oder andere Grafikattribute

[3] `https://en.wikipedia.org/wiki/Data_parallelism` - Abgerufen: 27. März 2018

kodiert werden [28]. Die Pixel-Shader konnten dann frei programmiert beliebige Berechnungen auf den kodierten Daten durchführen und die berechneten Ergebnisse wieder als Farbdaten ausgeben [28].

Das Interesse die parallelen Fähigkeiten der Grafikhardware zur Berechnung von beliebigen Problemen einzusetzen führte dazu sie dem mittlerweile als General-Purpose Computation on Graphics-Processing-Unit (GPGPU)-Programmierung bezeichneten Programmierparadigma zu öffnen und mit entsprechenden Schnittstellen zu unterstützen. Daraus entwickelten sich über die Zeit die GPGPU-Schnittstellen Open Computing Language (OpenCL) und die NVidia Compute Unified Device Architecture (CUDA). Durch sie können beliebige Berechnungen mit der Grafikhardware durchgeführt werden. Die Berechnung wird dabei in einem Programm definiert das Kernel genannten wird.

2.3.1 OpenCL

Die Open Computing Language wurde von Apple entwickelt und in Zusammenarbeit mit anderen Firmen der Khronos Group zur Standardisierung eingereicht [4]. Durch das Etablieren als offenen Standard wird er von vielen Grafikhardwarehersteller unterstützt [5]. Der OpenCL-Standard ist dabei eine allgemeine Schnittstelle zur parallelen Programmierung und beschränkt sich damit nicht ausschließlich auf Grafikhardware. Kernels werden in OpenCL mit OpenCL C geschrieben, einer auf der Programmiersprache C basierenden Sprache die um einige Datentypen erweitert wurde [6].

2.3.2 NVidia CUDA

Die Compute Unified Device Architecture ist die von NVidia entwickelte, proprietäre Schnittstelle zur GPGPU-Programmierung auf NVidia Grafikhardware. Ähnlich wie OpenCL stellt auch CUDA eine Programmiersprache zum Schreiben von Kernels zur Verfügung. Sie heißt CUDA C und war die erste Programmiersprache zur General-Purpose-Programmierung von Grafikhardware [28]. Sie basiert ebenfalls auf C und erweitert es um einige Datentypen und Keywords.

In diesem Kapitel wurde eine Einordnung des Radiosity-Verfahrens im Kontext der Beleuchtungsrechnung gegeben. Das Verfahren wurde in seiner ursprünglich beschriebenen

[4] `https://de.wikipedia.org/wiki/OpenCL` - Abgerufen: 27. März 2018
[5] `https://www.khronos.org/opencl/` - Abgerufen: 27. März 2018
[6] `https://de.wikipedia.org/wiki/OpenCL#OpenCL_C` - Abgerufen: 27. März 2018

Form vorgestellt. Auch wurde die Historie der GPGPU-Programmierung vorgestellt. Nachdem die benötigten Grundlagen nahegelegt wurden, beschäftigt sich das nächste Kapitel mit einem Überblick über die zum Radiosity-Verfahren entstandene Forschung.

3 Verwandte Arbeiten und Stand der Technik

Dieser Abschnitt präsentiert den Stand der Technik in Bezug auf das Radiosity-Verfahren. Es werden die Schwierigkeiten des Verfahrens vorgestellt sowie Arbeiten die sich mit deren Lösung befassen. Während die Problembereiche des Verfahrens umfänglich wiedergeben werden, stellt diese Arbeit nicht den Anspruch einen voll-umfänglichen Überblick über sämtliche Forschung auf dem Gebiet zu geben. Stattdessen wird sich auf die im Laufe der Recherchearbeiten begutachtete Forschung beschränkt. Da das Radiosity-Verfahren bereits vor 34 Jahren beschrieben wurde und Gegenstand intensiver Forschung war, existieren bereits eine Reihe von Sammelwerken die einen guten und umfassenden Überblick geben [12, 1, 34].

3.1 Formfaktoren

Die Bestimmung der Formfaktoren stellt sich wie bereits in Abschnitt 2.2.1 angerissen wurde zwei Problemen. So muss zur Bestimmung eines Formfaktors das zweifache Integral aus Formel 2.3 ausgewertet werden. Für beliebige Formen ist dies allerdings schwer zu realisieren [10]. Das zweite Problem ist die Lösung des Verdeckungsproblems, also zu ermitteln ob sich zwei Patches sehen können. Da die Formfaktorberechnung unter einem vertretbaren Rechenaufwand nicht trivial ist, haben sich unterschiedliche Verfahren entwickelt, welche die Komplexität mittels Approximation reduzieren.

3.1.1 Hemicube

1985 entwickelten Cohen und Greenberg einen Algorithmus zur Bestimmung von Formfaktoren der auch das Verdeckungsproblem löst. Das Hemicube Algorithmus genannte Verfahren verwendet einen Halbwürfel als Approximation für die in der Nusselt Analogie beschriebenen Halbkugel. Jede Seite des Halbwürfels wird in Zellen unterteilt, die Pixel

genannt werden. Die Anzahl der Pixel auf der Oberseite wird dann als die Auflösung des Halbwürfels verstanden. Abbildung 3.1 zeigt einen exemplarischen 7x7 Hemicube.

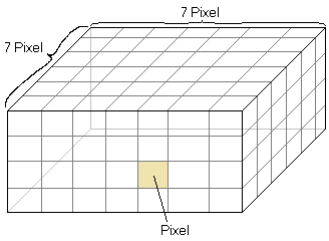

Abbildung 3.1: Ein Hemicube mit einer Auflösung von 7

Um den Formfaktor eines Patches zu bestimmen wird es zunächst auf die Seiten des Hemicubes projiziert und anschließend ermittelt, welche Pixel durch die Projektion getroffen wurden. Jedem Pixel ist ein Δ-Formfaktor zugewiesen. Die Δ-Formfaktoren der durch die Projektion getroffenen Pixel werden aufsummiert und ergeben dann den Formfaktor des Patches [10, 34]. Der Δ-Formfaktor eines Pixels kann abhängig davon, ob er sich auf der Oberseite oder einer der Außenseiten befindet, mit den Formeln 3.1 und 3.2 berechnet werden [10, 34]. Dabei sind die Δ-Formfaktoren nur abhängig von der Auflösung des Halbwürfels. Außerdem brauchen sie nur einmal berechnet werden, solange sich die Auflösung nicht ändert.

$$\Delta F_{Top} = \frac{1}{\pi(x^2 + y^2 + 1)^2} \Delta A \tag{3.1}$$

$$\Delta F_{Side} = \frac{z}{\pi(y^2 + z^2 + 1)^2} \Delta A \tag{3.2}$$

Pixel können wie bei einem Z-Puffer Tiefeninformationen speichern und Patches zuordnen [10, 1]. Wenn ein Pixel bei der Projektion eines Patches bereits ein näheres Patch kennt, gilt der Pixel durch die Projektion als nicht getroffen da das fernere Patch von dem näheren Patch verdeckt wird. Damit wird das Verdeckungsproblem ebenfalls gelöst.

1988 merken Cohen et al. an, dass die Formfaktorberechnung mit dem Hemicube durch Grafikhardware unterstützt werden kann [9]. Für jede Seite des Hemicubes kann ein Bild gerendert werden. Die von einem Patch getroffenen Pixel werden dabei in einer für jedes Patch eindeutigen Farbe markiert [1]. Anhand der Farbe eines Pixels kann

anschließend rekonstruiert werden, welches Patch auf dieses Pixel projiziert wurde. Die Grafikhardware löst beim Rendering bereits sowohl die Projektion als auch die Verdeckungsberechnung.

3.1.2 Cubic Tetrahedron

1991 beschreiben Beran-Koehn und Pavicic einen anderen Weg Formfaktoren zu bestimmen. Bei ihrem Ansatz wird ein kubischer Tetraeder verwendet um die Hemisphäre zu approximieren [4]. Der Vorteil dieses Ansatzes gegenüber dem Halbwürfel liegt darin, dass die drei Oberflächen des Tetraeders gleichartig sind und somit keine Unterscheidung zwischen Ober- und Außenseiten gemacht werden muss [4]. Abbildung 3.2 zeigt einen exemplarischen 4x4 Kubischen Tetraeder.

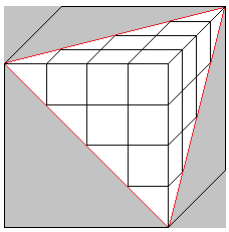

Abbildung 3.2: Ein kubischer Tetraeder mit einer Auflösung von 4

Weiterhin wird nur eine Formel zur Berechnung der Δ-Formfaktoren benötigt [5]. Sie kann Formel 3.3 entnommen werden.

$$\Delta F = \frac{u + v + 1}{\pi(u^2 + v^2 + 1)^2 3^{\frac{1}{2}}} \Delta A \tag{3.3}$$

3.1.3 Analytisch bestimmte Formfaktoren

1989 untersuchen Baum et al. verschiedene Gründe für Fehler in der Formfaktorbestimmung mit dem Hemicube-Verfahren [2]. Dabei präsentieren sie drei Annahmen die durch das Hemicube-Verfahren getroffen werden [2]. Werden diese Annahmen verletzt, treten Fehler in der Formfaktorberechnung auf. Sie schlagen deshalb vor Formfaktoren analytisch zu berechnen, wenn diese Annahmen gebrochen werden [2]. Die Sichtbarkeitsprüfung kann dagegen weiterhin durch das Hemicube-Verfahren geschehen [2].

3.1.4 Stochastic Ray-Casting

Eine weitere Möglichkeit zur Bestimmung der Formfaktoren ergibt sich aus dem Einsatz stochastischer Methoden. Dabei werden die Formfaktoren bestimmt, indem von einem Patch ausgehend Strahlen in die Umgebung geschossen werden. Die Strahlen werden bei Austritt aus der umgebenden Halbkugel in die Richtung der Oberflächennormale am Austrittsort gebrochen [1]. Um den Aufwand beherrschbar zu halten werden die Strahlen zufällig aus der gesamten Fläche des Patches und in einer beschränkten Zahl geschossen. Je größer die Zahl der Strahlen, desto genauer die Formfaktoren. Der Formfaktor für ein Patch j ermittelt sich dann aus der Anzahl der Strahlen eines Patches i die j getroffen haben geteilt durch die Anzahl aller geschossenen Strahlen [24, 1]. Im Gegensatz zum Hemicube-Verfahren ist keine aufwändige Projektion erforderlich und Aliasing-Artefakte werden durch die zufällige Verteilung reduziert [33, 1].

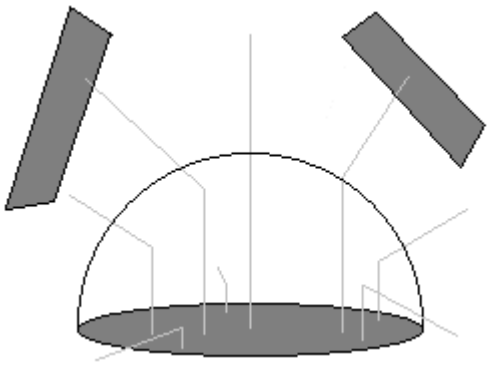

Abbildung 3.3: Darstellung des stochastischen Ray-Castings mit acht geschossenen Strahlen. Zwei treffen dabei auf das linke Patch, was einen Formfaktor von $2/8 = 0.25$ ergibt. Das rechte Patch erhält einen Formfaktor von $1/8 = 0.125$.

3.2 Meshing Strategien

Ein weiteres Problem in der Radiosity-Berechnung ist die Struktur und die Auflösung der Polygonnetze aus denen Objekte und Oberflächen modelliert sind. Dies äußert sich in zwei Ausprägungen. Zum Einen müssen sie gewisse Anforderungen erfüllen um in dem Radiosity-Verfahren eingesetzt werden zu können. Baum et al. haben diese Anforderungen 1991 untersucht [3]. Zum Anderen müssen die Netze fein genug aufgelöst sein um Lichteffekte wie Schatten einfangen zu können.

Wie bereits beschrieben berechnet das Radiosity-Verfahren für jeden Patch einen Lichtausstoß. Dieser wird über das gesamte Patch als konstant angenommen. Das führt automatisch zu einem Qualitätsproblem. Oberflächen auf denen Schattenkanten, oder allgemeiner abrupte Änderungen der Intensität, verlaufen erfordern eine viel feinere Auflösung von Patches als es Oberflächen tun über deren Fläche sich die Intensität nur langsam verändert.

Meshing Strategien beschäftigen sich also mit der angemessenen Unterteilung einer Umgebung. Es muss entschieden werden welche Oberflächen wie fein unterteilt werden und somit Qualität gegen Laufzeit und Platzbedarf abgewogen werden. Durch den Laufzeit- und Platzaufwand von $O(n^2)$ führt eine zu feine Unterteilung schnell zu einer deutlich aufwändigeren Berechnung. Eine zu grobe Unterteilung dagegen stellt Änderungen der Intensität nicht ausreichend dar.

3.2.1 Hierarchisches Meshing

1986 haben Cohen et al. vorgeschlagen Polygonnetze hierarchisch zu unterteilen. Patches werden in Elemente unterteilt, die untereinander wiederum in einer Baumstruktur organisiert sein können [11]. Bei dieser Aufteilung werden den Patches und Elementen unterschiedliche Bedeutungen zugeschrieben. So geben die Patches das Licht in die Umgebung ab und die Elemente nehmen es auf [11, 9]. Bei dieser Aufteilung können die Patches größer sein, während die Elemente feiner aufgelöst sind. Da Patches als Sender und Elemente als Empfänger fungieren, brauchen nur Patch-zu-Element-Formfaktoren berechnet werden anstelle von Element-zu-Element-Formfaktoren [9]. Der Lichtausstoß eines Patches setzt sich dann aus dem nach Fläche gewichteten, durchschnittlichen Lichtausstoß seiner Elemente zusammen [9].

3.2.2 Adaptive Unterteilung

Während die hierarchische Unterteilung bereits eine feinere Auflösung ermöglicht, muss noch immer entschieden werden wie fein eine Oberfläche unterteilt werden muss. Dies hängt davon ab, wie abrupte Farbverläufe über die Oberflächen der Umgebung fallen. Da diese Information vor der Berechnung nicht zur Verfügung steht, schlagen Cohen et al. vor Elemente adaptiv während der Berechnung zu unterteilen [11]. So reicht ein Element für ein Patch, über dessen Fläche sich die Intensität kaum oder gar nicht ändert, aus.

An Schattenkanten muss jedoch viel feiner unterteilt werden [11]. Und auch für Patches die als Lichtquelle dienen kann eine weitere Unterteilung notwendig sein. Jedes Patch wird im Radiosity-Verfahren als Punktlichtquelle gehandhabt [1, 34]. Eine Lichtquelle deren Patches zu grob unterteilt sind wird nicht akkurat wiedergegeben. Das Ziel der adaptiven Unterteilung ist es also den Detailgrad von Patches und Elementen nur an den Stellen zu erhöhen, an denen es notwendig ist.

3.2.3 Discontinuity Meshing

Die vorher beschriebenen Verfahren unterteilen Polygonnetze in gleichmäßige Quadrate oder Dreiecke [18]. In Kombination mit der adaptiven Unterteilung wurden Schattenkanten so nur angenähert. Um Schattenkanten bereits vor der Radiosity-Berechnung zu berücksichtigen, beschrieb Heckbert 1992 ein Verfahren zur Unterteilung von Polygonnetzen entlang der Schattenkanten. Dafür werden zunächst die Schattenkanten bestimmt. Sie entstehen dort, wo die Sichtbarkeit eines Empfängers für einen Sender durch eine andere Oberfläche unterbrochen wird [18]. Anschließend werden die Elemente entlang der Schattenkanten positioniert, sodass ihre Ränder den Schattenkanten folgen um sie angemessen in dem Polygonnetz zu repräsentieren [18]. 1993 griffen Lischinski et al. das Verfahren auf und kombinierten es mit der adaptiven Unterteilung [23].

3.3 Lösungsansätze

Nachdem in den vorangegangenen Abschnitten dieses Kapitels Probleme vorgestellt wurden, die sich insbesondere auf die Qualität auswirken, beschäftigt sich dieser Abschnitt mit der Lösung der Radiosity-Gleichung (Formel 2.2) aus Abschnitt 2.2.

3.3.1 Full Matrix Radiosity

Der originale Lösungsansatz von Goral et al. sieht die Konstruktion und anschließende Lösung einer Matrix vor [16]. Die Radiosity-Gleichung muss für jedes Patch gelöst werden, woraus sich ein System von n Gleichungen ergibt. Es wird die Radiosity-Matrix genannt und seine Lösung entspricht der Lösung des Radiosity-Verfahrens [16, 34].

Da zunächst die komplette Matrix aufgebaut wird, müssen vorher alle Formfaktoren berechnet werden.

$$\begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_i \end{bmatrix} = \begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & -\rho_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \dots & 1-\rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} \tag{3.4}$$

3.3.2 Progressive Refinement Radiosity

1988 haben Cohen et al. erstmals Unternehmungen angestrengt, das Radiosity-Verfahren in interaktivem Zeitaufwand zu realisieren [9]. Das dabei entstandene „Progressive Refinement Radiosity“ genannte Verfahren nimmt sich des Problems an indem es Teillösungen der Radiosity-Gleichung berechnet die angezeigt werden können, während die Lösung im Hintergrund immer weiter voranschreitet und realistischer wird [9]. Das Radiosity-Verfahren bietet sich durch seine Blickwinkel-Unabhängigkeit besonders für eine progressive Verfeinerung an, da die Berechnungen im World-Space stattfinden [9].

Wie bereits in Abschnitt 2.2 und 2.2.1 erwähnt, hat das Radiosity-Verfahren einen Laufzeitaufwand und Speicherbedarf von jeweils $O(n^2)$. Zusätzlich werden die Formfaktoren im ursprünglichen Verfahren vorab berechnet [16, 9]. Durch die Erzeugung von Teillösungen wird die Radiosity-Berechnung schrittweise durchgeführt. Ein Schritt verarbeitet genau ein Patch, wodurch nur noch die Formfaktoren eines Patches pro Schritt benötigt werden. Um dem hohen Speicherbedarf und anfänglichen Rechenaufwand entgegenzuwirken, berechnen Cohen et al. die Formfaktoren deshalb erst bei Bedarf [9].

Um einen visuell eleganten Übergang von einer unbeleuchteten zu einer beleuchteten Szene zu erhalten, verwenden Cohen et al. den in den Grundlagen in Abschnitt 2.2.3 beschriebenen Shooting-Ansatz. Weiterhin werden die Patches bei jedem Schritt in der Reihenfolge ihres Lichtanteils verarbeitet, um möglichst schnell ein brauchbares Bild erzeugen zu können [9]. Auf diese Weise werden Lichtquellen zuerst verarbeitet, da sie initial über einen Lichtausstoß $B_i > 0$ verfügen. Danach werden die Patches die am meisten Licht von den Lichtquellen erhalten haben verarbeitet und so weiter [9]. Diese Vorgehensweise hat auch zur Folge, dass mit jedem Schritt immer weniger visuelle

Veränderung stattfindet. Der Lichtanteil eines Patches definiert sich hierbei nach dem mit der Fläche gewichteten Lichtausstoß des Patches $B_i A_i$ [9].

Die Laufzeitkomplexität des Verfahrens verändert sich nicht. Stattdessen wird die Interaktivität dadurch gewährleistet, dass bereits Teillösungen angezeigt werden können. Die Teillösungen konvergieren dabei zur richtigen Lösung [9]. Da insbesondere in den später durchgeführten Schritten sehr geringer Lichtaustausch stattfindet, kann eine Toleranzgrenze definiert werden nach der die Lösung als konvergiert angesehen wird [9, 1].

Baum et al. merken 1989 an, dass die mit dem Progressive Refinement Radiosity Verfahren erzeugten Bilder zu einer anderen Lösung konvergieren als der ursprüngliche Ansatz [2].

Umgebungslichtterm

Durch den Progressive Refinement Radiosity Algorithmus entwickelt sich die zu berechnende Umgebung von einer anfänglich dunklen Darstellung zu einer immer helleren. Das hat zur Folge, dass Patches, die keiner direkten Beleuchtung ausgesetzt sind, unter bestimmten Umständen sehr lange dunkel bleiben [9]. Um diesem Problem zu begegnen wurde ein sogenannter Umgebungslichtterm eingeführt. Er erfüllt einen rein ästhetischen Aspekt und wird mit der fortschreitenden Berechnung einer Lösung immer kleiner. Der Umgebungslichtterm wird dafür zur Anzeige den temporären Lichtausstößen der Patches hinzugefügt, in der Lösung aber nicht weiter berücksichtigt [9]. Er basiert dabei auf der aktuellen Schätzung der Lichtausstöße aller Patches in der Umgebung [34].

3.3.3 Instant Radiosity

1997 entwickelte Keller ein Verfahren namens Instant Radiosity [22]. Die Idee dabei ist Strahlen von den Lichtquellen in die Umgebung zu schießen und bei einer Kollision mit einer Oberfläche eine Punktlichtquelle an dem Schnittpunkt zu erzeugen [22]. Während die originalen Lichtquellen dann ausschließlich für die direkte Beleuchtung eingesetzt werden, geschieht die indirekte Beleuchtung anhand der erzeugten Punktlichtquellen [22]. Der Ansatz arbeitet dabei im Image-Space und berechnet keine Intensitäten in diskretisierten Patches [22]. Durch den großen Einsatz von Hardwareunterstützung erreicht Keller damit Renderingraten von einigen Sekunden [22]. Trotz des Namens hat

der Ansatz allerdings nicht viel mit dem eigentlichen Radiosity-Verfahren gemein und repräsentiert eher eine weitere Möglichkeit die globale Beleuchtungsrechnung zu lösen.

3.4 Erweiterungen

Das Radiosity-Verfahren ist in seiner ursprünglichen Form ausschließlich auf die Wechselwirkung zwischen diffusen Oberflächen beschränkt. Spekulare Reflexionen oder transparente Oberflächen unterstützt das Verfahren nicht. Im Folgenden werden Erweiterungen vorgestellt, die sich damit beschäftigen auch spekulare Wechselwirkungen zu berücksichtigen.

