UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich Informatik
Arbeitsgruppe Computergrafik

Radiosity auf der Grafik-Hardware

Diplomarbeit

vorgelegt von

Frederik Nünning

Betreuer	Dipl. Inf. Thorsten Grosch
Prüfer	Prof. Dr.-Ing. Stefan Müller

August 2005

Aufgabenstellung für die Diplomarbeit
Herr Frederik Nünning
(Matrikel-Nr. 200210413)

Thema: Radiosity auf der Grafik-Hardware

Globale Beleuchtungsmodelle werden in den neusten 3D-Programmen unterstützt und benötigen enorme Rechenleistungen. Eine Art, dem Problem der hohen Rechenleistung entgegen zu treten, ist es die GPU so zu nutzen, dass sie die Berechnungen durchführt, bzw. unterstützt. Moderne GPUs warten mit großen Leistungsreserven auf und können für diese Aufgabe gut genutzt werden.

Eines dieser Beleuchtungsmodelle ist das Radiosity Verfahren. Hiermit lässt sich eine betrachterunabhängige, globale Beleuchtungssimulation in einer diffusen Umgebung erstellen. Nach der Simulation ist die Navigation in Echtzeit in der photorealistisch beleuchteten Szene möglich. Durch die schnelle Entwicklung der Grafik Hardware in den letzten Jahren ist es möglich geworden, diese Simulation auf der Grafikkarte berechnen zu lassen. Für einige Radiosity Varianten gibt es bereits Umsetzungen für die GPU, diese Verfahren sollen in dieser Diplomarbeit untersucht werden. Darauf aufbauend soll ein Konzept für eine Hierarchische Radiosity Simulation mithilfe der Grafik-Hardware entwickelt werden. Dieses Konzept soll umgesetzt und das Laufzeitverhalten mit einer Radiosity Simulation auf der CPU verglichen werden.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Untersuchung bekannter GPU-basierter Radiosity Algorithmen (Full-Matrix, Progressive Refinement)
2. Entwurf eines eigenen Verfahrens zur Umsetzung von Hierarchischem Radiosity mit der Grafik Hardware
3. Implementierung des Verfahrens
4. Testen mit verschieden komplexen Szenen und Zeitvergleich mit den CPU Verfahren
5. Dokumentation der Ergebnisse

Koblenz, den

Betreuer: Dipl.-Inf. Thorsten Grosch

- Prof. Dr. Stefan Müller -

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass die vorliegende Arbeit selbstständig verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

....., den ....

Ort, Datum Unterschrift

Danksagung

Ich möchte an dieser Stelle allen danken, die es mir ermöglicht haben in der sechs monatigen Diplomarbeitsphase meine Arbeit fertig zu stellen. Primär gilt mein Dank meinem Betreuer, Dipl. Inf. Thorsten Grosch, und dem wissenschaftlichen Mitarbeiter, Dipl. Inf. Matthias Biedermann, die mir mit ihren Ratschlägen und wichtigen Tipps jederzeit zur Verfügung standen. Des weiteren Dank an Herrn Prof. Dr. Ing. Stefan Müller, der als Prüfer und Schirmherr dieser Arbeit fungierte.

Auf der privaten Seite richte ich mich dankend an meine Familie, meine Freundin Mareike, und an meine Freunde. Sie haben mich immer wieder ermutigt, auch in schwierigen Zeiten, durchzuhalten und das Ziel im Auge zu behalten.

Des Weiteren bedanke ich mich bei den jenigen, die ich auf Grund gleicher oder ähnlicher Arbeitsgebiete kontaktiert und um nützliche Tipps, Hilfestellungen und anregende Diskussionen ersucht habe.

Frederik Nünning

Inhaltsverzeichnis

1 Einleitung **1**

2 Radiosity **3**
2.1 Beleuchtungssysteme 4
2.2 Grundlagen von Radiosity 5
2.2.1 Radiometrie & Photometrie 5
2.2.2 Reflexions-Funktion 7
2.3 Die Radiositygleichung 8
2.3.1 Formfaktoren 8
2.3.2 Radiositygleichung 12
2.3.3 Allgemeiner Radiosity Algorithmus 15

3 Radiosity-Algorithmen **16**
3.1 Direkte & Indirekte Methoden 16
3.2 Relaxationsmethoden 17
3.2.1 Jacobi-Verfahren 17
3.2.2 Gauss-Seidel-Verfahren 17
3.3 Progressive Refinement 18
3.4 Hierarchisches Radiosity 19
3.4.1 Initial Linking 22
3.4.2 Refine 23
3.4.3 Gather 23
3.4.4 PushPull 24

4 GPU Programmierung **26**
4.1 NVidias Cg Shading Language 26
4.2 CPU zu GPU 28
4.2.1 Textur als Array 28
4.2.2 Fragment Programm als Computational Kernel 29
4.2.3 Eins-zu-Eins Pixel zu Texel Mapping 29
4.2.4 Viewport-Sized Quad als Data Stream Generator 29
4.2.5 Copy-To-Texture als Feedback 30
4.3 RenderTexture 30

5 Radiosity auf der GPU **31**
5.1 Full-Matrix-Methode . . . 31
5.2 Progressive-Refinement-Methode . . . 37
5.2.1 Sichtbarkeit . . . 38
5.2.2 Formfaktoren . . . 38
5.2.3 Wahl des nächsten Senders . . . 39
5.2.4 Adaptive Unterteilung . . . 40
5.2.5 Darstellung . . . 40

6 Hierarchisches Radiosity auf der GPU **42**
6.1 Refine . . . 44
6.2 Gather . . . 45
6.3 PushPull . . . 47

7 Ergebnisse & Bewertung **49**
7.1 Ergebnisbilder . . . 49
7.2 Berechnungszeiten . . . 50
7.3 Profiling . . . 52
7.4 Zusammenfassung der Ergebnisse . . . 53

8 Fazit **55**

9 Ausblick **56**

10 Anhang & Quellcode **58**
10.1 Pseudocodebeispiele des hierarchischen Radiosity Algorithmus . . . 58
10.1.1 Refine . . . 58
10.1.2 RefinementOrakel . . . 59
10.1.3 Subdivide . . . 59
10.1.4 Gather . . . 59
10.1.5 PushPull . . . 60
10.2 Radiosity GPU-Algorithmen . . . 61
10.2.1 Drei Shadermodell zum Lösen der FM-Matrix . . . 61
10.2.2 Shader zum Berechnen von Formfaktoren . . . 62
10.2.3 Shader als Refinement-Orakel . . . 63
10.2.4 Shader für Mittelpunktberechnung . . . 64
10.2.5 Shader zum Berechnen von Gatherwerten . . . 65
10.2.6 Shader zur Addition von Gatherwerten . . . 65
10.2.7 Shader zur Addition der Gatherwerte (Push) . . . 66
10.2.8 Shader zum Berechnen der Radiosity (PULL) . . . 66

Abbildungsverzeichnis

2.1 links: **ohne Radiosity**, rechts: **mit Radiosity** 3
2.2 Graphische Darstellung der RE von Kajiya[Mül01a] 5
2.3 Graphische Darstellung des Formfaktor Integrals[Mül01d] 9
2.4 Veranschaulichung des Nusselt-Analogon[Mül01d] 10
2.5 Skizze zur Prisma-Formfaktorberechnung[Mül01d] 11
2.6 Skizze zur zweiten Formfaktorvereinfachung[Mül01d] 12
2.7 Skizze zur Radiositygleichung[Mül01d] 13

3.1 Pseudocode des Jacobi-Algorithmus 17
3.2 Pseudocode des Gauss-Seidel-Algorithmus 18
3.3 Skizze Gather- und Shootalgorithmus 18
3.4 Skizze zur 2-Level Hierarchie[Mül01b] 20
3.5 Radiosity mit Multilevel-Hierarchie, Datenstruktur Quadtree[Mül01b] . . 21
3.6 Schritt 1 HR: Initiales Verlinken[Mül01b] 22
3.7 Schritt 2: Refine[Mül01b] 23
3.8 Schritt 3: Gather[Mül01b] 23
3.9 Schritt 4: PushPull[Mül01b] 24
3.10 Darstellung der HR-Links auf verschiedenen Hierarchieleveln[Mül01b] . . 25

4.1 Skizze der Grafikpipeline mit Vertex und Fragmentdaten[FK03] 27
4.2 Skizze einer programmierbaren Pixelpipeline[FK03] 28

5.1 Radiositylösung der Cornellbox-Szene[CHH03] 32
5.2 Entwurf eines FM Algorithmus auf der GPU 34
5.3 Anfangs- und konvergierte Szene der FM-Methode auf der GPU 35
5.4 Full-Matrix GPU-Methode als drei Shader Modell 36
5.5 Beispielszene berechnet mit Progressive Refinement auf der GPU[CjHL04] 37
5.6 FF zwischen Differntialfläche *di* und Polygon j[CjHL04] 39
5.7 Szene mit 260.000 Elementen berechnet mit PR auf der GPU[CjHL04] . . 41