3.4.1 Nicht-Diffuse Oberflächen

1986 haben Immel et al. eine Erweiterung vorgeschlagen durch die das Radiosity-Verfahren auch spekulare Reflexion berechnen kann [19, 32]. Ihre Lösung berechnet dabei für jede Richtung pro Patch eine Intensität [19, 32]. Dadurch bleibt das Verfahren weiterhin blickwinkelunabhängig [19, 32]. Allerdings stellt die Methode sehr hohe Anforderungen an die Rechenleistung [19, 30] und hat mit Aliasing-Problemen zu kämpfen [19].

3.4.2 Multi-Pass Methoden

Da das Ray Tracing Verfahren gerade im Bereich der spekularen Reflexion seine Stärken hat, liegt die Überlegung nahe Radiosity und Ray Tracing zu kombinieren. Wallace et al. schlagen deshalb 1987 vor mittels des Radiosity-Verfahrens als Prä-Prozess die diffusen Wechselwirkungen und mittels Ray-Tracing als Post-Prozess die spekularen Wechselwirkungen zu berechnen [32]. Da die diffusen Intensitäten auch von spekularen Wechselwirkungen beeinflusst werden, erweitern Wallace et al. das als Prä-Prozess eingesetzte Radiosity-Verfahren, sodass ihr Einfluss berücksichtigt wird ohne die konkreten spekularen Anteile zu berechnen [32]. So erhalten zum Beispiel zwei Oberflächen die sich durch eine spiegelnde Oberfläche sehen können einen weiteren Formfaktor [32]. Der Post-Prozess erzeugt anschließend mittels Ray Tracing ein Bild das die spekularen Anteile ebenfalls berücksichtigt [32]. Weitere Verbesserungen auf diesem Gebiet liefern [30, 27, 8].

4 Konzept

Nachdem die Grundlagen in Kapitel 2 nahegelegt wurden, wird in diesem Kapitel das Konzept für die Umsetzung des Radiosity-Verfahrens im Rahmen dieser Arbeit vorgestellt. Laut Aufgabenstellung ist die Entwicklung einer software- und einer hardware-gestützten Implementation die Zielsetzung um eine Vergleichbarkeit zwischen beiden zu erreichen. Die Software-Implementation ist dabei gleichzeitig die Grundlage für die Hardware-Implementation. Sie soll nach ihrer Fertigstellung kopiert werden, damit geeignete Stellen durch von Hardware unterstütztem Code ausgetauscht werden können und somit zwei separat lauffähige Implementationen zur Verfügung stehen.

4.1 Software-Radiosity

Die Grundlage für das Konzept der Software-Variante bildet das Buch „Radiosity : A Programmer's Perspective“ von Ian Ashdown von 1994 [1]. Ashdown beschreibt in diesem Buch die Implementation des Radiosity-Verfahrens in der Programmiersprache C++ und stellt den entstandenen Code zur Verfügung. Dabei bezieht er die bis dahin entstandene Forschung in die Implementation mit ein, oder weist darauf hin an welchen Stellen er es nicht tut.

Da das Ziel dieser Arbeit die Entwicklung eines Prototyps ist, werden einige Abstriche vorgenommen. Diese Abstriche werden an entsprechender Stelle erwähnt. So verzichtet diese Arbeit auf die Modellierung von Elementen und geht nur von einer Unterteilung in Patches aus.

4.1.1 Formfaktoren mittels Hemicube-Verfahren

Die Berechnung der Formfaktoren wird mit dem in Abschnitt 3.1.1 beschriebenen Hemicube-Verfahren realisiert. Während die Betrachtung der Grundlagen relativ übersichtlich ist, gilt dies für die Umsetzung nicht mehr. Tatsächlich ist die Berechnung der Formfaktoren in der Umsetzung des Radiosity-Verfahrens der aufwändigste Part [1].

So müssen Δ-Formfaktoren berechnet, Patches auf den Hemicube projiziert und die Δ-Formfaktoren der von einem Patch getroffenen Pixel aufsummiert werden, um einen Formfaktor zu berechnen. Im Folgenden wird nahegelegt wie diese Herausforderungen gelöst werden.

Δ-Formfaktoren

Wie in Abschnitt 3.1.1 beschrieben, lässt sich für jeden Pixel eines Hemicubes ein Δ-Formfaktor berechnen. Dieser Formfaktor bleibt für jeden Pixel immer gleich und braucht somit nur einmal berechnet werden. Um das zu erreichen, können die Δ-Formfaktoren in einem zweidimensionalen Array zwischengespeichert werden. Um alle Δ-Formfaktoren eines $N * N$-Hemicubes zu speichern, müssen für die Oberseite $N * N$ Formfaktoren und für die vier Außenseiten insgesamt $4 * N * \frac{N}{2}$ Formfaktoren gespeichert werden.

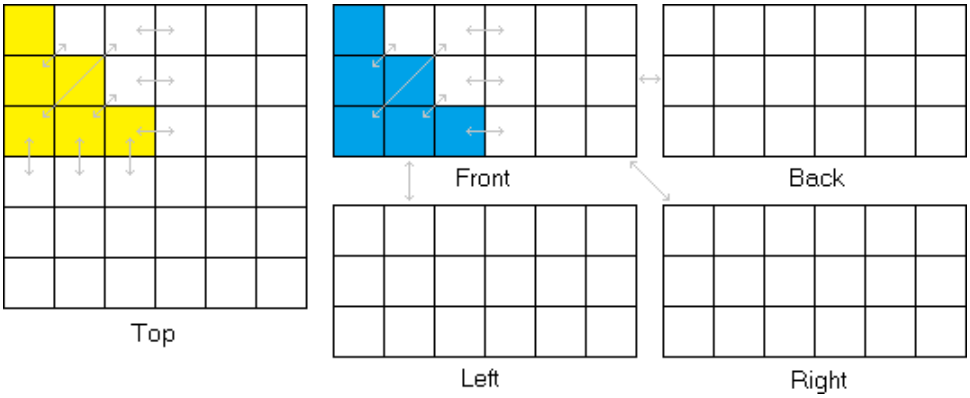

Abbildung 4.1: Exemplarische Darstellung der Symmetrien eines 6x6 Hemicubes. Die Pfeile zeigen die Symmetrien auf. Auf der Top-Seite reicht es aus die Formfaktoren der markierten Felder zu speichern, um den Formfaktor jedes anderen Feldes der Oberseite ermitteln zu können. Bei den Außenseiten (Front, Left, Back und Right) reicht es ebenfalls aus die Formfaktoren der markierten Felder zu speichern um den Formfaktor jedes Feldes jeder Außenseite ermitteln zu können.

Interessant hierbei ist, dass sowohl die Oberseite als auch die Außenseiten Symmetrien aufweisen, was es wiederum ermöglicht den Speicherbedarf zu reduzieren [1]. Auf der Oberseite fällt so eine achtfache Symmetrie auf und für eine Außenseite eine vierfache Symmetrie. Für Außenseiten gilt zusätzlich, dass jede Seite dieselben Formfaktoren hat. Abbildung 4.1 veranschaulicht die Symmetrien. Damit kann der Speicherbedarf für die Δ-Formfaktoren deutlich eingeschränkt werden [1].

Patch-Formfaktoren

Um nun die Formfaktoren von allen Patches zueinander zu berechnen, muss jedes Patch in der Umgebung für jedes andere Patch auf einen Hemicube projiziert werden der sich auf dem gerade betrachteten Patch befindet und in Richtung seiner Normalen ausgerichtet ist. In Bezug auf die Projektion, kann der Hemicube als ein Betrachtungssystem verstanden werden [1]. Jede Seite des Hemicubes bildet dabei ausgehend von dem Hemicube Mittelpunkt ein Frustum. Die Umgebung kann nun wie in einer Grafikpipeline üblich auf die View-Plane eines jeden dieser Frusta projiziert werden.

Polygon Clipping

Ashdown löst die Projektion, indem das betrachtete Patch zunächst für jede Seite des Hemicubes zugeschnitten wird. Dadurch erhält man für jede Hemicube-Seite ein Polygon dessen Eckpunkte allesamt im Frustum der Seite liegen. Ashdown verwendet zum Zuschneiden der Patches den Sutherland-Hodgman-Algorithmus[1] [1]. Diese Algorithmen werden auch Clipping-Algorithmus genannt.

Bei dem Sutherland-Hodgman-Algorithmus wird ein Polygon gegen eine beliebige Zahl von Ebenen, sogenannten Clipping-Planes geschnitten [31]. Die Clipping-Planes sind in diesem Fall die Seitenflächen des Frustums der aktuellen Hemicube-Seite. Nun wird das Polygon nacheinander gegen jede Clipping-Plane geschnitten, wie es in Algorithmus 4.1 dargestellt wird. Eine visuelle Darstellung des Verfahrens kann Abbildung 4.2 entnommen werden.

Scan Conversion

Anhand der zugeschnittenen Polygone kann mittels eines Scan-Conversion-Algorithmus[2] ermittelt werden, welche Pixel des Hemicubes von der Projektion getroffen werden [1]. Die Projektion auf die Hemicube-Seiten kann als das Zeichnen eines Bildes pro Seite aus der Sicht des Hemicube-Ursprungs verstanden werden. Der Scan-Conversion-Algorithmus wird dann eingesetzt um die Polygone aus dem View-Space der Hemicube-Seiten in den Image-Space zu diskretisieren und auszufüllen.

Dafür werden zunächst die Eckpunkte des Polygons in den Image-Space transformiert

[1] `https://de.wikipedia.org/wiki/Algorithmus_von_Sutherland-Hodgman` - Abgerufen: 27. März 2018
[2] `https://www.youtube.com/watch?v=23HEwdcphg4` - Abgerufen: 27. März 2018

Algorithmus 4.1: Sutherland-Hodgman Polygon Clipping Algorithm

```
OutPolygon ← ToClipPolygon
foreach ClippingPlane in Frustum do
    InPolygon ← OutPolygon
    OutPolygon ← ∅
    S ← InPolygon.Last()
    foreach E in InPolygon do
        if S inside ClippingPlane ⊕ E inside ClippingPlane then
            OutPolygon.Add(ClipEdge.Intersect(S, E))
        end
        if E inside ClippingPlane then
            OutPolygon.Add(E)
        end
        S ← E
    end
end
```

und ermittelt über welche als Scanlines bezeichneten Zeilen des Bildes sich das Polygon erstreckt. Anschließend werden die nun im Image-Space befindlichen Kanten des Polygons als Eckpunkt-Paare verarbeitet. Für jede Scanline die sie schneiden wird die X- und Z-Koordinate des Schnitts ermittelt und in der Scanline vermerkt. Die Y-Koordinate entspricht hierbei der Nummer der Scanline und braucht nicht gespeichert werden. Bei einem konvexen Polygon können so in jeder Scanline maximal zwei Schnitte entstehen. Aus der Sicht der Scanline sind dies der Anfang und das Ende des Polygons.

Nachdem diese Informationen vorliegen, können die Scanlines nacheinander durchlaufen werden und ermittelt werden welche Pixel der Scanline das Polygon trifft. Dem Pixel wird dann das Patch und seine Tiefeninformation zugeordnet [1]. Anhand der Tiefeninformation kann entschieden werden welches Patch dem Pixel zugeordnet bleiben soll, wenn ein Patch ein anderes überdeckt [1]. Nachdem ermittelt wurde, welche Pixel von der Projektion getroffen wurden, brauchen ihre Δ-Formfaktoren nur noch aufsummiert werden um den Formfaktor für das Patch zu bestimmen. Abbildung 4.3 stellt diesen Ablauf nochmal grafisch dar.

4.1.2 Progressive Refinement Radiosity

Das Progressive Refinement Radiosity Verfahren wird sowohl in der Shooting- als auch der Gathering-Variante umgesetzt. Während die Shooting-Variante naheliegender ist, da sie

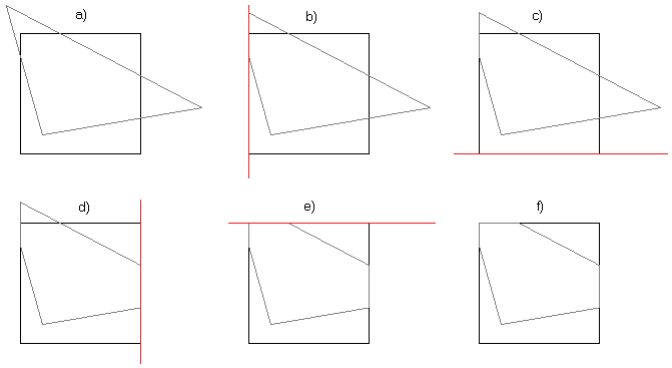

Abbildung 4.2: Schrittweise Darstellung des Zuschneidens von einem Dreieck gegen ein Polygon mit vier Seiten in 2D.

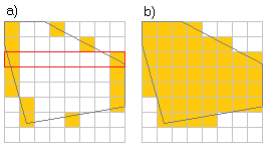

Abbildung 4.3: a) Zeigt vereinfacht die ermittelten Start- und End-Koordinaten der Scanlines für das in Abbildung 4.2 zugeschnittene Polygon b) Zeigt wieder vereinfacht das Füllen der Zeilen um das Polygon den Pixeln zuzuweisen

sowohl im originalen Paper als auch in Ashdowns Implementation genutzt wird [9, 1], so soll die Software-Implementation als Grundlage für die Hardware-Implementation dienen. Wie in Abschnitt 4.2 beschrieben wird, bietet sich für die Hardware-Implementation die Gathering-Variante besonders an. Um mögliche Fehler in der Hardware-Implementation besser erkennen zu können, wird auch die Gathering-Variante in Software umgesetzt.

Der in Abschnitt 3.3.2 beschriebene Umgebungsterm wird in dem Konzept und der Umsetzung nicht weiter betrachtet, da er ausschließlich einen vorübergehenden ästhetischen Aspekt hat. Weiterhin wird ein durch den Algorithmus gesteuertes adaptives Meshing nicht vorgenommen. Für einen Prototyp sollte es ausreichen vor Anwenden des Algorithmus eine angemessene Unterteilung bereitzustellen. Zuletzt wird eine einfache Interpolation der Patch-Lichtausstöße nach Vertex-Lichtausstößen umgesetzt. Während in der flach schattierten Darstellung besser zu erkennen ist, welche Farbwerte die einzel-

nen Patches wirklich erhalten, unterstützt die Interpolation maßgeblich die Ästhetik der Ergebnisse.

4.2 Hardware-Radiosity

Wie bereits in der Einleitung dieses Kapitels beschrieben soll die Hardware-Variante des Radiosity-Verfahrens als eine Kopie der Software-Variante angelegt und an geeigneten Stellen durch Code mit Hardwareunterstützung ausgetauscht werden. Die Berechnung der Formfaktoren muss nicht mit Hardwareunterstützung umgesetzt werden. Laut dem originalen Paper über das Progressive Refinement Radiosity Verfahren werden die Formfaktoren bei jedem Schritt neu berechnet um den Platzbedarf zu reduzieren [9]. Für den Prototyp reicht es aber aus die Formfaktoren initial berechnen zu lassen. Ausschlaggebend ist die Vergleichbarkeit der Radiosity-Implementationen.

Coombe und Harris fiel 2005 auf, dass der Shooting-Ansatz ungünstig mit Grafikhardwareunterstützung zu realisieren ist [14]. Grafikhardware bietet massiv parallele Rechenleistung, die davon profitiert Datenmengen parallel durch denselben Code zu verarbeiten. Zwar ist dies beim Shooting-Ansatz auch möglich, doch verteilen sich die Schreiboperationen ausgehend von einem Patch in die komplette Umgebung [14]. Wenn nun mehrere Patches parallel verarbeitet werden, können sie sich dabei in die Quere kommen. Der Gathering-Ansatz bietet sich hier besser an. Durch das Sammeln des eintreffenden Lichts und aktualisieren des eigenen Lichtausstoßes kommen sich die Kernels nicht in die Quere und schreiben in ihre eigenen Speicher. Aufgrund der Parallelisierung kann - genügend Hardware-Shader vorausgesetzt - eine komplette Iteration mit einem Hardwareaufruf berechnet werden.

5 Umsetzung

Nachdem die konzeptionellen Grundlagen festgelegt wurden, beschäftigt sich dieses Kapitel mit der Umsetzung der bei dieser Arbeit entstandenen Implementationen für das Radiosity-Verfahren. Zuerst wird die Einrichtung des Projekts geschildert. Danach werden allgemein verwendete Datenstrukturen vorgestellt, die nicht direkt an der Radiosity-Implementation beteiligt sind. Anschließend wird vorgestellt welche Umgebungen umgesetzt wurden die das Verfahren berechnen soll. Danach wird die Implementation des Radiosity-Verfahrens in Software erläutert und welche konkreten Schwierigkeiten sich dabei ergeben haben. Daran anschließend wird die Umsetzung von ausgewählten Teilen der Software-Radiosity-Lösung mit Grafikhardwareunterstützung nahegelegt. Abgeschlossen wird dieses Kapitel mit einem Einblick in die Arbeitsweise in der das Projekt umgesetzt wurde.

5.1 Aufbau des Projekts

Das Projekt wurde in C# mit der Community-Edition von Microsoft Visual Studio 2015 umgesetzt. Die Projektmappe besteht aus vier Modulen. Eine Übersicht der Module und Komponenten ist Abbildung 5.8 auf Seite 52 zu entnehmen. Das Modul „CG_Framework“ stellt die grundlegenden APIs für das Anzeigefenster und Rendering zur Verfügung. Der Code wurde vor der Implementation vom Betreuer bereitgestellt und nur minimal angepasst. „CUDA_Runtime_8.0“ beinhaltet den Code welcher durch die Grafikhardware ausgeführt wird. Auf die Einrichtung dieses Moduls wird in Abschnitt 5.4.5 näher eingegangen. „RealtimeRadiosity“ enthält sämtlichen eigens geschriebenen C#-Code für dieses Projekt. Es hat eine Abhängigkeit zu den beiden vorher genannten Modulen. In „RealtimeRadiosityTest“ wurde der Code zum Testen des Moduls „RealtimeRadiosity“ untergebracht und hat folglich eine Abhängigkeit zu diesem. Auf die Komponenten der Module in den folgenden Abschnitten eingegangen.

5.2 Datenstrukturen

Dieser Abschnitt beschreibt die in diesem Projekt implementierten Datenstrukturen die nicht direkt an der Radiosity-Implementation beteiligt sind, als Grundlagen aber unerlässlich sind. Das betrifft die Repräsentation von Polygonnetze, das Rendering und den Szenengraphen.

5.2.1 Halbkantendatenstruktur

Das Radiosity-Verfahren basiert auf der Berechnung des Lichtausstoßes von Oberflächen. Typischerweise werden Oberflächen in der Computergrafik als Facetten von Objekten realisiert, die durch Polygonnetze repräsentiert werden. Im Rahmen dieses Projekts wurden zur Modellierung von Polygonnetzen die Schnittstellen IMesh für das Netz, IFacet für die Facetten und IVertex für die Eckpunkte angelegt. Eine Facette entspricht hierbei einem Radiosity-Patch. So finden sich in den Facetten auf Radiosity bezogene Attribute wie der Lichtausstoß, die Leuchtkraft oder die Reflektivität. Genaueres kann dem Klassendiagramm in Abbildung 5.9 auf Seite 53 entnommen werden.

Um die Schnittstellen zu implementieren wurde die Halbkantendatenstruktur eingesetzt [7]. Sie zeichnet sich dadurch aus Nachbarschaftsanfragen besonders schnell beantworten zu können [1].

In der Halbkantendatenstruktur referenzieren sich Facetten, Eckpunkte und Halbkanten gegenseitig. Die Halbkanten sind dabei die wichtigste Komponente. Durch sie kann die Nachbarschaft von Facetten und Eckpunkten traversiert werden. So hält eine Halbkante jeweils eine Referenz auf ihren Nachfolger, ihren Vorgänger, ihre gegenüberliegende Halbkante, ihren Startknoten und die Facette an der sie liegt. Facetten und Eckpunkte referenzieren jeweils nur eine beliebige anliegende Halbkante. Abbildung 5.1 stellt diese Relationen für eine Halbkante grafisch dar.

Die implementierte Datenstruktur bietet auch einige Funktionen zum Verändern des Netzes an. So können Eckpunkte und Facetten hinzugefügt oder entfernt werden. In Bezug auf das Radiosity-Verfahren ist die Möglichkeit das Polygonnetz unterteilen zu können interessant. Der Algorithmus beschränkt sich hierbei auf Facetten mit drei oder

[1] `https://de.wikipedia.org/wiki/Polygonnetz#Laufzeitvergleich` - Abgerufen: 27. März 2018

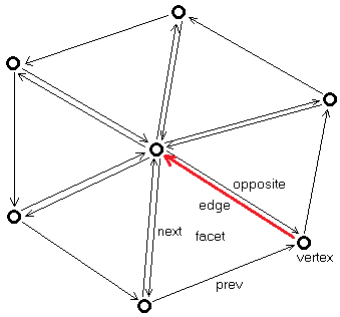

Abbildung 5.1: Ein Polygonnetz modelliert mit der Halbkanten-Datenstruktur. Es werden die Relationen der markierten Kante dargestellt.

vier Eckpunkten. Sie können besonders einfach unterteilt werden, wie Abbildung 5.2 zeigt.

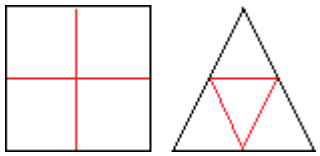

Abbildung 5.2: Unterteilung von einem Quadrat und einem Dreieck in jeweils vier Quadrate bzw. Dreiecke.

5.2.2 Rendering

Zum Rendern wird OpenGL eingesetzt, das durch eine Compilezeit-Abhängigkeit zu OpenTK bereitgestellt wird. OpenTK nimmt dabei die Rolle eines low-level Wrappers für OpenGL ein [2]. Um Objekte mit OpenGL zu zeichnen werden sogenannte Vertex Buffer Objekte aufgebaut [3]. Dies sind Puffer, zum Beispiel für Positions-, Normalen-, Farb- oder Texturinformationen von Eckpunkten. Diese Puffer werden im Grafikspeicher angelegt. Wenn die Puffer im Grafikspeicher vorliegen, können sie direkt von der Grafikhardware gerendert werden [4] und brauchen nur bei Änderungen angepasst werden.

Im Rahmen dieses Projekts werden die unterschiedlichen Vertex Buffer Objekte auf

[2] `https://github.com/opentk/opentk` - Abgerufen: 27. März 2018
[3] `https://www.khronos.org/opengl/wiki/Vertex_Specification` - Abgerufen: 27. März 2018
[4] `https://en.wikipedia.org/wiki/Vertex_buffer_object` - Abgerufen: 27. März 2018

C#-Seite durch eine VertexBufferObject genannte Klasse gebündelt. Sie besteht aus einer Liste von Render-Vertices, welche die Positions-, Normalen-, Farb- und Texturinformationen beinhalten. Die Klasse kapselt somit das Anlegen und Befüllen der Puffer im Grafikspeicher. Durch die Angabe eines PrimitiveType wird festlegt, wie die Puffer zu zeichnen sind [5]. Das Klassendiagramm in Abbildung 5.3 stellt dies noch einmal grafisch dar.

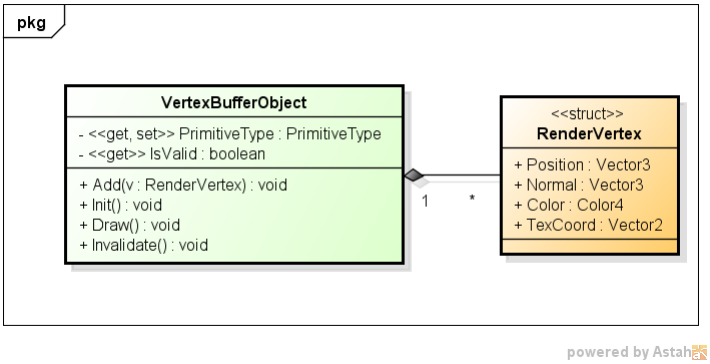

Abbildung 5.3: Die am Rendering beteiligten Klassen

5.2.3 Szenengraph

Zur Organisation und Strukturierung der Darstellungen von Polygonnetzen wurde eine Szenengraphstruktur geschaffen. Für die Debugbarkeit ist es von großem Interesse, unterschiedliche Aspekte eines Polygonnetzes visualisieren zu können. Diese Aspekte sollen nicht immer gleichzeitig sichtbar sein, sondern nach Bedarf an- oder abschaltbar. Die interessanten Aspekte für ein Polygonnetz umfassen dabei folgende Punkte:

- Die berechneten Intensitäten (Farben) mittels flat-shading und vertex-shading
- Der aktuelle Lichtausstoß, Leuchtkraft und Reflektivität der Patches
- Die Facetten- und Eckpunktnormalen
- Das Gitternetz
- Eventuelle Außenkanten bei offenen Polygonnetzen

Für diese Aspekte wurden jeweils Szenengraphknoten implementiert. Sie können dem Klassendiagramm in Abbildung 5.10 auf Seite 54 entnommen werden. Zusätzlich wurde

[5] `https://www.khronos.org/opengl/wiki/Primitive` - Abgerufen: 27. März 2018

mit der Klasse MeshNode ein Knoten eingeführt, der die anderen Knoten in sich bündelt. Er ermöglicht das bereits erwähnte An- und Abschalten der unterschiedlichen Aspekte. Hierbei muss unterschieden werden zwischen Knoten, die das gesamte Polygonnetz oder nur Aspekte davon darstellen. Während Knoten die das Polygonnetz darstellen nicht gleichzeitig angezeigt werden sollten, ist das für Knoten die nur Aspekte wie Eckpunktnormalen darstellen durchaus zulässig. Das erklärt die unterschiedliche Handhabung der von AbstractShadedMeshNode erbenden Klassen gegenüber denen, die direkt von AbstractMeshVBONode erben. Dieser Aufbau ermöglicht das an- und abschalten von Knoten die Aspekte darstellen und das umschalten zwischen unterschiedlichen Repräsentationen des Polygonnetzes, im Diagramm Shadings genannt.

Eine weitere Fähigkeit die jeder Knoten mit sich bringt, ist die Möglichkeit sein(e) VertexBuffer-Objekt(e) zu aktualisieren. Bei der Radiosity-Berechnung verändern sich die Farbwerte der Patches, was auch interaktiv dargestellt werden soll. Die Refresh-Methode kann nach jedem Schritt in der Radiosity-Berechnung aufgerufen werden, um so die Darstellung zu aktualisieren.

Im Rahmen dieses Projekts wurde weiterhin eine Szene angelegt, die für eine Umgebung sämtliche auf Radiosity bezogenen Funktionen bereitstellt. Sie erhält als Eingabe ein Polygonnetz das die Umgebung repräsentiert und eine Implementation des Radiosity-Verfahrens. In ihrem Szenengraphen wird der MeshNode für das übergebene Polygonnetz eingetragen. Weiterhin stehen einige Knoten zum visuellen Debuggen zur Verfügung. So gibt es ein Koordinatenkreuz zur besseren Orientierung und einen Hemicube der auf unterschiedliche Facetten gesetzt werden und darstellen kann, wie andere Facetten durch seine Frusta geschnitten werden.

Die Szene stellt einige Aktionen zur Verfügung, die per Tastendruck durchgeführt werden können. Die Tabelle in Abbildung 5.4 listet diese auf. Besonders interessant hierbei ist das Unterteilen des Polygonnetzes und die Radiosity-Berechnung. Durch das Unterteilen kann jedes Polygonnetz beliebig detailliert aufgelöst werden. Dadurch können beliebig feine Ergebnisse der Radiosity-Berechnung betrachtet werden. Die Radiosity-Berechnung wird in einem eigenen Thread durchgeführt, sodass die Anwendung weiter bedient werden kann.

Taste	Aktion
C	Schaltet die Darstellung eines Koordinatenkreuzes am Ursprung an/aus
M	Schaltet die Darstellung des schattierten Polygonnetzes an/aus
N	Schaltet zwischen den verschiedenen Darstellungen des Polygonnetzes um
W	Schaltet die Darstellung des Wireframe an/aus
F	Schaltet die Darstellung der Facetten-Normalen an/aus
V	Schaltet die Darstellung der Vertex-Normalen an/aus
B	Schaltet die Darstellung der Außenkanten an/aus
8	Schaltet eine Darstellung des Polygon-Clippings mit dem Hemicube an/aus
9	Schaltet die Darstellung eines Hemicubes an/aus
0	Erzeugt ein Bild aus der Sicht des Hemicubes und speichert es im Dateisystem
.	Verschiebt den Hemicube auf ein anderes Patch
,	Wählt ein anderes Patch aus, das vom Hemicube aus geclipped werden soll
-	Schaltet zwischen den Hemicube-Oberflächen zum Clipping um
S	Unterteilt alle Patches des Polygonnetzes einmal
D	Setzt das Polygonnetz zurück
E	Schaltet zwischen den Radiosity-Verfahren um
T	Schaltet den Formfaktor-Cache an/aus
R	Führt die Radiosity-Berechnung durch oder bricht sie ab

Abbildung 5.4: Aktionen der Radiosity-Szene

5.3 Umgebungen

Im Rahmen dieser Arbeit wurden verschiedene Umgebungen modelliert die unterschiedliche Zwecke erfüllen. Sie werden im Folgenden vorgestellt und können Abbildung 5.11 auf Seite 55 entnommen werden.

5.3.1 Einfacher Raum

Als erste Umgebung wurde ein einfacher Raum modelliert der als Minimalbeispiel einfach zu debuggen sein sollte. Er hat eine weiße Decke als Lichtquelle, eine rote, eine grüne und zwei hellgraue Wände sowie einen hellgrauen Boden. Die rote und grüne Wand sollen dabei zu den Effekten des Farbblutens führen.

5.3.2 Schatten Raum

Um eine Umgebung zu realisieren in der Schattenwurf auftritt, wurde eine Kopie der ersten Umgebung um einen hellgrauen Boden auf halber Höhe ergänzt. Der Bereich

unter diesem Boden kann nicht direkt von der Lichtquelle beleuchtet werden. Hier soll gezeigt werden, dass auch die indirekte Beleuchtung funktioniert.

5.3.3 Box Raum

Zuletzt wurde ein Raum modelliert in dem ein paar farbige Boxen stehen. Er ist einer Umgebung von Shao und Badler nachempfunden [29]. Im Gegensatz zu den vorherigen Umgebungen ist die Lichtquelle nicht die gesamte Decke, sondern eine Lampe an der Decke deren Fläche ein Viertel der Decke beträgt. Mit der Umgebung sollen Effekte wie Farbbluten, Schatten und indirekte Beleuchtung gezeigt werden.

5.4 Radiosity

Dieser Abschnitt beschreibt die Implementation der drei Varianten des Radiosity-Verfahrens. Die Klassendiagramme in den Abbildungen 5.13 und 5.12 geben einen Überblick über die Implementierten Klassen. Die Implementationen teilen sich viele Gemeinsamkeiten, wie zum Beispiel die Berechnung der Formfaktoren die im Folgenden nahegelegt wird. Auf die Unterschiede wird in den Abschnitten 5.4.3, 5.4.4 und 5.4.5 eingegangen.

5.4.1 Formfaktorberechnung

Für die Berechnung der Formfaktoren wurde eine Schnittstelle angelegt. Sie bietet eine Methode zum festlegen einer Umgebung und eine Methode zum Berechnen der Formfaktorzeile eines Patches der Umgebung an. Der Einsatz einer Schnittstelle ermöglicht den konkret eingesetzten Algorithmus zur Formfaktorberechnung auszutauschen.

Hemicube

Wie bereits im Konzept beschrieben wurde, soll der Hemicube-Algorithmus zur Formfaktorberechnung eingesetzt werden. Die Implementation orientiert sich dabei an der C-Implementation von Ashdown. Das Klassendiagramm in Abbildung 5.12 auf Seite 56 gibt eine Übersicht über die dafür angelegten Klassen. Der Hemicube implementiert hierbei die Formfaktorberechner-Schnittstelle. Er stellt einen Konstruktor zur Verfügung dem die Auflösung des Hemicubes übergeben wird. Der Konstruktor berechnet die Δ-Formfaktoren und erzeugt einen Zellpuffer der den Hemicube-Pixeln einer Hemicube-

Seite entspricht. Weiterhin implementiert der Hemicube die Methode zum Berechnen der Formfaktorzeile zu einem Patch. Sie kann Algorithmus 5.1 entnommen werden.

In der Implementation von Ashdown wird das Formfaktor-Array nicht als Rückgabeparameter zurückgegeben, sondern als Übergabeparameter übergeben und die Inhalte überschrieben. Auf diesem Weg kann das Array wiederverwendet werden und muss nicht mehrfach erzeugt und wieder zerstört werden [1]. Für die Implementation in diesem Projekt wird die Wiederverwendung des Formfaktor-Arrays in dem Hemicube gekapselt. Ein Aufrufer erhält mit dem Aufruf der ComputeFormFactors-Methode eine Referenz auf das intern gehaltene Array. Durch die Wiederverwendung sollte ein Aufrufer die Referenz nur solange wie nötig halten, da die Werte sonst überschrieben werden könnten.

Algorithmus 5.1: ComputeFormFactors

```
Input: Patch i
foreach F_ji in F_i do
    F_ji ← 0
end Setze das Formfaktorarray zurück
Orientation ← Bestimme die Orientierung des Hemicubes anhand des Patches i
foreach Seite in Hemicube-Seiten do
    CellBuffer.Reset()
    Matrix ← Bilde Transformationsmatrix für aktuelle Seite und Orientierung
    foreach Patch j in Environment do
        if i ≠ j ∧ ¬Clipper.BackFaceCull(j, Orientation.Origin) then
            ClippedPolygon ← Clipper.Clip(j, Matrix)
            Scanner.Scan(id, ClippedPolygon, CellBuffer)
        end
    end
    CellBuffer.SumDeltas(Seite, F_i, Deltas)
end
```

Da das Formfaktor-Array wiederverwendet wird, müssen am Anfang alle Einträge auf 0 zurückgesetzt werden. Danach wird die Orientierung des Hemicubes aus dem Patch bestimmt um den Hemicube auf diesem Patch zu platzieren. Nun wird für jede Seite die Projektion aller anderen Patches der Umgebung auf die Hemicube-Seiten vorgenommen. Dafür wird zunächst der Zellpuffer zurückgesetzt und eine Transformationsmatrix gebildet, mit deren Hilfe die Eckpunkte der Patches in den View-Space der Hemicube-Seite transformiert werden. Nun wird jedes andere Patch der Umgebung zugeschnitten und anschließend durch den Scan-Conversion-Algorithmus auf die Seite projiziert. Die Er-

gebnisse werden in den Zellpuffer geschrieben. Zuletzt werden die Δ-Formfaktoren der Patches, anhand der getroffenen Pixel der Seite, durch den Zellpuffer aufsummiert und in das Formfaktor-Array geschrieben.

Es muss berücksichtigt werden, dass der Hemicube nicht Thread-sicher ist. Mehrere Threads sollten die ComputeFormFactors-Methode daher nicht konkurrierend verwenden. Das liegt daran, dass der Zellpuffer und die Implementation der Scan-Conversion mutable Datentypen verwenden. Auch die Wiederverwendung des Formfaktor-Arrays steht einer Nutzung in mehreren Threads entgegen. Nachdem nun der Ablauf erklärt wurde, wird im Folgenden die Implementation der Komponenten des Hemicubes näher erläutert.

Delta-Formfaktoren

Die Δ-Formfaktoren wurden unter Rücksichtnahme auf die in Abschnitt 4.1.1 unter Delta-Formfaktoren beschriebenen Symmetrien implementiert um Speicherplatz zu sparen. Dafür werden zwei getrennte zweidimensionale Array verwendet. Eins für die Formfaktoren der Oberseite und eins für die Außenseiten des Hemicubes [1]. Die Anzahl der Elemente jeder Zeile entspricht dabei ihrem Index in dem Array plus eins. Das Array weist also eine treppenartige Form auf die den in Abbildung 4.1 markierten minimal benötigten Δ-Formfaktoren entspricht. Um den berechneten Δ-Formfaktor eines Pixels zu erhalten wurde für die beiden Arrays eine Methode bereitgestellt, die den Δ-Formfaktor eines Pixels anhand seiner Zeile und Spalte aus dem jeweiligen Array ermittelt. Dafür müssen die Zeile und Spalte wenn nötig horizontal, vertikal oder diagonal gespiegelt werden. Werte die außerhalb der Auflösung liegen werden nicht angenommen.

Zellpuffer

Die nächste Komponente ist der Zellpuffer, welcher vereinfacht gesagt das Bild repräsentiert das auf eine Hemicube-Seite gezeichnet wird. Er hat dieselbe Auflösung wie der Hemicube. Ein einzelner Eintrag enthält die zugeordnete Facette sowie ihre Entfernung zum Hemicube-Ursprung und repräsentiert einen Hemicube-Pixel. Der Zellpuffer stellt eine Methode zum aktualisieren eines Pixels bereit der die ID eines Patches und seine Entfernung zum Hemicube-Ursprung übergeben wird. Verweist der Pixel bereits auf ein anderes Patch kann anhand der Entfernung entschieden werden, welches Patch der Pixel referenzieren soll.

Weiterhin wird eine Methode zum aufsummieren der Deltaformfaktoren angeboten. Sie

iteriert über alle Pixel, summiert die Δ-Formfaktoren pro referenziertem Patch auf und schreibt sie in das übergebene Formfaktor-Array. Außerdem wird eine Methode zum Zurücksetzen des Puffers angeboten.

Der Zellpuffer entspricht nur einer Hemicube-Seite. Diesen Ansatz hat Ashdown gewählt um den Speicherbedarf zu reduzieren [1]. Es wäre auch möglich einen separaten Zellpuffer für jede Seite einzusetzen. Im Rahmen dieses Projekts wurde jedoch der ressourcensparendere Ansatz von Ashdown eingesetzt.

Polygon Clipping

Das im Konzept unter Abschnitt 4.1.1 beschriebene Clipping wird durch die Clipper-Klasse umgesetzt. Sie referenziert ein Frustum das aus fünf Clipping-Planes im View-Space des Hemicubes besteht. Eine Clipping-Plane für jede Seite des Frustums und eine Near-Clipping-Plane. Die Clipper-Klasse bietet eine Methode an um zu ermitteln ob ein Patch vom Hemicube-Ursprung aus gesehen werden kann. So kann geprüft werden, ob das Zuschneiden eines Patches und die nachfolgende Projektion auf den Hemicube überhaupt notwendig ist. Zum Zuschneiden eines Patches wird die Clip-Methode angeboten. Sie extrahiert die Eckpunkte des übergebenen Patches, transformiert sie in den View-Space des Hemicubes und führt dann den im Konzept beschriebenen Sutherland-Hodgman-Algorithmus aus.

Im Gegensatz zu Ashdown wird bei der Transformation in den View-Space nur die Rotation und Translation berücksichtigt, nicht jedoch die Projektion und Normalisierung. Nach eigenem Ermessen sind letztere in der Scan-Conversion besser aufgehoben, da sie sich mit der Projektion auf die Hemicube-Seiten beschäftigt. Das bedeutet aber auch, dass in dieser Implementation zwei Matrixmultiplikationen pro Eckpunkt pro Patch durchgeführt werden müssen. Die Matrixmultiplikation dagegen nur in der Scan-Conversion durchzuführen hätte zur Folge, dass das Clipping im World-Space geschehen müsste und die Clipping-Planes für jedes Patch erneut im World-Space definiert werden müssten.

Scan Conversion

Die letzte Komponente ist die Scan-Conversion die wie im Konzept in Abschnitt 4.1.1 die Projektion der zugeschnittenen Patches auf die Hemicube-Pixel realisiert. Die Implementation entspricht der Beschreibung aus dem Konzept in Abschnitt 4.1.1 und damit der

Implementation von Ashdown. Die ScanConversion-Klasse verfügt über ein entsprechend der Auflösung dimensioniertes Array von Scanlines das wiederverwendet wird um der ständigen Neuerzeugung dieses Arrays zu entgehen. Das führt allerdings dazu, dass diese Klasse wie weiter oben bereits angemerkt mutabel und nicht Thread-sicher ist. Weiterhin ist die Klasse dafür zuständig die Einträge des Zellpuffers entsprechend dem Ergebnis der Scan-Conversion zu aktualisieren.

Darstellung der Projektion

Um bessere Einsicht in die Projektion auf die Hemicube-Seiten zu erhalten, wurde die Formfaktorberechner-Schnittstelle um eine weitere Methode ergänzt. Sie erzeugt ein Bild aus der Sicht des Patches. Dadurch kann nachgeschaut werden, ob die Projektion zufriedenstellend funktioniert hat. Ein Beispiel eines solchen Bildes ist in Abbildung 5.5 zu sehen.

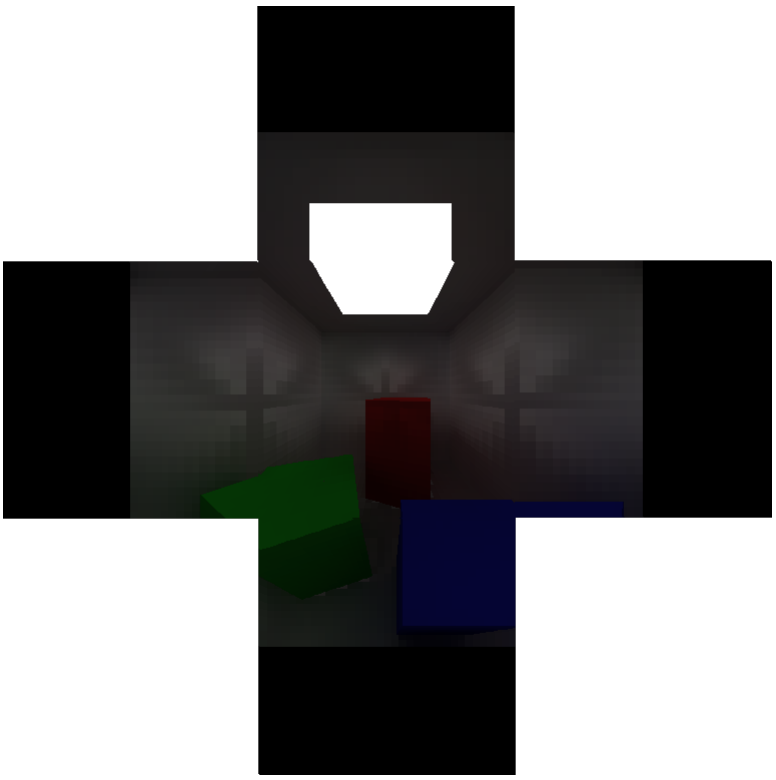

Abbildung 5.5: Sicht eines Patches in dem Box Raum durch einen Hemicube mit einer Auflösung von 256x256 Pixeln

Formfaktoren cachen

Um Formfaktoren nicht mehrfach berechnen zu müssen wurde ein Cache eingeführt. Er implementiert die Schnittstelle für einen Formfaktorberechner und erhält eine Referenz auf einen Delegaten. Der Delegat ist dabei der eigentliche Formfaktorberechner und wird benutzt um die Formfaktoren noch nicht im Cache vorhandener Patches zu berechnen. Zusätzlich verfügt der Cache über ein zweidimensionales Array für die Formfaktoren. Die vom Delegaten berechneten Formfaktoren werden dann in dem Array gecached. Das Diagramm in Abbildung 5.6 zeigt diesen Aufbau.