6.1 Modell zur Umsetzung von HR auf der GPU 43

7.1 Szene mit 75.000 Polygonen berechnet mit HR auf der GPU 49
7.2 Szene mit 75.000 und 1.700 Polygonen 50
7.3 Wireframe-Ansicht der Szene aus Abbildung 7.1 50

1 Einleitung

Die Entwicklung der Computerwelt erfolgt so rasant, dass es kaum möglich ist, alle neuen Entwicklungen und Trends verfolgen zu können. In all unseren Lebenslagen helfen uns Computer, alltägliche Situationen besser oder vielleicht auch nur einfacher bzw. schneller zu meistern. Moderne Computer, vernetzt mit dem eigenen Wohnhaus, sind in der Lage, fremde Personen an der Tür zu empfangen, darauf zu achten, wann es regnet, um eventuell offen stehende Fenster zu schliessen, oder das Bad um halb sieben abends, nach der Arbeit, einzulassen.

Diese Fähigkeiten werden der mittlerweile enormen Rechenleistung, die heutige Prozessoren den Rechnern verleihen, verdankt. Nach dem Mooreschen Gesetz[1] verdoppelt sich die Anzahl der Transistoren eines Prozessors in 18 bis 24 Monaten. Somit wird die Komplexität eines Prozessors immer größer, womit auch eine Erhöhung der Leistungsfähigkeit und des Funktionsumfanges einhergeht. Die vorliegende Arbeit beschäftigt sich mit dem Einsatz einer Technologie, deren Rechenleistung sich im Vergleich zu Prozessoren innerhalb von ca. sechs Monaten verdoppelt, den sogenannten GPUs[2]. GPUs sind die Hauptkomponente einer Grafikkarte, ähnlich den Prozessoren, CPUs [3], auf den Motherboards, sind aber nicht für den gleichen Einsatz gedacht. Eine GPU (Graphics Prozessing Unit bzw. Grafik-Prozessor-Einheit) beschleunigt die Darstellung von 3D-Szenen mit gleichzeitiger Berechnung von Effekten, wie z.B. Licht, Transparenzen und Verformungen.

Mooresches Gesetz

Wenn diese Einheit doch so schnell ist, warum wird sie nicht anstelle der CPU eingesetzt? Das Einsatzgebiet einer GPU ist die Berechnung von 3D-Koordinaten sowie von Farbwerten und sie ist für diese Aufgabe hoch optimiert. Daher ist die Bearbeitungs-, Daten- und Befehlsstruktur eine ganz andere als die der CPU. Aus diesem Grunde lassen sich Programme, die für eine CPU erstellt wurden, nicht auf der GPU ausführen. Hierzu ist eine Portierung der Algorithmen mit gleichzeitiger Anpassung an die Strukturen einer GPU nötig. Diese Portierung ist nicht einfach und erfordert spezielles Umdenken, um genau das gleiche Ergebnis zu erhalten, sofern es überhaupt, aufgrund der verschiedenen Grundstrukturen, möglich ist. Warum sollte dann der Aufwand betrieben werden die Programme zu portieren? Dies hat mit der zuvor genannten Eigenschaft zu tun, dass Grafikkarten über eine enorme Rechenleistung verfügen und daher in der Lage sind Rechenoperationen schneller durchzuführen als herkömmliche Prozessoren. Dabei sind aber einige Dinge zu beachten, die später im Kapitel 4 auf Seite 26 angesprochen und erläutert werden.

CPU vs GPU

Durch das Streben, immer schnellere Computer zu bauen, ist in der Computergrafik der Wunsch entstanden, computergenerierte Szenen schneller berechnen zu können und

[1] auch bekannt als Moore's Law
[2] Graphics Processing Unit
[3] Central Processing Unit

vor allem photorealistisch aussehen zu lassen. Photorealistische Szenen zeichnen sich dadurch aus, dass diese kaum oder gar nicht von Photografien der realen Welt unterscheidbar sind. Hierzu gehören z.B. Berechnungen der Lichtsimulation, Schattenwurf und Detailgetreue. Unter Lichtsimulationsberechnungen werden verschiedene Tatsachen zusammengefasst. Lichtsimulationen beschäftigen sich damit, wie sich „Licht“ innerhalb von Räumen „verhält“ oder was passiert, wenn Licht mit Objekten interagiert. Die Interaktion von Licht mit einem Glas Wasser lässt auf dem Tisch einen hellen Fleck entstehen, in der Fachsprache genannt Kaustik. Mit Lichtsimulation in Räumen ist die direkte und indirekte Lichtverteilung gemeint. So strahlt eine Lichtquelle direkt bestimmte Elemente in einem Raum an. Dabei ist zu beachten, dass Licht z.B. über Reflexionen an andere Elemente im Raum weitergegeben wird. Dieses Licht nennt man indirektes Licht, da es nicht direkt von einer Lichtquelle ausgestrahlt wurde.