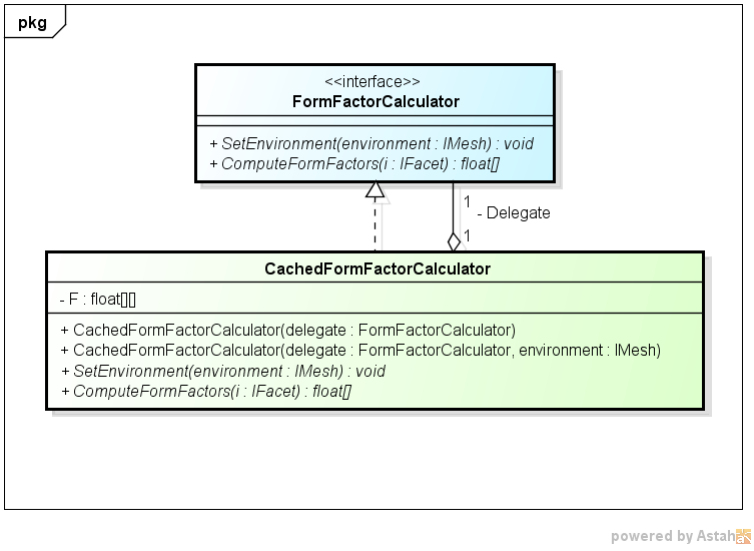

Abbildung 5.6: Der Formfaktor-Cache

Dieser nach dem Proxy-Pattern[6] umgesetzte Ansatz ermöglicht den einfachen Wechsel zwischen Cache und keinem Cache. Außerdem lässt sich der Cache so elegant der Radiosity-Implementation hinzufügen ohne den Algorithmus selbst dafür anzupassen.

Der Cache bietet einen Konstruktor für die direkte Initialisierung und einen für die Initialisierung bei Bedarf an. Das entspricht den Prinzipien der eager Initialization und lazy Initialization. Bei der eager Initialisierung wird die komplette Formfaktormatrix bereits bei Aufruf des Konstruktors berechnet. Bei der lazy Initialisierung erst, wenn der Radiosity-Algorithmus die Formfaktoren benötigt. Im Rahmen dieses Projekts hat

[6] `http://www.oodesign.com/proxy-pattern.html` - Abgerufen: 27. März 2018

sich die lazy Variante bewährt, da so nicht die komplette Formfaktormatrix berechnet werden muss, bevor die Berechnung durchgeführt werden kann. Ändert sich die Anzahl der Patches der übergebenen Umgebung, so wird der Cache wieder zurückgesetzt und wird erneut nach Bedarf aufgebaut.

5.4.2 Progressive Refinement Radiosity

Auch für das Progressive Refinement Radiosity Verfahren wurde eine Schnittstelle angelegt. Über sie können Listener[7] registriert werden, die bei dem Abschluss einer Iteration bzw. eines Schrittes benachrichtigt werden. So kann später die Anzeige nach Bedarf aktualisiert werden. Weiterhin ist es möglich die Toleranzgrenze festzulegen bis zu der das Verfahren berechnen soll und die maximale Anzahl an Iterationen nach denen die Berechnung abgebrochen wird, wenn bis dahin die Toleranzgrenze nicht erreicht wurde. Zuletzt bietet die Schnittstelle eine Methode zum Durchführen der Berechnung. Bei Abschluss der Berechnung wird von ihr ein Statistik-Objekt zurückgegeben, das unter anderem die benötigte Zeit beinhaltet.

Wie dem Klassendiagramm in Abbildung 5.13 auf Seite 57 zu entnehmen ist, hat sich für die Gemeinsamkeiten der Implementationen eine abstrakte Klasse gebildet. So benötigt jede Implementation einen Formfaktorberechner. Weiterhin gibt die Klasse den für die Software-Implementationen gleichen Ablauf mittels Template-Methoden[8] vor.

Algorithmus 5.2 zeigt die wichtigsten Teile der Calculate-Methode. So werden zunächst die Lichtausstöße B_i der Patches mit ihrer Leuchtkraft E_i initialisiert. Dann wird die initiale Menge des ungesendeten Lichts der Umgebung berechnet. Anhand dieses Wertes kann später der aktuelle Grad der Konvergenz berechnet werden, der initial mit 1 beginnt und im Laufe der Berechnung gegen 0 konvergiert. Anschließend werden solange Iterationen durchgeführt, bis die Lösung konvergiert ist, oder die maximale Anzahl an Iterationen überschritten wurde.

Eine Iteration verarbeitet in der Regel alle Patches der Umgebung. Dies geschieht indem zunächst die Patches aus der Umgebung extrahiert werden. Anschließend wird für jedes Patch ein Schritt durchgeführt. Dafür wird das gewählte Patch zunächst aus der extrahierten Liste entfernt, damit es in dieser Iteration nicht erneut verarbeitet

[7] `http://www.oodesign.com/observer-pattern.html` - Abgerufen: 27. März 2018
[8] `http://www.oodesign.com/template-method-pattern.html` - Abgerufen: 27. März 2018

Algorithmus 5.2: Calculate

```
Input: Environment
foreach Patch i in Environment do
  | B_i ← E_i
end Initialisiere die Lichtausstöße der Patches
initialFlux ← Menge des ungesendeten Lichts in Environment
convergence ← 1
iteration ← 0
while convergence >= ConvergenceTolerance ∧ iteration < MaxIterations do
  | DoIteration(Environment, initialFlux)
  | iteration++
end
```

wird. Danach werden die Formfaktoren zu dem ausgewählten Patch berechnet. Wenn die Formfaktoren vorhanden sind wird ein Schritt in der Radiosity-Berechnung durchgeführt. Nach dem Schritt wird die aktuelle Menge des ungesendeten Lichts in der Umgebung berechnet. Daraus wird zuletzt der aktuelle Grad der Konvergenz berechnet, indem die aktuelle Menge des ungesendeten Lichts durch die Initiale geteilt wird. Sollte die Konvergenz erreicht werden bevor für jedes Patch ein Schritt durchgeführt wurde, so endet die Iteration und damit der Algorithmus ebenfalls. Algorithmus 5.3 stellt diesen Ablauf nochmal in Pseudocode dar.

Algorithmus 5.3: DoIteration

```
Input: Environment, initialFlux
patches ← ExtractFacets(Environment)
while convergence >= ConvergenceTolerance ∧ patches.Count > 0 do
  | Patch i ← patches[0]
  | patches ← patches \ i
  | F_i ← alle Formfaktoren nach Patch i
  | DoStep(i, Environment, F_i)
  | currentFlux ← Menge des ungesendeten Lichts in Environment
  | convergence ← currentFlux / initialFlux
end
```

Die Einführung von Iterationen ist eine Abweichung zu Ashdowns Implementation in der Schritte und Iterationen nicht unterschieden werden. Aus diesem Grund iteriert Ashdown vor jedem Schritt über alle Patches um das nächste zu verarbeitende Patch auszuwählen. Um einer weiteren Schleife über alle Patches vor jedem Schritt entgegenzuwirken wurde die ExtractFacets-Methode auf Iterationen-Ebene etabliert.

Nachdem nun der allgemeine Ablauf bekannt ist, kann auf die Unterschiede der Software-Implementationen eingegangen werden. Wie bereits erwähnt, gibt es zwei Methoden die dafür von Interesse sind. Zum Einen die Methode DoStep, sie dient der Radiosity-Berechnung eines Schrittes und unterscheidet sich folglich für die beiden Varianten. Zum Anderen die Methode ExtractFacets, mit deren Hilfe die für die Iteration zu verarbeitenden Patches aus der Umgebung ausgewählt und sortiert werden können. Standardmäßig wird hier die native Sortierung aus dem Polygonnetz übernommen. Erbende Klassen können aber auch andere Implementationen bereitstellen.

5.4.3 Shooting Radiosity

Die implementierte Shooting-Variante betrachtet in einer Iteration nur die Patches, die einen größeren Lichtanteil haben als der Durchschnitt in der Umgebung. Dadurch wird der Verlauf des Lichts nachempfunden. Die erste Iteration berechnet dann die direkte Beleuchtung, die zweite Iteration die indirekte Beleuchtung mit einer Reflexion, die dritte Iteration mit zwei Reflexionen und so weiter. Die Patches werden absteigend nach ihrem Lichtanteil sortiert um hellere Sender möglichst früh zu verarbeiten. Auf diesem Weg entwickelt sich die Beleuchtung elegant und die Berechnung konvergiert schneller [9].

Die DoStep-Methode der Shooting-Variante schießt den Lichtausstoß des ausgewählten Patches auf alle anderen Patches der Umgebung. Dafür muss zunächst mit Formel 2.4 der wechselseitige Formfaktor berechnet werden [9, 2], da nur der Formfaktor von allen Patches j zu Patch i vorliegt, zum schießen aber die Formfaktoren von Patch i zu allen Patches j benötigt werden. Dann kann das Patch i auf dem anderen Patch j eintreffende Licht berechnet werden, um es seinem Lichtausstoß B_j hinzuzufügen. Nachdem das Patch i seinen Lichtausstoß auf alle anderen Patches verteilt hat, wird sein eigener Lichtausstoß auf 0 gesetzt. Algorithmus 5.4 fasst diesen Ablauf nochmal zusammen.

5.4.4 Gathering Radiosity

Die Gathering-Variante verarbeitet in einer Iteration alle Patches der Umgebung aufsteigend sortiert nach ihrem ungesendeten Lichtanteil. Das heißt es werden immer die Patches mit dem geringsten ungesendeten Lichtanteil als nächstes verarbeitet. Da in der Gathering-Variante der Lichtausstoß des verarbeiteten Patches mit dem gesammelten

Algorithmus 5.4: DoStep für die Shooting-Variante

Input: Patch i, Environment, F_i
foreach *Patch j in Environment* **do**
  $F_{ji} \leftarrow F_i[j]$
  if $i \neq j \wedge F_{ij} < 0$ **then**
    $F_{ij} = F_{ji} * A_i/A_j$
    $B_j = B_j + \rho_j * B_i * F_{ji}$
  end
end
$B_i \leftarrow 0$

Licht überschrieben wird, verhindert diese Sortierung, dass der Lichtausstoß von helleren Patches überschrieben wird, bevor dunklere Patches den Anteil aufgenommen haben. Weiterhin kann die Menge der zu verarbeitenden Patches in einer Iteration nicht wie in der Shooting-Variante reduziert werden, da mit jedem Schritt immer nur ein Lichtausstoß aktualisiert wird.

Die DoStep-Methode der Gathering-Variante sammelt das eintreffende Licht von allen anderen Patches zu dem betrachteten Patch. Anders als bei der Shooting-Variante liegen hier bereits die richtigen Formfaktoren vor und sie können direkt verwendet werden. Bevor die Summe den Lichtausstoß des Patches ersetzen kann muss sie mit der Reflektivität des Patches multipliziert werden. Algorithmus 5.5 stellt diesen Ablauf nochmal als Pseudocode dar.

Algorithmus 5.5: DoStep für die Gathering-Variante

Input: Patch i, Environment, F_i
$\Delta B_i \leftarrow 0$
foreach *Patch j in Environment* **do**
  $F_{ji} \leftarrow F_i[j]$
  if $i \neq j \wedge F_{ij} < 0$ **then**
    $\Delta B_i = \Delta B_i + B_j * F_{ji}$
  end
end
$B_i \leftarrow \rho_i * \Delta B_i$

5.4.5 Hardware Gathering Radiosity

Nachdem die Implementation der Software-Varianten nahegelegt wurde, beschreibt dieser Abschnitt die Entwicklung der mit Hardware unterstützten Radiosity-Implementation. Zur Umsetzung wurde NVidia CUDA ausgewählt. Während OpenCL als Open-Source Software und etablierter Standard die besten Voraussetzungen für eine breite Unterstützung bietet, so ermöglicht der proprietäre Ansatz von NVidia oftmals eine bessere Integration und Performance [21] [9].

Um die Grafikhardware für general-purpose Berechnungen mit CUDA nutzen zu können, muss das CUDA Toolkit installiert sein [10]. Außerdem muss der CUDA-Code aus C# heraus aufgerufen werden können um eine nahtlose Integration in das Projekt zu gewährleisten. Der nächste Abschnitt beschäftigt sich mit diesem Problem.

Einbindung von CUDA

Wie in Abschnitt 2.3.2 beschrieben, wird CUDA-Code in einem eigenen Dialekt der Programmiersprache C geschrieben. Um mit C# CUDA-Code auszuführen wird also ein Wrapper benötigt. ManagedCUDA[11] ist ein solcher Wrapper und findet in diesem Projekt Anwendung. Der CUDA-Code befindet sich, wie in Abschnitt 5.1 bereits erwähnt, in dem Modul „CUDA_Runtime_8.0“. Um den Code mittels ManagedCUDA aufrufen zu können müssen zunächst einige Konfigurationen an dem Modul vorgenommen werden [12].

Am wichtigsten ist die Konfiguration des Ausgabeverzeichnisses und -formats. Als Ausgabeverzeichnis wird ein Ordner „kernels“ im Modul „RealtimeRadiosity“ ausgewählt. Als Ausgabeformat und Compilation Type wird .ptx festgelegt. Parallel Thread Execution (PTX) ist eine low-level Assemblersprache in die der high-level CUDA-C Code kompiliert wird [25]. Nun wird der kompilierte CUDA-Code als .ptx-Dateien in dem kernels-Ordner abgelegt. Die .ptx-Dateien müssen noch dem RealtimeRadiosity-Modul hinzugefügt und für das Kopieren in das Ausgabeverzeichnis konfiguriert werden, damit sie zur

[9] `https://create.pro/blog/opencl-vs-cuda/` - Abgerufen: 27. März 2018
[10] `http://docs.nvidia.com/cuda/cuda-installation-guide-microsoft-windows/index.html` - Abgerufen: 27. März 2018
[11] `https://kunzmi.github.io/managedCuda/` - Abgerufen: 27. März 2018
[12] `https://github.com/kunzmi/managedCuda/wiki/Setup-a-managedCuda-project` - Abgerufen: 27. März 2018

Laufzeit gefunden werden können. Weitere hilfreiche Konfigurationen können dem Link in Fußnote 12 entnommen werden.

Beim Schreiben von CUDA-Code muss darauf geachtet werden, eine Reihe von Includes und Defines am Dateianfang anzugeben. Außerdem ist es hilfreich die geschriebenen Funktionen mit einem „extern C“-Scope zu umschließen. Dadurch wird sichergestellt, dass die geschriebenen Funktionen in den kompilierten .ptx-Dateien denselben Namen tragen. Auch diese Informationen können dem Link in Fußnote 12 entnommen werden. Nachdem nun die Vorbedingungen zur Nutzung von CUDA geklärt sind, beschäftigt sich der nächste Abschnitt mit den Grundlagen der CUDA-Programmierung.

Programmieren in CUDA

Um eine einfache Unterscheidung zwischen Code der auf der CPU bzw. der GPU ausgeführt wird haben sich zwei Begriffe etabliert. Die CPU und Arbeitsspeicher werden als „Host“-System bezeichnet, während für die GPU und ihren Speicher der Begriff „Device“ verwendet wird [28]. Eine Methode die auf dem Device ausgeführt wird bezeichnet man als Kernel [25, 28].

Ein Kernel unterscheidet sich von einer normalen Methode insofern, dass wenn er aufgerufen wird er parallel von einer bestimmten Anzahl an Threads ausgeführt wird [25]. In CUDA-C werden Kernel-Methoden mit dem Keyword ___global___ deklariert [25, 28]. Beim Aufruf des Kernels wird die Anzahl der Threads die diesen Kernel ausführen sollen festgelegt [25]. Dabei erhält jeder Thread eine individuelle Thread-ID, auf die im Kernel zugegriffen werden kann [25]. Threads werden in Blöcken gruppiert die vom selben Prozessor-Kern verarbeitet werden [25]. Dadurch unterliegt die Anzahl der Threads pro Block einer Obergrenze die in der Umsetzung berücksichtigt werden muss. Die Anzahl der Blöcke wird ebenfalls bei der Ausführung eines Kernels festgelegt. Da die Thread-ID nur innerhalb eines Blocks individuell ist, kann im Kernel auch auf die Block-ID und -Größe zugegriffen werden um eine global eindeutige Thread-ID zu berechnen [25]. Methoden die auf dem Device ausgeführt aber nicht von Host-Code aufgerufen werden sollen werden mit ___device___ deklariert.

Da der Host und das Device jeweils über ihren eigenen Speicher verfügen, müssen Daten die auf dem Device benötigt werden in den Device-Speicher und die berechneten Ergebnisse in den Host-Speicher kopiert werden. Dafür stehen auf der Device-Seite drei

unterschiedliche Speicher zur Verfügung: Der globale Speicher, der konstante Speicher und der Texturspeicher [25]. Der Host-Code ist dafür zuständig diese Speicher entsprechend zu allozieren, deallozieren und zwischen dem Host und Device zu transferieren [25]. Zusätzlich stehen Device-Threads noch ein lokaler Speicher und ein mit den anderen Threads eines Blocks geteilter Speicher zur Verfügung [25].

Implementation mit CUDA

Nachdem die allgemeinen Grundlagen für die Programmierung in CUDA bekannt sind, kann die in diesem Projekt entstandene Hardware-Implementation betrachtet werden. Sie besteht aus einem C# Teil und einem CUDA-C Teil. Der C#-Code wird durch die Klasse GPUProgressiveRefinementRadiosity realisiert (siehe Klassendiagramm in Abbildung 5.13 auf Seite 57). Wie im Klassendiagramm zu erkennen ist implementiert sie die Schnittstelle ProgressiveRefinementRadiosity und verfügt über einige Methoden der abstrakten Implementation die für die Hardware-Implementation zum Teil angepasst wurden. Während der Ablauf derselbe geblieben ist, wird eine Iteration nun mit jedem Hardwareaufruf parallel verarbeitet.

Da in jeder Iteration alle Schritte parallel berechnet werden, muss die Formfaktor-Matrix bereits vor der ersten Iteration berechnet werden. Die Formfaktoren ändern sich jedoch nicht, weshalb sie auch nur einmal in den Device-Speicher kopiert werden müssen. Die sich ändernden Daten die für eine Iteration benötigt werden müssen dagegen in jeder Iteration erneut in den Device-Speicher kopiert werden. Das betrifft die Patches und ihre Eigenschaften. Während in der Halbkantendatenstruktur die komplette Kenntnis über die Eckpunkte und Facetten vorliegen, so werden für eine Iteration ausschließlich die Reflektivität, der Lichtausstoß und der Flächeninhalt der Patches benötigt. Zusätzlich wird in dem Device-Speicher ein Array alloziert in das die vom Kernel berechneten Lichtausstöße geschrieben werden. Es braucht nur einmal vor der ersten Iteration alloziert werden, da der Kernel den Einträgen direkt ihre Werte zuweist.

Um einen CUDA-Kernel auszuführen muss die Anzahl der Threads und Blöcke festgelegt werden. Da in einer Iteration alle Patches mit einem Hardwareaufruf verarbeitet werden sollen, wird die Anzahl der Threads mit der Anzahl der Patches dimensioniert gedeckelt gegen die maximale Anzahl von Threads in einem Block des eingesetzten Grafikprozessors. Die Anzahl der Blöcke ergibt sich dann aus der Anzahl Patches durch die maximale Anzahl unterstützter Threads plus 1 falls in der Division ein Rest übrig bleibt. Ist dies

festgelegt, wird der Kernel mit den Zeigern auf die im Device-Speicher angelegten Arrays gestartet.

Der CUDA-Code enthält zwei Methoden. Eine mit ___device___ annotierte Helfermethode, die ähnlich der Software-Implementation das Sammeln der Radiosity-Werte für ein Patch umsetzt und damit einem Schritt entspricht. Die andere Methode ist der mit ___global___ deklarierte Kernel. Durch ihn wird die Helfermethode für jeden Thread mit einem anderen Patch aufgerufen. Abgesehen von der Ausführung auf der Grafikhardware unterscheidet sich der Code noch in einigen Details von der Software-Implementation. So werden die berechneten Radiosity-Werte nicht direkt den Lichtausstößen der Patches hinzugefügt, sondern zunächst in ein Array im Device-Speicher geschrieben. Diese Werte werden den Patch-Lichtausstößen erst vom Host-Code hinzugefügt. Ein weiterer Unterschied findet sich in der Formfaktor-Matrix. Sie wird dem Device-Speicher als flattened Array[13] übergeben, da das Kopieren von mehrdimensionalen Arrays einen Performance-Engpass darstellen kann [14].

5.5 Arbeitsweise

Dieser Abschnitt gibt Einblick in die Arbeitsweise in der das Projekt durchgeführt wurde. Das umfasst das Vorgehen während der Anforderungserhebung und der Projektdurchführung. Dabei wird auch auf den Einsatz eines Kanban-Boards eingegangen. Abgeschlossen wird dieses Kapitel mit der Vorgehensweise beim Test der entstandenen Software.

5.5.1 Iteratives Vorgehen

Vor Beginn der Arbeit wurde der Umfang mit dem Betreuer zusammen diskutiert und in einem Exposee festgehalten. Um eine grobe Vorstellung über den Aufwand zu erhalten wurden für das Exposee bereits abstrakte Aufgabenpakete definiert. Die dabei entstandenen Aufgabenpakete waren zunächst das Einrichten eines Projekts, die Implementation der CPU-Variante, die Implementation der GPU-Variante und das Anlegen von Demo-Szenen. Die Anforderungen an diese Aufgabenpakete wurden mit der Zeit immer genauer ausgearbeitet. Um dabei schnell in die Richtung von sichtbaren Ergebnissen zu kommen, genügte zunächst ein Überblick über die größten Probleme der

[13] `https://www.dotnetperls.com/flatten-array` - Abgerufen: 27. März 2018
[14] `https://stackoverflow.com/questions/44163375/c-sharp-managedcuda-2d-array-to-gpu` - Abgerufen: 27. März 2018

Aufgabenpakete. Die Details wurden deshalb erst während der Umsetzung analysiert. Durch diesen Ansatz konnten im periodischen Austausch mit dem Betreuer neu entdeckte Probleme schnell besprochen werden.