Photorealistische Computergrafik

Es gibt zwei Arten von Algorithmen, die Lichtsimulationen berechnen können. Zum einen sind es lokale und zum anderen globale Beleuchtungssysteme. Globale Beleuchtungssysteme erfordern ein sehr viel höheres Maß an Berechnungen als lokale. Lokale Beleuchtungssysteme berücksichtigen nur bestimmte Formen von Lichtwegen. Globale Beleuchtungssysteme hingegegen berücksichtigen die Eigenschaften der Szene, die der Lichtquellen und Objekteigenschaften. Das Kapitel 2.1 beschreibt ausgiebig die Eigenschaften und Unterschiede der Beleuchtungssysteme. Eines der bekanntesten globalen Beleuchtungsmodelle heisst **Radiosity**.

Globale Beleuchtung

Die beiden soeben vorgestellten Themen Radiosity und GPU sind Grundlage dieser Diplomarbeit. Sie beschäftigt sich mit der Darlegung und Umsetzung von Radiosity-Methoden auf die Befehls- und Datenstruktur einer GPU. Dazu ist es nötig, die Programmierung einer Shading-Sprache zu erlernen und die Radiosity-Methoden auf der CPU zu studieren. Im Anschluss werden diese Algorithmen in computergenerierten Szenen getestet und die Geschwindigkeit ausgewertet. Zudem wird die Situation beleuchtet, wie ein Algorithmus für die GPU umzustellen ist, welche Probleme es zu lösen gilt und ob die Umsetzung eines solchen Algorithmus auf die GPU generell vernünftig erscheint.

Radiosity auf der GPU

2 Radiosity

Was ist Radiosity und wie funktioniert es im Detail? Dieses Kapitel beschreibt die grundlegenden physikalischen Konzepte als Basis für Radiosity und geht auf verschiedene Lösungsmodelle ein. Diese werden in Form verschiedener Algorithmen vorgestellt.
Radiosity wird heutzutage eingesetzt, um globale Beleuchtung in computergenerierten Szenen zu ermöglich. So wird Radiosity z.B. in professionellen Softwarepaketen wie 3D Studio Max, Maya und Maxon Cinema4D eingesetzt. Globale Beleuchtungen lassen Szenen photorealistisch und natürlich erscheinen. In Zukunft wird es sogar Computerspiele geben, die globale Beleuchtung unterstützen werden. Im Moment ist die Rechenleistung heutiger Desktop-PCs aber noch zu gering, um globale Beleuchtung in Echtzeit und serienreif einsetzen zu können. Die Abbildung 2.1 zeigt den Einsatz von Radiosity. Die mit