5.5.2 Kanban Board

Zur Verfolgung der Aufgaben und geleisteten Arbeitszeit sowie dem Erhalt des Überblicks wurde ein Kanban-Board eingesetzt. Es wurde während der gesamten Arbeit sehr intensiv genutzt um den Fortschritt transparent darzustellen und diente auch als Ablage für Ideen, Quellen und Referenzen. Das Board besteht aus den drei Spalten „Backlog“, „Doing“ und „Closed “. Während diese Unterteilung nur rudimentär ist, hat sie sich für nur einen Bearbeiter als ausreichend erwiesen. Um den Fortschritt einer Aufgabe zu vermerken, wurden alle Aufgaben während der Bearbeitung mit Unteraufgaben versehen die nach Abschluss abgehakt werden konnten. Die Abbildungen 5.14 (Seite 58) und 5.15 (Seite 59) zeigen das Board und eine ausgewählte Aufgabe.

5.5.3 Test

Der Test der entstandenen Softwareteile beschränkt sich größtenteils auf „visuelles Debuggen“. Damit ist gemeint, dass erzeugte Bilder mit einer Erwartungshaltung betrachtet wurden um Unstimmigkeiten aufzudecken. Um das visuelle Debuggen weiter zu unterstützen wurden Szenengraph-Knoten angelegt die relevante Aspekte visualisieren. In Abschnitt 5.2.3 wurde bereits beschrieben welche Aspekte eines Polygonnetzes dabei interessant sind. Zusätzlich wurde für das Clipping ein Szenengraph-Knoten angelegt, mit dessen Hilfe das Zuschneiden der Patches für den Hemicube nachvollziehbar dargestellt werden kann. Abbildung 5.7 zeigt ein Beispiel.

Automatisierte Tests stellen sich in diesem Umfeld als schwierig dar. Für sämtliche Teilschritte wie der Formfaktorberechnung, dem Zuschneiden von Patches, der Scan-Conversion und letztendlich der Berechnung des Radiosity-Verfahrens müssen Umgebungen modelliert werden in denen die gewünschten Auswirkungen zunächst definiert werden müssen. Deshalb beschränken sich die automatisierten Tests auf einfachere Teile. So sind die Berechnung der Fläche, des Mittelpunkts und der Normale von Facetten mit automatisierten Tests abgedeckt. Weiterhin wurden die in Abschnitt 4.1.1 beschriebenen Symmetrien der Δ-Formfaktoren und die in Abschnitt 2.2.1 geltenden Regeln

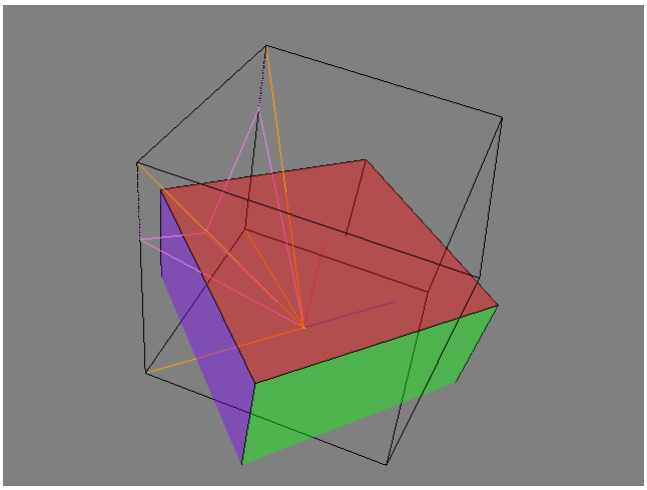

Abbildung 5.7: Darstellung des Clippings für ein Patch gegen den Hemicube. Mit orangenen Linien markiert ist das Patch das zugeschnitten werden soll. Die lilanen Linien stellen das zugeschnittene Polygon dar. Durch Ändern des Blickwinkels kann so leicht überprüft werden, ob die Linien des geschnittenen Polygons durch die Kanten des Hemicubes verlaufen.

zur Wechselseitigkeit (Formel 2.4), Energieerhaltung (Formel 2.5) und Selbstsichtbarkeit (Formel 2.6) mit automatisierten Test versehen.

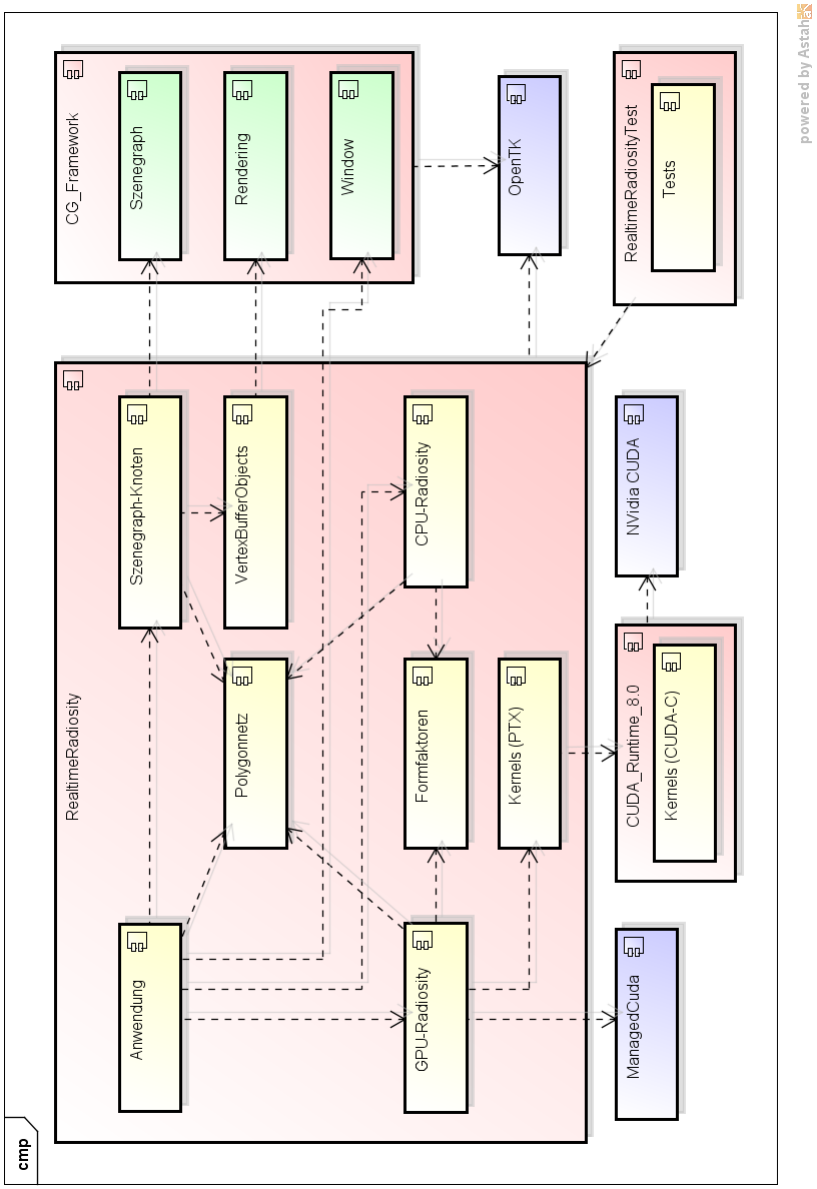

Abbildung 5.8: Komponentendiagramm des Projekts. Es stellt die vier Module (rot), ihre Komponenten (gelb für eigenen Code und grün für den Code vom Betreuer), Abhängigkeiten und Fremdbibliotheken (blau) dar.

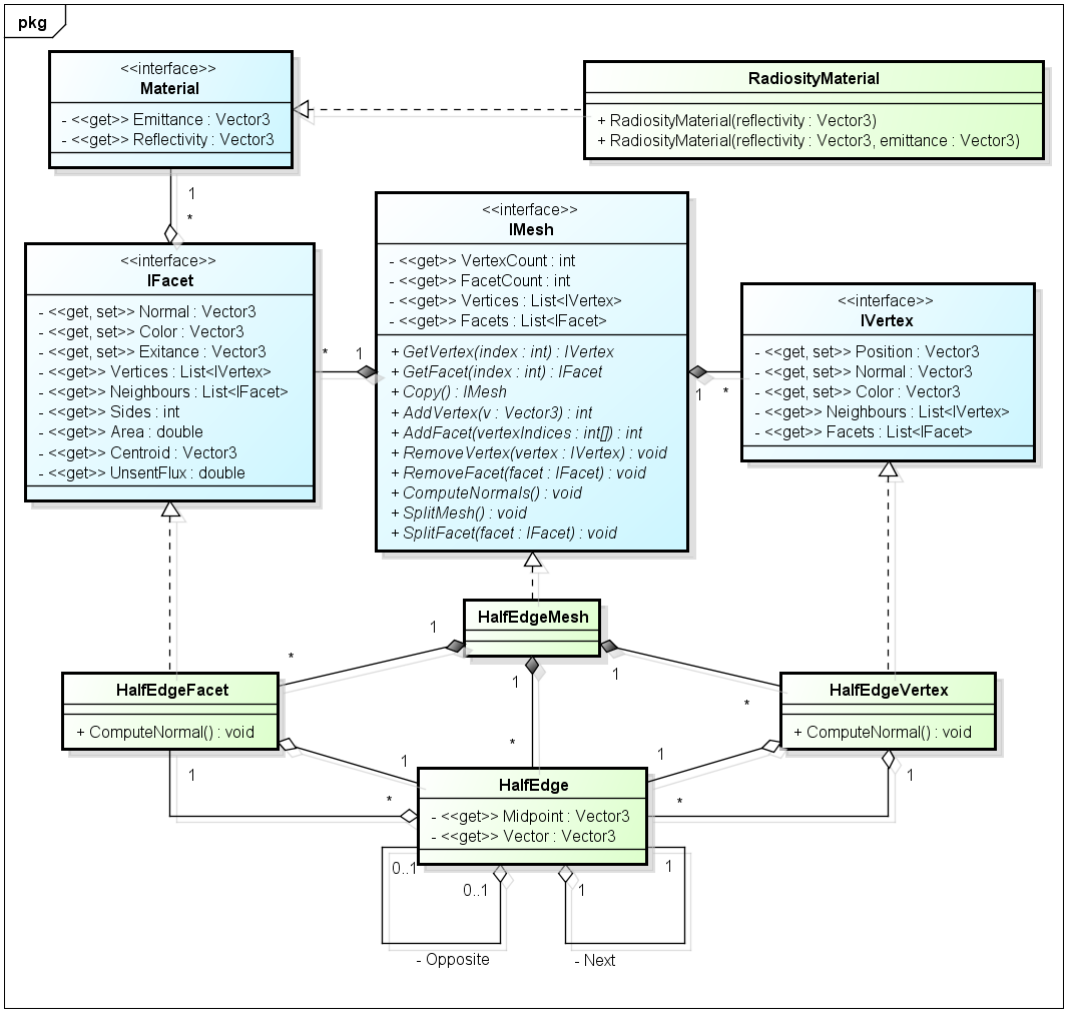

Abbildung 5.9: Klassendiagramm der Polygonnetz-Datenstruktur

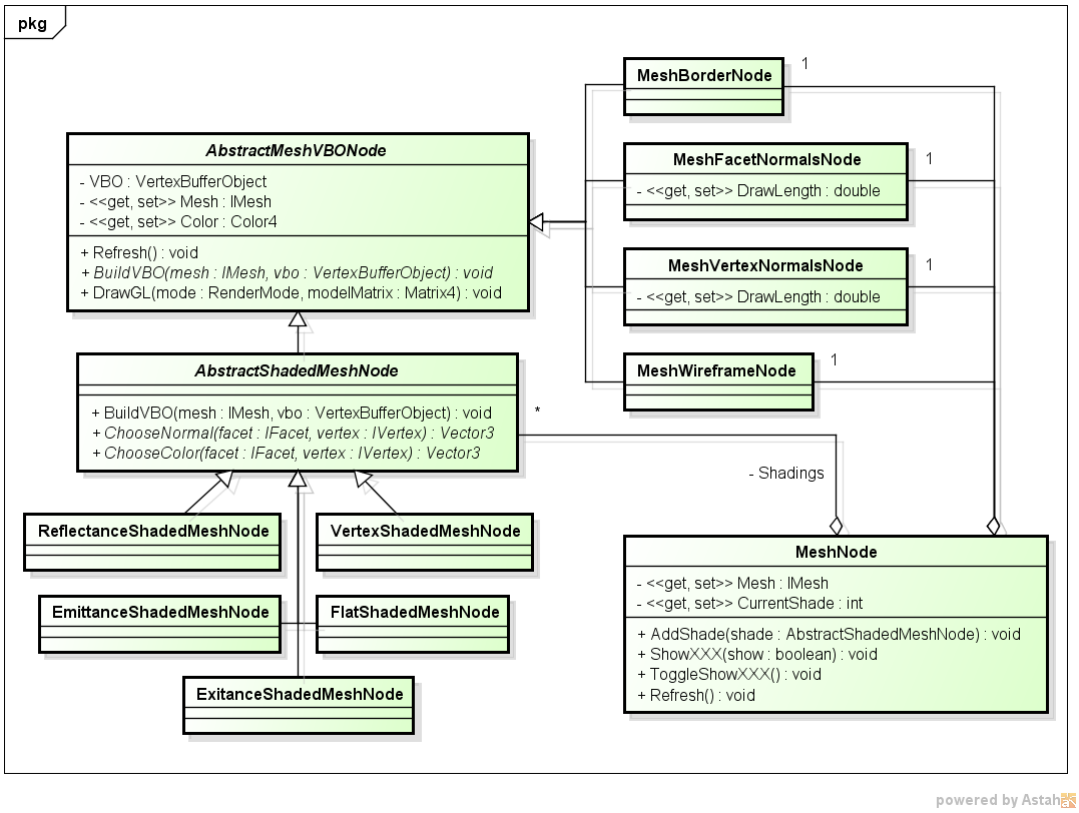

Abbildung 5.10: Klassendiagramm der Szenengraphknoten zur Darstellung von Meshes

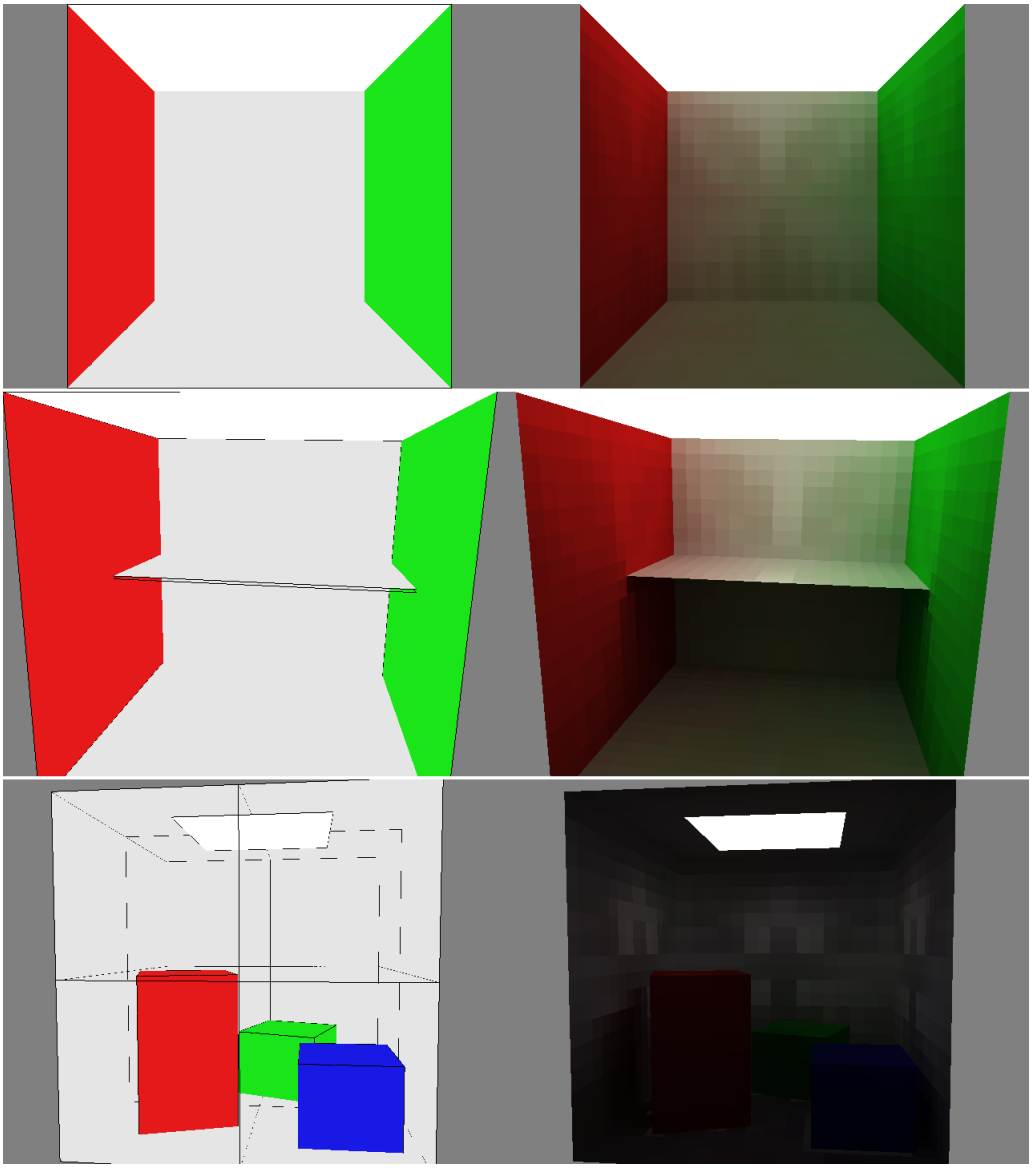

Abbildung 5.11: Zeilen: Einfacher Raum, Schatten Raum, Box Raum. Spalten: Unbeleuchtet (in den Reflektivitäten der Oberflächen), Beleuchtet (Mit der GPU-Radiosity-Implementation berechnet). Bild 1,2: Es ist subtiles Farbbluten zu erkennen. Bild 2,2): Hier ist ein Schattenwurf unter dem mittleren Boden zu erkennen. Bild 3,2: Durch die kleinere Lichtquelle fällt die Umgebung deutlich dunkler aus als ihre Vorgänger.

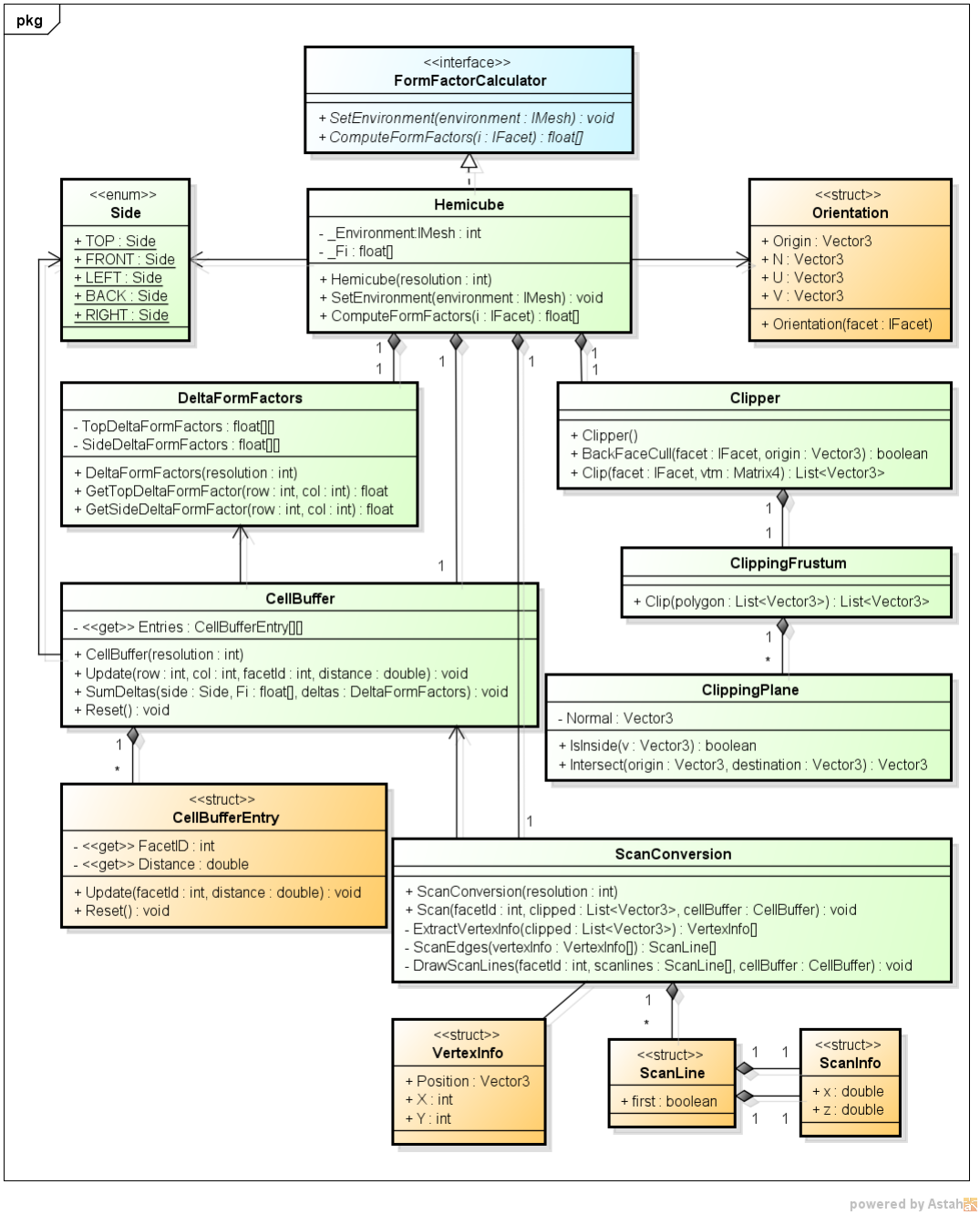

Abbildung 5.12: Klassendiagramm der an der Hemicube-Implementation beteiligten Klassen

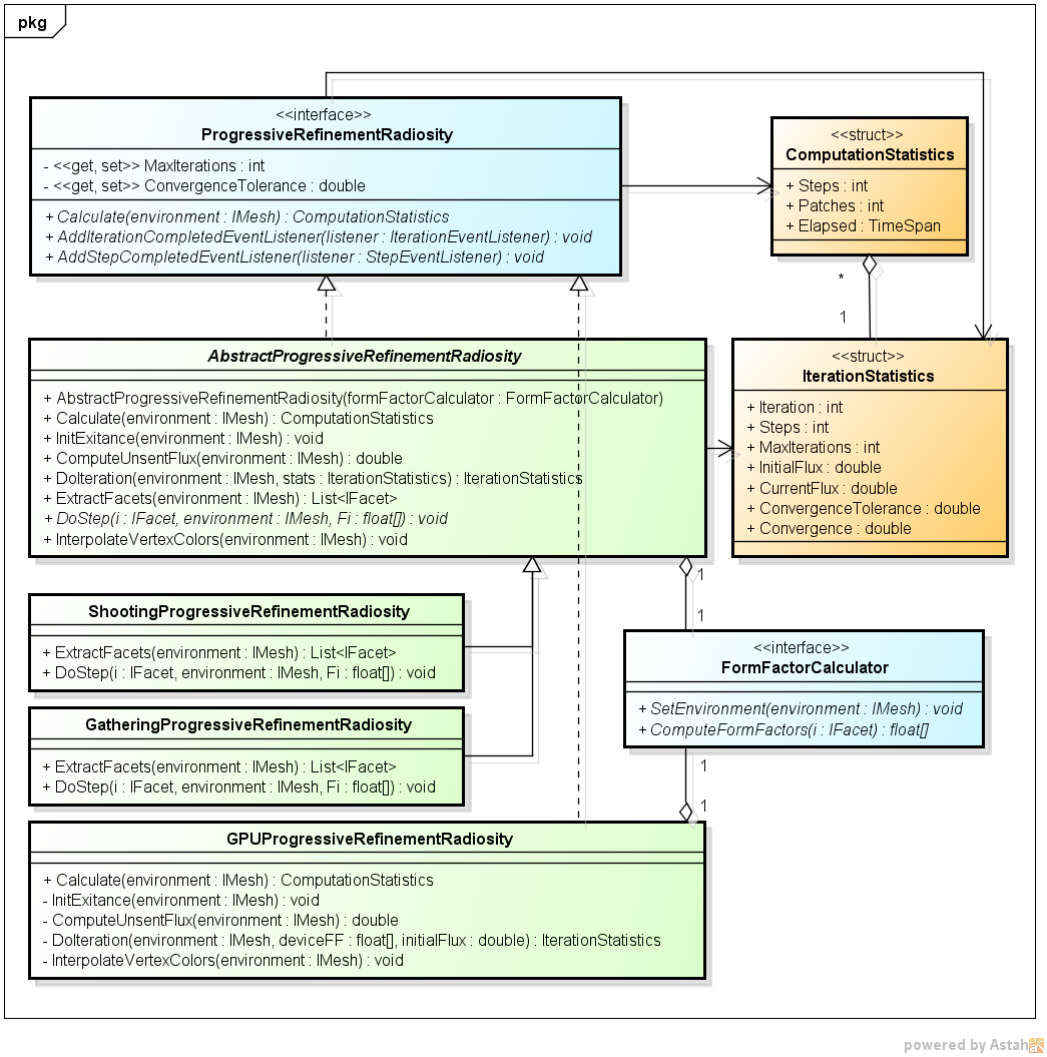

Abbildung 5.13: Klassendiagramm der an der Radiosity-Implementation beteiligten Klassen

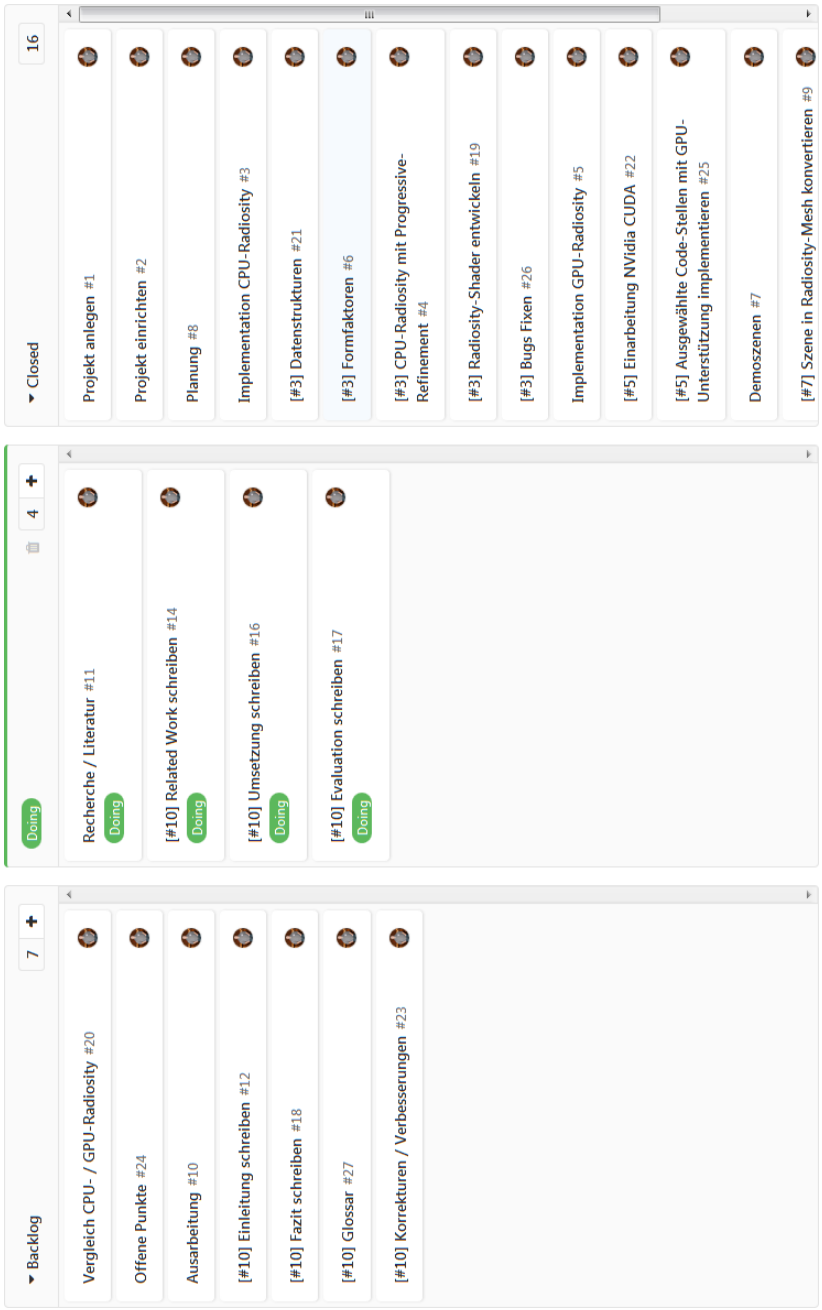

Abbildung 5.14: Das Kanban-Board zum einem Zeitpunkt an dem die Ausarbeitung geschrieben wurde. Aufgabenpakete sind daran zu erkennen, dass sie keine Nummer in eckigen Klammern im Titel tragen. Aufgaben tragen die Nummer ihres Aufgabenpaketes in eckigen Klammern in ihrem Titel.

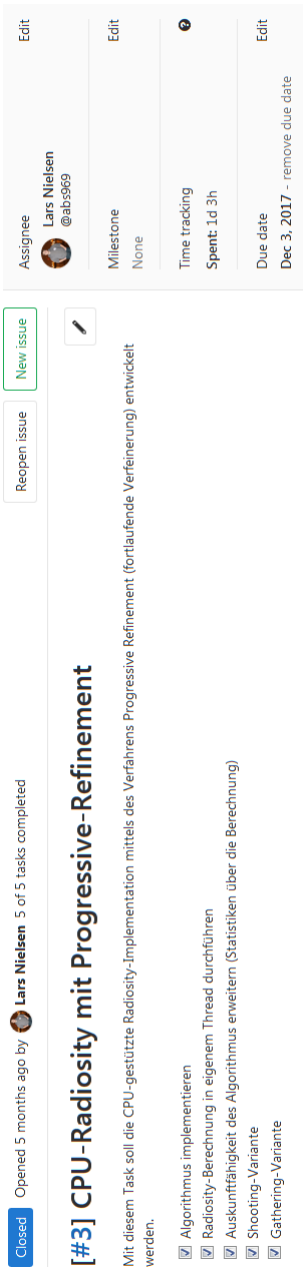

Abbildung 5.15: Eine exemplarische Aufgabe aus dem Kanban-Board. Es sind die abgehakten Unteraufgaben zu erkennen. Auf der rechten Seite ist die gebuchte Zeit und das geplante Ende-Datum der Aufgabe zu sehen.

6 Evaluation

Dieses Kapitel widmet sich der kritischen Bewertung der entstandenen Software. Dazu gehört der Vergleich der Software- und Hardware-Varianten wobei auch das Caching der Formfaktoren einbezogen wird. Anschließend wird darüber resümiert ob die Echtzeit-Fähigkeit die mit dem Titel dieser Arbeit angekündigt wird erreicht werden konnte. Abgeschlossen wird das Kapitel mit einer Einschätzung zum Grad der Hardwareunterstützung der in diesem Projekt erreicht wurde.

6.1 Vergleich

In diesem Abschnitt werden die drei entstandenen Implementationen miteinander hinsichtlich Aussehen und Laufzeit verglichen. Dafür wurden die modellierten Umgebung mit unterschiedlichen Patch-Auflösungen durch die drei Implementationen bis zu einem Konvergenzgrad von 0,01 berechnet. Dies wurde für jede Patch-Auflösung zweimal durchgeführt. Beim ersten Mal mit der einmaligen Berechnung der Formfaktoren, die nachfolgend aus dem Cache wiederverwendet werden. Beim zweiten Mal werden die Formfaktoren direkt aus dem Cache verwendet und gar nicht berechnet. Zunächst wird das Aussehen der Ergebnisse untersucht.

6.1.1 Aussehen

Dieser Abschnitt behandelt das visuell unterschiedliche Verhalten der entstandenen Implementationen. Es werden sowohl die Unterschiede während der Lösungsfindung verglichen, als auch die konvergierten Lösungen. Eine bildliche Darstellung kann Abbildung 6.1 auf Seite 66 entnommen werden.

Die Shooting-Implementation berechnet in einem Schritt den Lichtausstoß eines Patches der allen anderen Patches hinzugefügt wird. Dadurch wird in jedem Schritt der Lichtausstoß jedes anderen Patches um den Anteil des eintreffenden Lichts erhöht. Die Umgebung wird so entsprechend der physikalischen Lichtausbreitung beleuchtet. In Bild

(1,1) aus Abbildung 6.1 ist die obere hintere Ecke noch dunkel, da nicht alle 256 Patches der Lichtquelle verarbeitet wurden. In Bild (1,2) ist die Umgebung bereits ausgeleuchtet und ein Farbbluten ist ebenfalls zu erkennen.

Die Gathering-Implementation berechnet in einem Schritt den Lichtausstoß eines Patches aus dem eintreffenden Licht aller anderen Patches. Dadurch wird pro Schritt immer nur ein Patch erhellt. Das gestaltet sich nicht so elegant wie der Shooting-Ansatz, wie Zeile 2 aus 6.1 entnommen werden kann. In Bild (2,1) fallen besonders die schwarzen, noch nicht verarbeiteten Patches auf. Die Implementation berechnet dunklere Patches in einer Iteration zuerst. In Bild (2,2) sind die in der Iteration bereits verarbeiteten Patches deshalb sehr deutlich zu erkennen.

Die GPU-Implementation berechnet die Schritte einer Iteration parallel durch die Grafikhardware. Dadurch wird die komplette Umgebung mit Abschluss einer Iteration erhellt. Durch diesen Ansatz fällt die Formfaktorberechnung in der ersten Iteration besonders negativ auf. Bis alle Formfaktoren berechnet wurden, kann die Radiosity-Berechnung nicht durchgeführt werden. Werden die Formfaktoren aus dem Cache verwendet, erhellt sich die Umgebung jedoch sehr schnell und gleichmäßig.

Die konvergierten Lösungen in der letzten Spalte von Abbildung 6.1 unterscheiden sich ebenfalls. Die Gathering- und Hardware-Variante unterscheiden sich nur in der Helligkeit. Die Gathering-Variante ist etwas heller, da die Patches in der Reihenfolge des geringsten Lichtbeitrags verarbeitet werden. Dadurch erhält ein später verarbeitetes Patch in einer Iteration mehr Licht als ein Vorgänger, da diese bereits mehr Licht aufgenommen haben. Die Shooting-Variante unterscheidet sich dagegen deutlich von den anderen. Möglicherweise liegt das an Gleitkommaungenauigkeiten in der Bestimmung des wechselseitigen Formfaktors. Baum et al. haben 1989 ebenfalls entdeckt, dass das Progressive Refinement Verfahren andere Ergebnisse liefert [2]. Der genauen Ursache für die Unterschiede wurde im Rahmen dieses Projekts jedoch nicht nachgegangen.

6.1.2 Laufzeitanalyse

Nachdem das Aussehen der von den Implementationen erzeugten Lösungen verglichen wurde, wird im Folgenden die Laufzeit bis zum Erreichen der Toleranzgrenze von 0,01 verglichen. Die in diesem Abschnitt besprochenen Daten sind den Abbildungen 6.2, 6.3 und 6.4 auf den Seiten 67, 68 und 69 zu entnehmen. Sie zeigen die Laufzeit der

Verfahren einmal inklusive der einmaligen Formfaktorberechnung und einmal exklusive der Formfaktorberechnung in der jeweiligen Umgebung.

Während die Shooting- und Gathering-Implementationen sowohl mit Formfaktorberechnung als auch ohne jeweils sehr nahe beieinander liegen, setzt sich die Hardware-Implementation mit wachsenden Patchzahlen deutlich von den anderen Implementationen ab. So liegt die Laufzeit der Hardware-Implementation ohne Formfaktorberechnung selbst bei 10240 Patches noch im Sekundenbereich. Die Software-Implementationen liegen dagegen in der Nähe von 15 Minuten. Bei kleineren Patchzahlen benötigt die Hardware-Implementation dagegen mehr Zeit. Das lässt sich mit dem zusätzlichen Overhead erklären der durch das Kopieren von Daten zwischen dem Host- und Device-Speicher entsteht.

Auffällig ist auch, dass die Laufzeit der Gathering-Implementation für den einfachen Raum und den Schatten Raum unter der Laufzeit der Shooting-Implementation liegt, dies im Box Raum jedoch nicht mehr gilt. Da die Shooting-Implementation in jeder Iteration nur Patches verarbeitet die genug Licht in die Umgebung einbringen, werden Patches die keinen Beitrag dazu leisten erst verarbeitet wenn sie genug Licht erhalten haben. In der Gathering-Implementation dagegen kann nicht trivial entschieden werden, ob es sich lohnt ein Patch zu verarbeiten. Deshalb verarbeitet sie auch Patches die in einer Iteration gar kein Licht erhalten können.

6.2 Realtime-Fähigkeit

In der Informatik ist der Begriff der Echtzeitsysteme[1] bereits vorbelegt. So gilt an diese Systeme oft die Anforderung vor Ablauf eines Zeitintervalls, der gerne im Millisekundenbereich liegt, ein Ergebnis liefern zu müssen. Bei harten Echtzeitanforderung gilt das Nicht-Abliefern eines Ergebnisses in diesem Intervall als Versagen. Im Bereich der Computergrafik ist mit Realtime-Fähigkeit dagegen oft der Begriff Interaktivität gemeint [9]. Dies lässt sich eher mit weichen Echtzeitanforderungen gleichsetzen. Während für eine flüssige Darstellung eine Bildwiederholungsrate von mindestens 24 Bildern pro Sekunde wünschenswert ist [2], sind geringere Bildwiederholungsraten je nach Anwendungsfall vertretbar.

[1] `https://de.wikipedia.org/wiki/Echtzeitsystem` - Abgerufen: 27. März 2018
[2] `https://de.wikipedia.org/wiki/Bildfrequenz` - Abgerufen: 27. März 2018

Die in diesem Projekt entstandene Anwendung stellt weiche Echtzeitanforderungen. Wichtig ist hier die Interaktivität mit der Anwendung, sodass der Blickwinkel während der Radiosity-Berechnung gewechselt, die Berechnung abgebrochen oder zwischen den Repräsentationen des Polygonnetzes gewechselt werden kann. Dies wird gewährleistet, indem die Anzeige und die Berechnung in zwei parallelen Threads ausgeführt wird. In der Radiosity-Berechnung werden Ereignisse für den Abschluss eines Schrittes oder einer Iteration ausgelöst, wodurch die Anzeige mit den zuletzt berechneten Werten aktualisiert wird.

Die Laufzeitanalyse zeigt, dass die Hardware-Implementation eine Szene mit 10240 Patches in 4,8 Sekunden (ohne Formfaktoren) berechnet. Dies geschieht in der Szene über 12 Iterationen, was 0,4 Sekunden pro Bild entspricht. Bei einer Auflösung von 2560 wird dieselbe Szene über 12 Iterationen in 1,2 Sekunden oder 0,1 Sekunden pro Bild berechnet. Dies entspricht bei 10240 Patches 2,5 Bildern pro Sekunde und bei 2560 Patches in etwa 10 Bildern pro Sekunde. Um die bereits angesprochenen 24 Bilder pro Sekunde zu erreichen müsste die Berechnung in 0,042 Sekunden abgeschlossen sein. Geringere Zeiten werden erst bei Patchzahlen von 640 und niedriger erreicht.

Ein anderes Bild zeichnet sich, wenn die Shooting-Implementation betrachtet wird. Durch ihre Natur erzeugt sie nach jedem Schritt ein brauchbares Bild. Während sie für 10240 Patches (ohne Formfaktoren) 13,5 Minuten benötigt, entspricht das bei den 48180 benötigten Schritten etwa 59,5 Bildern pro Sekunde. Damit gestaltet sich die Variante deutlich flüssiger als die GPU-Implementation.

Durch den Aufbau mit den zwei Threads wirken sich die geringen Bildwiederholungsraten der GPU-Implementation nicht negativ auf die Interaktivität mit der Anwendung aus. Die absolute Berechnungszeit konnte durch die Hardwareunterstützung deutlich reduziert werden, was durchaus als Erfolg zu werten ist. Die geringen Bildwiederholungsraten der GPU-Implementation sind dagegen zwar ernüchternd, erwachsen aber aus dem Gathering-Ansatz der mit der Implementation verfolgt wurde. Es könnte sich also lohnen den Shooting-Ansatz mit Hardwareunterstützung umzusetzen. Da in diesem Projekt auch nicht alle Möglichkeiten der GPGPU-Programmierung ausgeschöpft wurden, beschäftigt sich der nächste Abschnitt damit wie die Hardwareunterstützung verbessert werden könnte.

6.3 Hardwareunterstützung

Wie der Laufzeitanalyse und dem vorigen Abschnitt entnommen werden kann, hat sich die Hardwareunterstützung bereits gelohnt. Dabei ist sie unter minimalem Aufwand entstanden. Es musste viel Zeit in das Verständnis des Radiosity-Verfahrens und die Software-Implementation investiert werden, da es eine große Menge von Konzepten beinhaltet. Die entstandene Hardware-Implementation schöpft also nicht alle Möglichkeiten der GPGPU-Programmierung aus und lässt großen Raum für weitere Verbesserungen.

In der Laufzeitanalyse fällt bei der Hardware-Implementation der Sprung von 1,18 Sekunden bei 2560 Patches zu 4,84 Sekunden bei 10240 Patches auf. Da alle Patches aus der Umgebung parallel berechnet werden, die berechneten Lichtausstöße allerdings in einem Thread durch die CPU wieder in die Datenstrukturen übernommen werden, liegt hier Potential für einen Engpass. Je mehr Patches die Umgebung enthält, desto mehr Werte müssen sequentiell wieder übernommen werden. Bei großen Patchzahlen könnten die Patches deshalb in Gruppen aufgeteilt und in mehreren asynchronen Hardware-Aufrufen verarbeitet werden. So könnte die CPU die Radiosity-Werte einer bereits verarbeiteten Gruppe übernehmen, nachdem der Grafikhardware die Verarbeitung der nächsten Gruppe beauftragt wurde.

Heckbert schlägt den Einsatz von Texturen zur Modellierung der Schattierungsinformationen vor [17, 14]. Dadurch muss die Unterteilung nicht mehr auf der Geometrie der Umgebung sondern kann hardware-unterstützt auf der Textur durchgeführt werden [17]. Coombe und Harris weisen zum Beispiel jedem Polygon zwei Texturen zu, deren Texel den Elementen in der Radiosity-Berechnung entsprechen [15]. Eine Textur enthält dabei den akkumulierten und eine Textur den verbleibenden Lichtausstoß [15]. Durch den Einsatz von Texturen kann der Texturspeicher der Grafikhardware genutzt werden.

Coombe und Harris setzen auch das Progressive Refinement Verfahren auf der Grafikhardware um. Dabei gehen sie grundsätzlich nach dem Shooting-Ansatz vor um den nächsten Sender auszuwählen, ermitteln dann aber die von dem Sender getroffenen Empfänger und berechnen die Lichtausstöße in den Texturen der Empfänger nach dem Gathering-Ansatz [14]. Auch die Auswahl des nächsten Senders wird mit Hardwareunterstützung umgesetzt. Um den gesamten Lichtausstoß einer Oberfläche zu ermitteln verwenden Coombe und Harris eine Mip-Map deren oberstes Element den

durchschnittlichen Lichtausstoß eines Texels der Textur der Oberfläche repräsentiert und multiplizieren diesen mit der Auflösung der Textur [14]. Der nächste Sender kann dann bestimmt werden, indem der umgekehrte Lichtausstoß der Oberflächen als Tiefeninformation und die ID der Oberfläche als Farbe in einem 1x1 Pixel Z-Puffer verarbeitet werden [14]. Die am Ende im Z-Puffer verbleibende Farbe entspricht der ID der nächsten Sender-Oberfläche.

Eine weitere Möglichkeit bietet der Einsatz von Grafik-Interoperabilität[3] in dem CUDA-Code. In der Implementation für dieses Projekt werden die für die Radiosity-Berechnung benötigten Werte aus den Datenstrukturen in den Device-Speicher kopiert, mit der Hardware verarbeitet, die Ergebnisse zurück in die Datenstrukturen auf dem Host kopiert aus denen wiederum VertexBuffer-Objekte erzeugt werden, die der Grafikhardware zur Anzeige übergeben werden. Mit Grafik-Interoperabilität könnte der CUDA-Code die VertexBuffer-Objekte direkt aktualisieren, wodurch potentiell eine Menge Overhead reduziert werden kann.

Mit den in diesem Abschnitt beschriebenen Ansätzen werden bereits erste potentielle Verbesserungsvorschläge für die Hardware-Implementation gegeben. Sie bieten wertvolle Anhaltspunkte denen weiterhin nachgegangen werden könnte. Das folgende Kapitel fasst diese Arbeit noch einmal zusammen und greift Verbesserungspotential auf.

[3] `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#graphics-interoperability` - Abgerufen: 27. März 2018

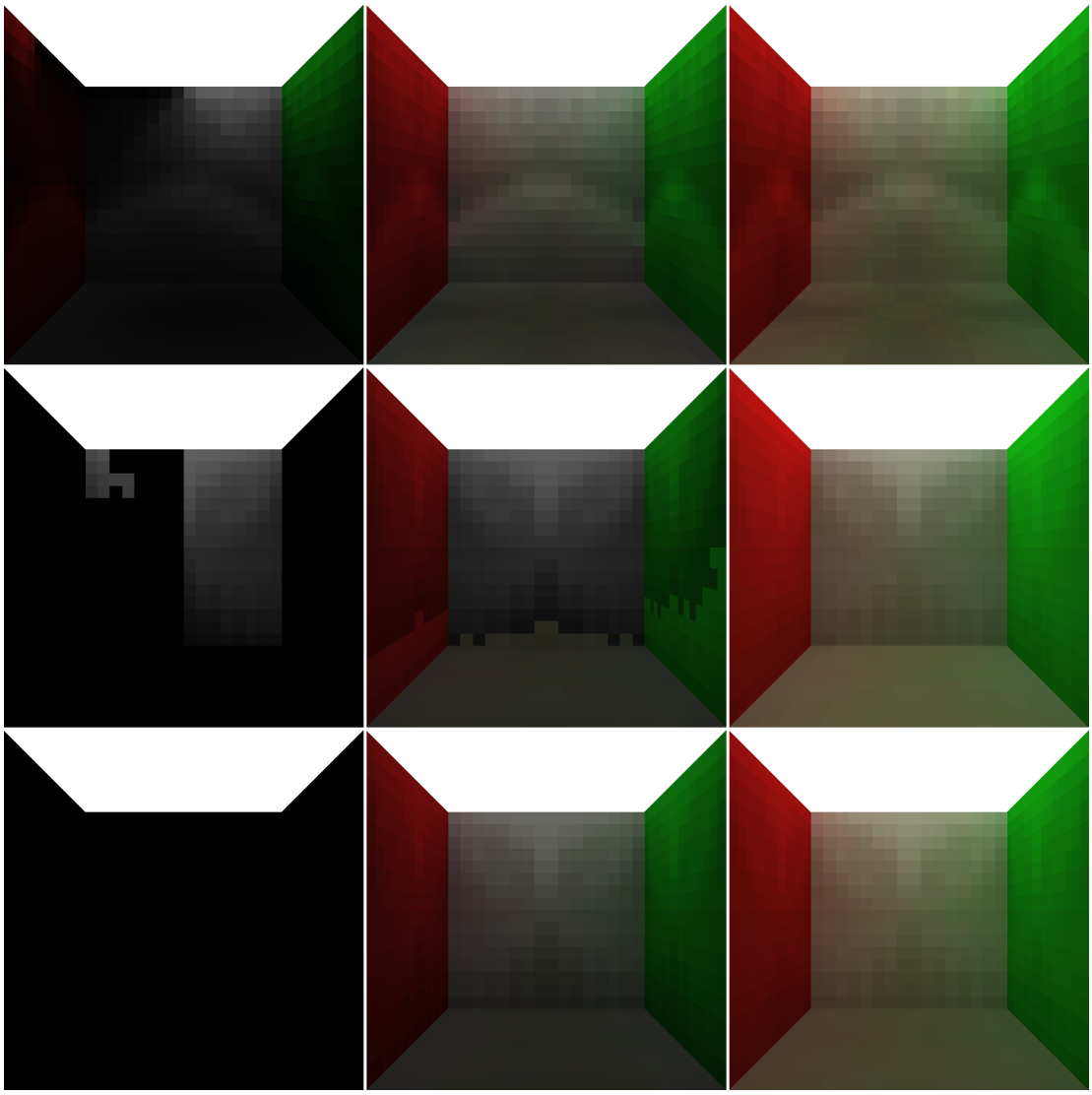

Abbildung 6.1: Zwischenergebnisse der verschiedenen Implementationen für den Simple Room mit 1536 Patches. Zeilen: Shooting, Gathering, GPU. Spalten: 138 Schritte, 1674 Schritte, konvergierte Lösung. Spalten für GPU (letzte Zeile): Vor der ersten Iteration, nach der ersten Iteration, konvergierte Lösung.

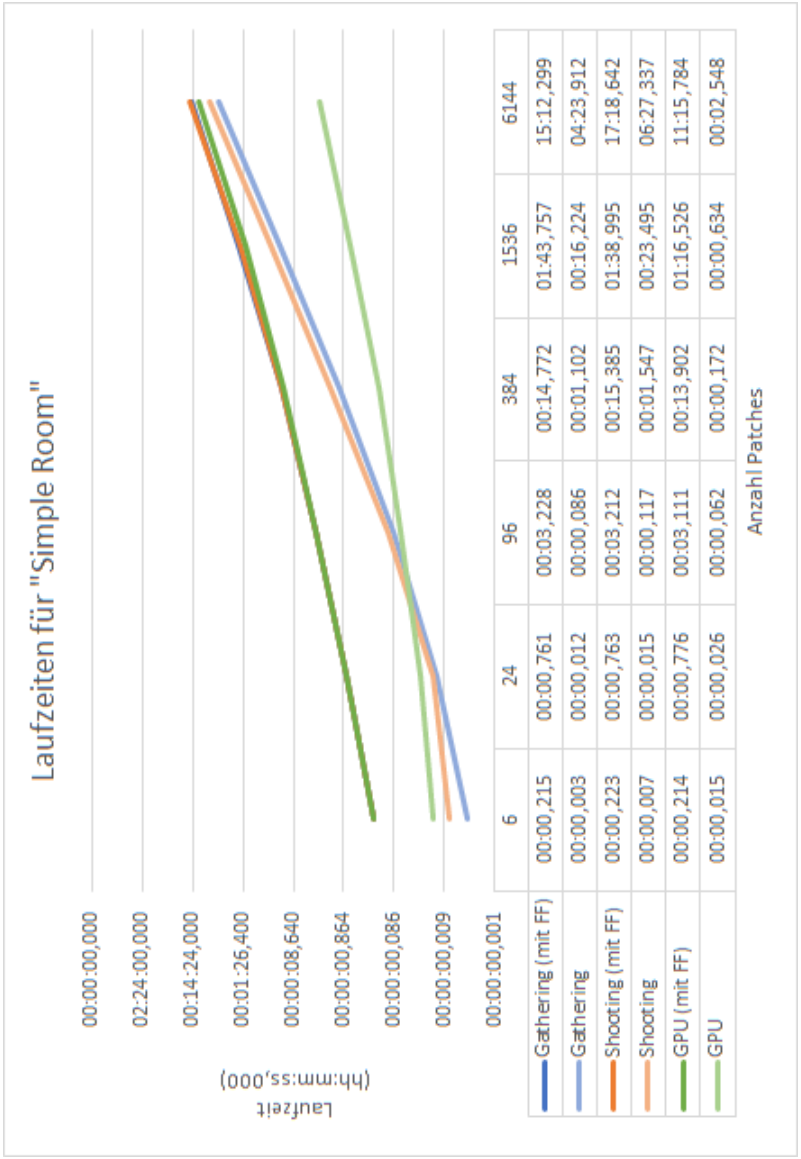

Abbildung 6.2: Laufzeiten der Radiosity-Varianten für die einfacher Raum Szene. Das Liniendiagramm verwendet auf der Laufzeit-Achse eine logarithmische Skalierung. Verglichen werden die unterschiedlichen Verfahren einmal inklusive der einmaligen Formfaktorberechnung (mit FF) und einmal exklusive.

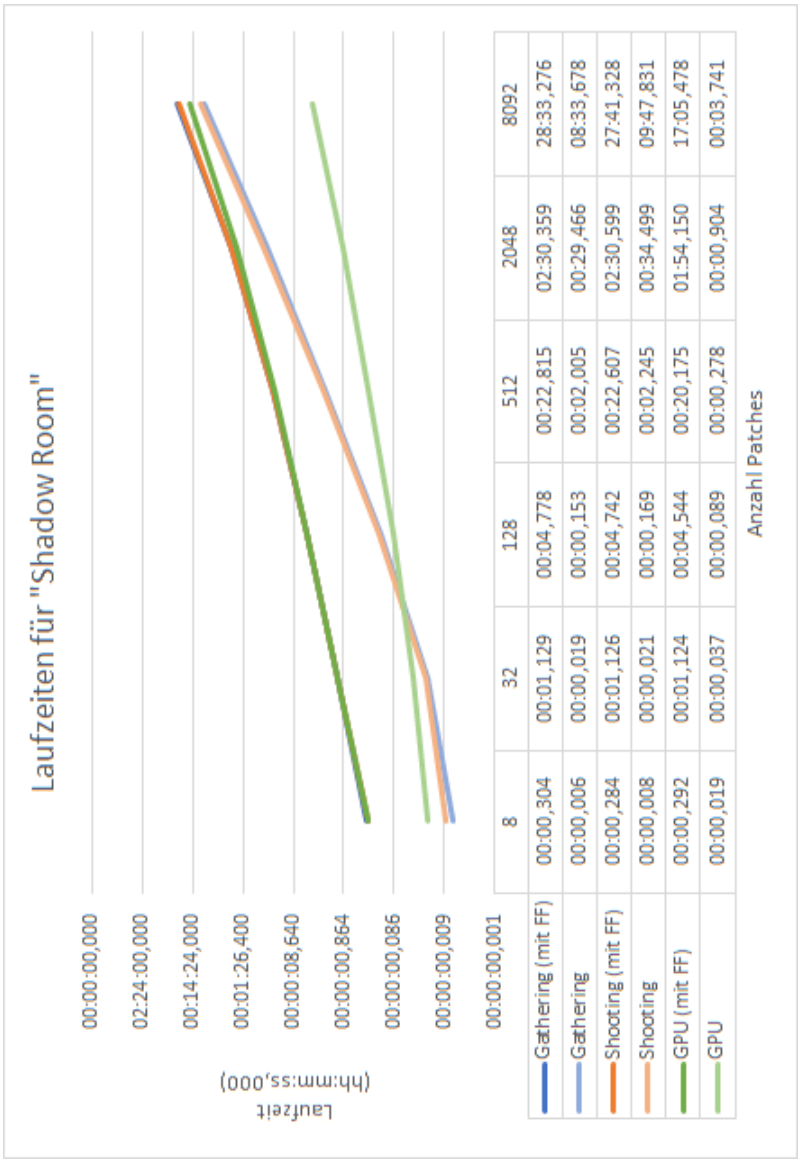

Abbildung 6.3: Laufzeiten der Radiosity-Varianten für die Schatten Raum Szene. Das Liniendiagramm verwendet auf der Laufzeit-Achse eine logarithmische Skalierung. Verglichen werden die unterschiedlichen Verfahren einmal inklusive der einmaligen Formfaktorberechnung (mit FF) und einmal exklusive.

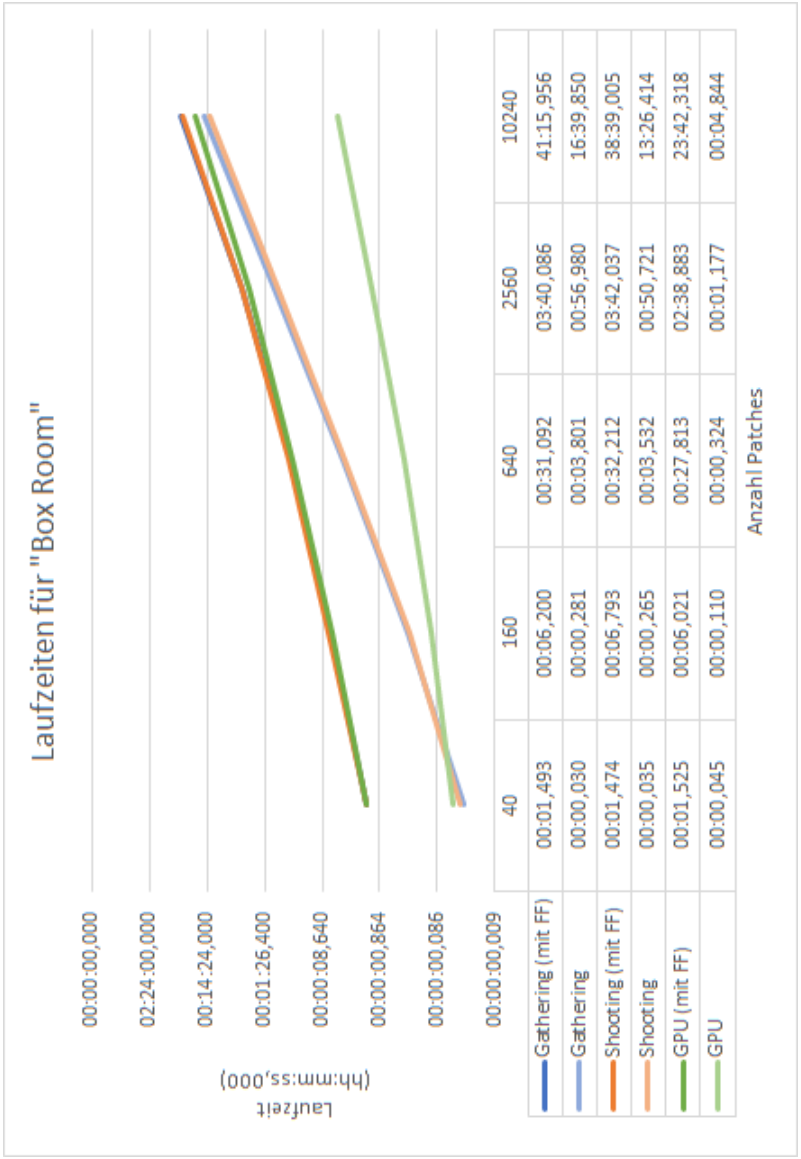

Abbildung 6.4: Laufzeiten der Radiosity-Varianten für die Box Raum Szene. Das Liniendiagramm verwendet auf der Laufzeit-Achse eine logarithmische Skalierung. Verglichen werden die unterschiedlichen Verfahren einmal inklusive der einmaligen Formfaktorberechnung (mit FF) und einmal exklusive.

7 Zusammenfassung

Mit dieser Arbeit wurde ein hardwareunterstützter Prototyp des Radiosity-Verfahren entwickelt. Dafür ist eine Software-Implementation des Progressive-Refinement-Radiosity-Verfahrens entstanden, die sich an der Implementation von Ashdown orientiert. Als Zwischenschritt zur Hardware-Implementation entstand eine weitere Software-Implementation welche das Progressive-Refinement-Radiosity-Verfahren von dem Shooting-Ansatz zurück in den Gathering-Ansatz des ursprünglichen Radiosity-Verfahrens umformuliert. Auf Basis dieses Zwischenschritts entstand eine hardwareunterstützte Implementation des Radiosity-Verfahrens die nach dem Gathering-Ansatz arbeitet.

Die entstandenen Implementationen wurden hinsichtlich des Aussehens ihrer Ergebnisse und der Laufzeit verglichen. Dabei ist ein deutlicher Performancegewinn durch die Hardwareunterstützung aufgefallen. Die Ziele der Realtime-Fähigkeit konnten dagegen nur teilweise erreicht werden. Es ist abzusehen, dass umfangreiche Umgebungen zu langen Rechenzeiten führen, die mit der entstandenen Implementation außerhalb von interaktiven Rechenzeiten liegen. Um auch umfangreichere Umgebungen interaktiv darzustellen muss die Umsetzung weiter verbessert werden. Der nächste Abschnitt beschäftigt sich mit Ideen, durch die die Implementation weiter beschleunigt werden kann.

7.1 Ausblick

Im Folgenden werden Erweiterungsmöglichkeiten vorgestellt um die Performance der entstandenen Lösungen zu verbessern. Die Erweiterungen ergeben sich aus Erkenntnissen die während der Arbeit gewonnen wurden.

7.1.1 Formfaktorberechnung

Vor Beginn des Projekts wurde der Aufwand für die Formfaktorberechnung unterschätzt. Während der Durchführung des Projekts hat sich herausgestellt, dass die Formfaktorbe-

rechnung ein größeres Problem ist, als die Radiosity-Berechnung selbst. Deshalb wurde viel Zeit in das Studium und die spätere Implementation der Formfaktorberechnung investiert. Obwohl durch Ashdown bereits eine funktionierende C-Implementation für das Hemicube-Verfahren bereitstand musste das Verständnis für die beteiligten Komponenten wie das Clipping, die Scan-Conversion, Delta-Formfaktorberechnung oder den Zellpuffer zunächst aufgebaut werden. Eine weitere Schwierigkeit bestand darin, dass das Radiosity-Verfahren auf die Formfaktoren angewiesen ist bevor die Berechnung sinnvolle Ergebnisse liefern kann.

Aus der Laufzeitanalyse (Abschnitt 6.1.2) kann entnommen werden, dass die Berechnung der Formfaktoren sehr aufwändig ist. Die Formfaktorberechnung als nächsten Schritt mit Hardwareunterstützung umzusetzen liegt deshalb nahe. Die Projektion der Umgebung auf die fünf Hemicube-Oberflächen aus der Sicht eines Patches entspricht bereits einem Rendering-Vorgang. Dieser kann auch mit gängiger Grafikhardware ausgeführt werden[9]. Baum et al. merken zudem an, dass die Formfaktorberechnung mittels Hemicube zu Aliasing-Problemen führt [2]. Die Bestimmung der Formfaktoren sollte mit einem anderen Verfahren geschehen, während der Hemicube für die Sichtbarkeitsprüfung weiterhin eingesetzt werden kann [14].

7.1.2 Auswahlstrategien für nächstes Patch

Die ExtractFacets-Methode (siehe Abschnitt 5.4.2) bietet den Software-Implementationen in jeder Iteration die Möglichkeit Patches für die Verarbeitung zu sortieren. Die Gathering-Implementation nutzt diese Möglichkeit, um Patches entsprechend ihres Lichtbeitrags aufsteigend zu verarbeiten. Da in der Gathering-Implementation der verbleibende Lichtausstoß des aktuellen Patches mit dem gesammelten Licht überschrieben wird, wird so sichergestellt, dass Licht nicht verloren geht. Anders würde eine Lichtquelle die verarbeitet wird bevor alle anderen Patches ihren Beitrag aufgenommen haben ihren vorherigen Lichtausstoß verlieren. Die Shooting-Implementation dagegen profitiert davon Patches die über einen großen verbleibenden Lichtausstoß verfügen bevorzugt zu verarbeiten. Hier können Patches die vor einer Iteration über keinen eigenen Lichtausstoß verfügen von der Iteration ausgenommen werden. Während sie in der Iteration zwar Licht erhalten können, so können sie auch in der nächsten Iteration verarbeitet werden. Sollte ein Patch dagegen während der Iteration kein Licht erhalten und trotzdem verarbeitet werden, wäre das verschwendete Rechenzeit. Insbesondere bei der Shooting-Variante ist der Ansatz interessant, da eine Iteration i in der gegenwärtigen Umsetzung die $i-1$te

Reflexion in der Umgebung berechnet, wobei die nullte Reflexion der direkte Beleuchtung entspricht.

7.1.3 Hardwareunterstützung

Die Verbesserung der Hardwareunterstützung bietet noch großes Potential. In Abschnitt 6.3 wurden bereits Ansätze für eine bessere Hardwareunterstützung vorgestellt. Dort werden bereits eine Reihe von Vorschlägen formuliert, denen es sich lohnen würde nachzugehen. Eine weitere Möglichkeit ist die bereits erwähnte Berechnung der Formfaktoren mit Hardwareunterstützung.

7.1.4 Modellierung der Umgebung

Die in diesem Projekt entstandenen Implementationen ermöglichen die Übergabe von nur einem Polygonnetz. Eine komplette Umgebung wird also nur durch ein Polygonnetz repräsentiert. Einzelne Objekte oder Oberflächen sind nicht in einem eigenen Polygonnetz. Während das Netz durchaus so aufgebaut werden kann, dass unterschiedliche Oberflächen wie Wände ihre Eckpunkte nicht teilen, versucht der implementierte Subdivision-Algorithmus für potentiell neue Eckpunkte aktiv bestehende Eckpunkte mit denselben Koordinaten wieder zu verwenden. Das heißt Oberflächen die nicht zusammen gehören verschmelzen miteinander. Als Ergebnis daraus erhalten Eckpunkte andere Normalen. Die Normalen von zwei rechtwinklig aufeinander stehenden Wänden sollten an ihren Rändern ebenfalls rechtwinklig von der Oberfläche weg zeigen. Durch die Verschmelzung zeigen diese Normalen nun von beiden Oberflächen jeweils in einem 45 Grad Winkel weg. Auf die Interpolation hat diese Verschmelzung die Auswirkung, dass ein Eckpunkt am Rand zwischen zwei Wänden die Farbe seiner anliegenden Patches beider Wände annimmt. Dadurch entsteht ein ungewollt weicher Übergang zwischen den Wänden. Abbildung 7.1.4 stellt diese Artefakte grafisch dar.

Es sollte also eine geeignetere Datenstruktur zum Modellieren einer kompletten Umgebung geschaffen werden. Die Interpolation könnte von der Trennung nicht zusammen gehörender Objekte in eigene Polygonnetze profitieren. Außerdem ermöglicht die Datenstruktur sie nur die gleichzeitige Unterteilung aller Patches des Netzes. Um zum Beispiel die adaptive Unterteilung (siehe 3.2.2) zu unterstützen wäre das lokale Aufteilen von einzelnen Patches wünschenswert. Worauf dabei zu achten ist haben Baum et al. untersucht [3]. Auch die weitere Unterteilung von Patches in Elemente klingt vielverspre-

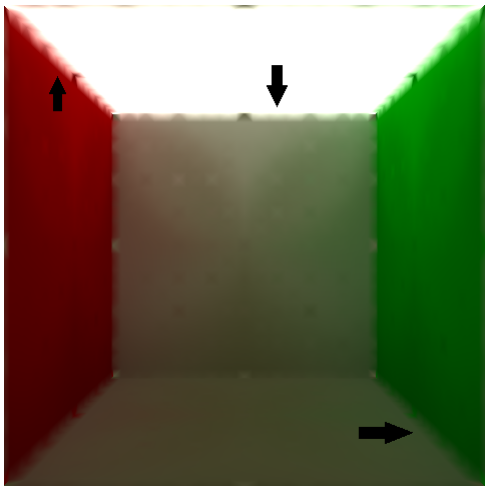

Abbildung 7.1: Ergebnis der Interpolation für den „Einfachen Raum“ mit einer Auflösung von 6144 Patches (6 * 1024). Die Pfeile zeigen ausgewählte Kanten an denen keine Interpolation geschehen sollte.

chend. Zuletzt ist die Modellierung der Umgebungen nicht besonders ausgearbeitet. Die Szenen wurden mit Programm-Code modelliert. Wünschenswert wäre das Importieren von Szenen aus gängigen 3D-Modellierungstools. Deshalb wäre ein Parser denkbar, der Umgebungen mit mehreren Polygonnetzen aus Dateien laden kann.

7.1.5 Caching der Formfaktoren

Durch das Cachen der Formfaktoren entsteht alleine für die Formfaktoren ein Speicherbedarf von $O(n^2)$. Während sich das wie bereits in der Laufzeitanalyse (siehe Abschnitt 6.1.2) erwähnt positiv auf die Laufzeit auswirkt kann der Speicherbedarf Folgen haben. Laufzeit hat eine weiche Grenze insofern, dass Zeit nicht limitiert ist. Speicherbedarf dagegen ist durch den verfügbaren Arbeitsspeicher und eventuell Festplattenspeicher für eine Auslagerungsdatei beschränkt. Wenn kein weiterer Speicher zur Verfügung steht führt dies unweigerlich zu einem Absturz der Anwendung. Der implementierte Cache lagert Formfaktoren nicht eigenständig aus. Deshalb kann das Auslagern auf die Festplatte vernachlässigt werden. Bei 32 bit pro Formfaktor benötigen die Formfaktoren einer Umgebung aus 512 Patches bereits einen Megabyte Arbeitsspeicher. Ein Gigabyte wird bei 16384 Patches erreicht.

Der Speicherbedarf könnte auf mehrere Wege reduziert werden. So brauchen Formfakto-

ren zwischen Patches die sich nicht sehen können auch nicht gespeichert werden da sie immer 0 sind. Eine weitere Möglichkeit ist die Unterteilung von Patches in Elemente. In dem Fall werden Patches gröber und die Elemente feiner aufgelöst. Wie bereits in Abschnitt 3.2.1 beschrieben können dann Patch-zu-Element-Formfaktoren berechnet werden, was den Speicherbedarf auf $O(patches * elemente)$ einschränkt.

Wenn die Formfaktorberechnung bedeutend beschleunigt werden kann, kann sogar über den Verzicht des Caches nachgedacht werden. Da in diesem Projekt der Fokus auf der Berechnung der Radiosity-Werte lag, wurden keine Anstrengungen unternommen die Formfaktorberechnung mit Hardwareunterstützung zu implementieren. Cohen et al. haben in ihrem Progressive Refinement Radiosity Verfahren bereits entschieden Formfaktoren „on the fly“ zu berechnen [9]. Im Rahmen dieses Projekts relativiert der Cache jedoch die Dauer der Formfaktorberechnung zufriedenstellend.

Glossar

Aliasing Effekt der entsteht wenn kontinuierliche Informationen diskretisiert werden, beim Rendern eines Bildes kann zum Beispiel Treppenbildung bei eigentlich durchgezogenen Linien entstehen. 23

Betrachtungssystem Ein Betrachtungssystem übernimmt den Vorgang eine Umgebung für einen Betrachter darzustellen, dabei werden verdeckte und nicht sichtbare Oberflächen von der weiteren Verarbeitung ausgenommen und Weltkoordinaten (World-Space) in Kamerakoordinaten (View-Space) und später in Bildkoordinaten (Image-Space) transformiert. 1, 26

Bildsynthese Der Vorgang für die Erzeugung eines Bildes aus (z.B. dreidimensionalen) Rohdaten, auch Rendering genannt. 1, 12, 78

blickwinkelabhängig Bedeutet, dass eine Berechnung abhängig vom Blickwinkel eines Betrachters ist, ändert sich dieser muss die Berechnung erneut durchgeführt werden. 7

blickwinkelunabhängig Bedeutet, dass eine Berechnung unabhängig vom Blickwinkel eines Betrachters ist, ändert sich dieser muss die Berechnung nicht erneut durchgeführt werden. 1, 7, 8, 23

Clipping Zuschneiden eines Polygons. 26, 75

Clipping-Plane Ebene im 3D Raum gegen die ein Polygon beim Clipping abgeschnitten wird. 26, 39

Device Bezeichnet in der Grafik- und GPGPU-Programmierung die Grafikhardware. 47

eager Initialization Das frühest mögliche Initialisieren eines Objektes, also noch bevor es benötigt wird. 41

Eckpunktnormale Die Normale eines Eckpunkts in einem Polygonnetz, sie setzt sich aus den Normalen der umliegenden Oberflächen zusammen. 33, 34

Facette Anderer Begriff für Polygone in einem Polygonnetz. 31

Farbbluten Optischer Effekt bei dem die Farbe einer angestrahlten, diffusen Oberfläche auf eine andere Oberfläche reflektiert und auf ihr sichtbar wird. 4, 5

Formfaktor Faktor der angibt welchen Anteil der Lichtausstoß einer Oberfläche auf eine andere Oberfläche hat, Symbol: F_{ij}. 2, 9, 15, 21, 24

Frustum Bezeichnet in der Computergrafik das Sichtfeld eines Betrachters und bildet in der Regel eine abgeschnittene Pyramide. 26, 34, 39

General-Purpose Computation on Graphics-Processing-Unit Bezeichnet den Einsatz von Grafikhardware zur Berechnung von beliebigen Problemen. 13, 76

GPGPU General-Purpose Computation on Graphics-Processing-Unit. 13, 14, 63, 64, 75

Host Bezeichnet in der Grafik- und GPGPU-Programmierung die CPU und ihren Arbeitsspeicher. 47

ideal diffus Eine Art der Lichtreflexion, bei der das von einer Oberfläche reflektierte Licht gleichmäßig in alle Richtungen gestreut wird. 4, 7, 8

ideal spekular Eine Art der Lichtreflexion, bei der das von einer Oberfläche reflektierte Licht an der Oberflächennormale gebrochen wird, sodass der Einfallswinkel gleich dem Ausfallswinkel ist. 4, 7

Image-Space Raum welcher mit der Auflösung eines zu rendernden Bildes übereinstimmt. 1, 22, 26, 75

Intensität Bezeichnet im Rahmen dieser Arbeit die Stärke des an einem Punkt gemessenen Lichts. 6, 8, 19, 22

Kernel In der GPGPU-Programmierung ein Programm das parallel auf der Grafikhardware ausgeführt wird. 13, 29, 47

konvex Ein Polygon das ausschließlich Eckpunkte enthält deren Innenwinkel maximal 180° betragen wird konvex genannt. 27

lazy Initialization Das Initialisieren eines Objektes bei Bedarf, also erst sobald es benötigt wird. 41

Leuchtkraft Bezeichnet im Rahmen dieser Arbeit das von einer Lichtquelle ausgestrahlte Licht, im Englischen auch Emittance genannt, Symbol: E_i. 8

Lichtanteil Bezeichnet im Rahmen dieser Arbeit den Anteil des Lichts den ein Patch der Umgebung beisteuert. Er hängt dabei vom Flächeninhalt des Patches ab. 21, 44

Lichtausstoß Bezeichnet im Rahmen dieser Arbeit das von einer Oberfläche reflektierte Licht, im Englischen auch Radiosity oder Exitance genannt, Symbol: B_i. 8, 9, 19, 76

Mip-Map Eine Textur die ein Motiv mehrmals in unterschiedlichen Auflösungen enthält. 64

Model-Space Bezeichnet einen Raum in dem alle Koordinaten relativ zu dem Ursprung eines Objekts definiert sind. 1

Normale Ein Vektor der orthogonal (also rechtwinklig) auf einer geometrischen Primitive steht. 32

Oberflächennormale Die Normale einer Oberfläche in einem Polygonnetz. 10, 18

Patch Einheit in die eine Oberfläche für die Berechnung mit dem Radiosity-Verfahren zerlegt wird, werden im Rahmen dieses Projekts als Facetten / Polygone umgesetzt. 8, 9, 15, 19, 20, 24, 37

Pixel Kurz für picture-element, bezeichnet einen Bildpunkt in einem Bild. Beim Hemicube werden die Zellen auch Pixel genannt, da für jede Seite des Halbwürfels ein Bild gerendert werden kann dessen Pixel den Hemicube-Zellen entsprechen. 7, 12, 15, 25, 26

Polygon Eine geometrische Primitive die aus mehreren Eckpunkten besteht, sie Repräsentiert eine Oberfläche in einem Polygonnetz. 26, 75

Polygonnetz Eine Datenstruktur zur Repräsentation von Objekten die aus Polygonen modelliert werden. 18–20, 31, 34

Reflektivität Bezeichnet im Rahmen dieser Arbeit die Farbe einer Oberfläche und hat direkten Einfluss auf das von ihr reflektierte Licht, im Englischen auch Reflectivity genannt. 8

Rendering Anderer Begriff für Bildsynthese. 1, 2, 17, 30, 31

Scanline Eine Zeile eines rasterisierten Bildes. 27, 40

Shader Recheneinheiten auf dem Grafikprozessor (auch Hardwareshader), kann auch ein Programm bezeichnen das von einem Hardwareshader ausgeführt wird (auch Softwareshader, Shaderprogramm oder in der GPGPU-Programmierung Kernel genannt) [1]. 12, 29

Shaderprogramm Bezeichnet ein Programm das von einem Shader ausgeführt wird (auch Shader, Softwareshader oder in der GPGPU-Programmierung Kernel genannt). 12

Szene Stellt eine Umgebung dar und ermöglicht die Interaktion mit dieser, zum Beispiel wechseln des Blickpunkts. 1, 4, 34

Szenengraph Eine Baumstruktur zur Organisation von Objekten in einer Szene. 31

Texel Kurz für Texturelement, bezeichnet einen Bildpunkt in einer Textur. 64, 65

Textur Eine zweidimensionale Datenstruktur, wird in der Computergrafik eingesetzt um zum Beispiel Bilder auf Oberflächen zu zeichnen (Skin), Tiefeninformationen zu ergänzen (Bumpmap), Reflexion zu simulieren (Environment Map) oder Beleuchtung zu speichern (Lightmap). Grafikhardware hat dedizierte Speicher für Texturen. 5, 64

Umgebung Eine Sammlung von geometrisch angeordneten Objekten die in einer Szene dargestellt werden können. 1, 4, 5, 8, 9, 34–36, 60, 64

View-Plane Die Bildebene eines Betrachtungssystems auf welches eine Szene projiziert wird. 1, 26

View-Space Bezeichnet einen Raum in dem alle Koordinaten relativ zum Betrachter definiert sind. 1, 26, 37, 75

[1] `https://de.wikipedia.org/wiki/Shader` - Abgerufen 27. März 2018

World-Space Bezeichnet einen Raum in dem alle Koordinaten relativ zum Ursprung einer Umgebung definiert sind. 1, 21, 75

Z-Puffer Auch Tiefenpuffer genannt. Ein Puffer der für jeden Pixel eine Tiefeninformation enthält. Typischerweise repräsentiert die Z-Achse in der Computergrafik die Tiefe, woher der Name Z-Puffer kommt.. 16, 65

Literaturverzeichnis

[1] Ashdown, I. : *Radiosity : A Programmer's Perspective*. John Wiley & Sons, Inc. (Wiley professional computing). http://www.gbv.de/dms/bowker/toc/9780471304883.pdf. – ISBN 0–471–30444–1. – Literaturverz. S. 477 - 488

[2] Baum, D. R. ; Rushmeier, H. E. ; Winget, J. M.: Improving Radiosity Solutions Through the Use of Analytically Determined Form-factors. In: *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '89). – ISBN 0–89791–312–4, 325–334

[3] Baum, D. R. ; Mann, S. ; Smith, K. P. ; Winget, J. M.: Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity Solutions. In: *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '91). – ISBN 0–89791–436–8, 51–60

[4] Beran-Koehn, J. C. ; Pavicic, M. J.: VI.2 - A CUBIC TETRAHEDRAL ADAPTATION OF THE HEMI-CUBE ALGORITHM. Version: 1991. http://dx.doi.org/https://doi.org/10.1016/B978-0-08-050754-5.50064-5. In: ARVO, J. (Hrsg.): *Graphics Gems II*. Morgan Kaufmann. – DOI https://doi.org/10.1016/B978–0–08–050754–5.50064–5. – ISBN 978–0–08–050754–5, 299–302

[5] Beran-Koehn, J. C. ; Pavicic, M. J.: VI.10 - DELTA FORM-FACTOR CALCULATION FOR THE CUBIC TETRAHEDRAL ALGORITHM. Version: 1992. http://dx.doi.org/https://doi.org/10.1016/B978-0-08-050755-2.50070-1. In: KIRK, D. (Hrsg.): *Graphics Gems III (IBM Version)*. Morgan Kaufmann. – DOI https://doi.org/10.1016/B978–0–08–050755–2.50070–1. – ISBN 978–0–12–409673–8, 324 - 328

[6] Blinn, J. F.: Models of Light Reflection for Computer Synthesized Pictures. In: *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '77), 192–198

[7] Campagna, S. ; Kobbelt, L. ; Seidel, H.-P. : Directed Edges—A Scalable Representation for Triangle Meshes. In: *Journal of Graphics Tools* 3 (1998), Nr. 4, 1-11. http://dx.doi.org/10.1080/10867651.1998.10487494. – DOI 10.1080/10867651.1998.10487494

[8] Chen, S. E. ; Rushmeier, H. E. ; Miller, G. ; Turner, D. : A Progressive Multi-pass Method for Global Illumination. In: *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '91). – ISBN 0–89791–436–8, 165–174

[9] Cohen, M. F. ; Chen, S. E. ; Wallace, J. R. ; Greenberg, D. P.: A Progressive Refinement Approach to Fast Radiosity Image Generation. In: *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '88). – ISBN 0–89791–275–6, 75–84

[10] Cohen, M. F. ; Greenberg, D. P.: The Hemi-cube: A Radiosity Solution for Complex Environments. In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '85). – ISBN 0–89791–166–0, 31–40

[11] Cohen, M. F. ; Greenberg, D. P. ; Immel, D. S. ; Brock, P. J.: An Efficient Radiosity Approach for Realistic Image Synthesis. In: *IEEE Computer Graphics and Applications* 6 (1986), March, Nr. 3, S. 26–35. http://dx.doi.org/10.1109/MCG.1986.276629. – DOI 10.1109/MCG.1986.276629. – ISSN 0272–1716

[12] Cohen, M. F. ; Wallace, J. R.: *Radiosity and Realistic Image Synthesis*. Academic Press, Inc., 1993. – ISBN 0–12–059756–X

[13] Cook, R. L. ; Torrance, K. E.: A Reflectance Model for Computer Graphics. In: *ACM Trans. Graph.* 1 (1982), jan, Nr. 1, 7–24. http://dx.doi.org/10.1145/357290.357293. – DOI 10.1145/357290.357293. – ISSN 0730–0301

[14] In: Coombe, G. ; Harris, M. : *Global Illumination Using Progressive Refinement Radiosity*. Addison-Wesley Professional. – ISBN 0–321–33559–7

[15] Coombe, G. ; Harris, M. J. ; Lastra, A. : Radiosity on Graphics Hardware. In: *ACM SIGGRAPH 2005 Courses*. ACM (SIGGRAPH '05)

[16] Goral, C. M. ; Torrance, K. E. ; Greenberg, D. P. ; Battaile, B. : Modeling the Interaction of Light Between Diffuse Surfaces. In: *Proceedings of the 11th Annual*

Conference on Computer Graphics and Interactive Techniques. ACM (SIGGRAPH '84). – ISBN 0–89791–138–5, 213–222

[17] Heckbert, P. S.: Adaptive Radiosity Textures for Bidirectional Ray Tracing. In: *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '90). – ISBN 0–89791–344–2, 145–154

[18] Heckbert, P. S.: Discontinuity Meshing for Radiosity. In: *Third Eurographics Workshop on Rendering*, 1992, S. 203–216

[19] Immel, D. S. ; Cohen, M. F. ; Greenberg, D. P.: A Radiosity Method for Non-diffuse Environments. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '86). – ISBN 0–89791–196–2, 133–142

[20] Kajiya, J. T.: The Rendering Equation. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '86). – ISBN 0–89791–196–2, 143–150

[21] Karimi, K. ; Dickson, N. G. ; Hamze, F. : A Performance Comparison of CUDA and OpenCL. In: *CoRR* abs/1005.2581 (2010). `http://arxiv.org/abs/1005.2581`

[22] Keller, A. : Instant Radiosity. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press/Addison-Wesley Publishing Co. (SIGGRAPH '97). – ISBN 0–89791–896–7, 49–56

[23] Lischinski, D. ; Tampieri, F. ; Greenberg, D. P.: Combining Hierarchical Radiosity and Discontinuity Meshing. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '93). – ISBN 0–89791–601–8, 199–208

[24] Maxwell, G. M. ; Bailey, M. J. ; Goldschmidt, V. W.: Calculations of the radiation configuration factor using ray casting. In: *Computer-Aided Design* 18 (1986), Nr. 7, 371 - 379. `http://dx.doi.org/https://doi.org/10.1016/0010-4485(86)90224-1`. – DOI https://doi.org/10.1016/0010–4485(86)90224–1. – ISSN 0010–4485

[25] NVidia: *CUDA Toolkit Documentation*. `http://docs.nvidia.com/cuda/`. Version: 2018

[26] Phong, B. T.: Illumination for Computer Generated Pictures. In: *Communications of the ACM* 18 (1975), jun, Nr. 6, 311–317. http://dx.doi.org/10.1145/360825.360839. – DOI 10.1145/360825.360839. – ISSN 0001–0782

[27] Rushmeier, H. E. ; Torrance, K. E.: Extending the Radiosity Method to Include Specularly Reflecting and Translucent Materials. In: *ACM Trans. Graph.* 9 (1990), jan, Nr. 1, 1–27. http://dx.doi.org/10.1145/77635.77636. – DOI 10.1145/77635.77636. – ISSN 0730–0301

[28] Sanders, J. ; Kandrot, E. : *CUDA by Example : An Introduction to General-Purpose GPU Programming*. 2. Addison-Wesley, Pearson Education http://www.gbv.de/dms/ilmenau/toc/638983517.PDF. – ISBN 978–0–13–138768–3

[29] Shao, M.-Z. ; Badler, N. I.: A Gathering and Shooting Progressive Refinement Radiosity Method. (1993)

[30] Sillion, F. ; Puech, C. : A General Two-pass Method Integrating Specular and Diffuse Reflection. In: *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '89). – ISBN 0–89791–312–4, 335–344

[31] Sutherland, I. E. ; Hodgman, G. W.: Reentrant Polygon Clipping. In: *Communications of the ACM* 17 (1974), jan, Nr. 1, 32–42. http://dx.doi.org/10.1145/360767.360802. – DOI 10.1145/360767.360802. – ISSN 0001–0782

[32] Wallace, J. R. ; Cohen, M. F. ; Greenberg, D. P.: A Two-pass Solution to the Rendering Equation: A Synthesis of Ray Tracing and Radiosity Methods. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '87). – ISBN 0–89791–227–6, 311–320

[33] Wallace, J. R. ; Elmquist, K. A. ; Haines, E. A.: A Ray Tracing Algorithm for Progressive Radiosity. In: *SIGGRAPH Comput. Graph.* 23 (1989), jul, Nr. 3, 315–324. http://dx.doi.org/10.1145/74334.74366. – DOI 10.1145/74334.74366. – ISSN 0097–8930

[34] Watt, A. H.: *3D-Computergrafik*. 3. Auflage. Pearson Studium, 2002. – ISBN 3–8273–7014–0. – Literaturverz. S. 595 - 601

[35] Whitted, T. : An Improved Illumination Model for Shaded Display. In: *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*. ACM (SIGGRAPH '79). – ISBN 0–89791–004–4, 14–

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 27. März 2018 Lars Nielsen