Radiosity in kommerzieller Software

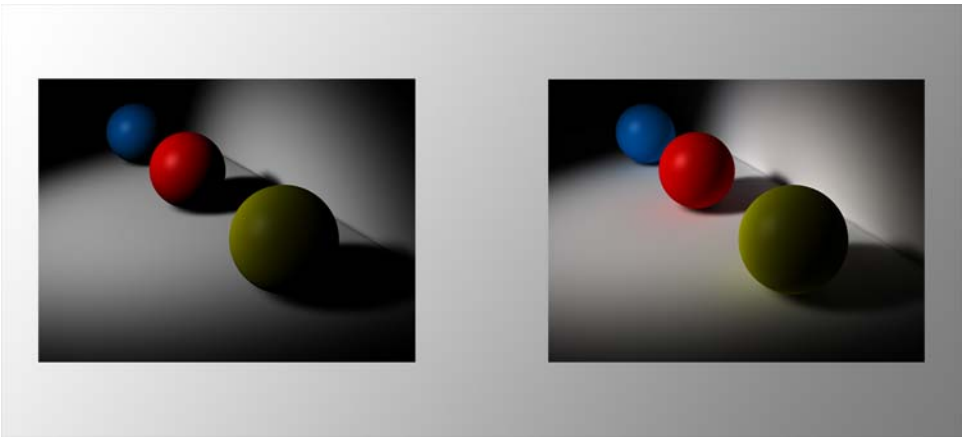


Abbildung 2.1: links: **ohne Radiosity**, rechts: **mit Radiosity**

Radiosity berechnete Szene wirkt „echter“ als ihr Pendant ohne Radiosity, da in dieser Szene alle Patches als Sender fungieren. Dies bedeutet, auch Flächen, die in ihrem Sinne keine Lichtquellen sind, treten als diese auf. Primär wird der direkte Lichtweg berechnet. Das bedeutet, Lichtquellen, wie z.B. Lampen oder Objekte, die Licht ausstrahlen, geben ihre Lichtenergie an ihre Umgebung ab. Sekundär wird die Reflexion des Lichtes direkter Lichtwege an anderen Objekten berechnet.
Als Beispiel kann ein durch ein kleines Fenster beleuchteter Kellerraum dienen. Es wird erwartet, dass das einfallende Licht nicht nur den Teil erhellt, auf den das Licht auf geradem Weg fällt, sondern auch den gesamten Raum geringfügig erhellt. Ausgehend von der primären Lichtquelle werden einige Teile des Kellers erhellt. Vielleicht sogar sehr stark. Somit entstehen in diesem Raum weitere Lichtquellen, die ihre Lichtenergie

Beispiel für Radiosity

abgeben können. Dieses Verteilen von Lichtenergie auf dem 2. Lichtweg wird „Verteilen indirekten Lichtes“ genannt. Dieses kann auf mehreren weiteren Lichtwegen fortgesetzt werden.
Das Berechnen der indirekten Lichtverteilung zwischen allen Elementen einer Szene ist sehr kostenintensiv. Bei n Patches in einer Szene müssen n^2 Lichtverteilungen betrachtet werden. Da alle Objekte einer Szene betrachtet werden, wird die indirekte Beleuchtung auch als das „*global illumination problem*“ bezeichnet.[CW93]

2.1 Beleuchtungssysteme

Lokale Beleuchtungssysteme stellen die Basis der Beleuchtung einer computergenerierten Szene dar. Ein solches Verfahren heisst **Scanline-Rendering** und wird mit folgender Gleichung umgesetzt:

$$I = d \cdot I_a + \sum_{i=1}^{\#Lq} (d \cdot cos\varphi_i + \rho \cdot cos^n \psi_i) \cdot I_{L_i} \tag{2.1}$$

Die Lichtintensität I wird berechnet mit dem spiegelnden und reflektierenden Reflexionsgrad, ρ und φ, dem ambienten Licht I_a, sowie mit den Energien der sich in der Szene befindlichen Lichtquellen I_{L_i}. Über eine Summierung über alle Lichtquellen $\#Lq_i$ wird die Gesamtenergie berechnet. Mit diesem Beleuchtungsmodell können Lichtwege von der Lichtquelle auf eine diffuse Fläche zum Auge berechnet werden. Alle anderen Lichtwege können nur simuliert werden. Weitere wichtige interessante Lichtwege sind: LA, LSA, LDSA,

Lokale Beleuchtungsmodelle

die Kürzel stehen für L=Lichtquelle, D=diffuse Fläche, S=spiegelnde Fläche, A = Auge.
Das **Ray-Tracing Beleuchtungsmodell** berechnet mithilfe von Schattenfühlern mehrere Lichtwege. Es können harte Schatten und LDS*A Lichtwege berechnet werden. Die Formel ist dem Scanline-Verfahren entnommen und um einen Schattenfühler δ_i und sowohl reflektierten als auch transmittierte Lichtintensitäten erweiterte:

$$I = d \cdot I_a + \sum_{i=1}^{\#Lq} (d \cdot cos\varphi_i + \rho \cdot cos^n \psi_i) \cdot I_{L_i} \cdot \delta_i + \rho \cdot I_R + \tau \cdot I_T \tag{2.2}$$

Der Ansatz von **globalen Beleuchtungssystemen** liegt nun darin, die Lichtwege LD*A[1], LS*DA und Halbschatten berechnen zu können, da diese mithilfe lokaler Beleuchtungssysteme nicht zu lösen sind. Aus dieser Überlegung geht die Rendering Equation von Kajiya aus dem Jahr 1986 hervor:

Rendering Equation

$$L_o\left(dA_e, d\vec{\omega}_o\right) = L_e\left(dA_e, d\vec{\omega}_o\right) + \int_{2\pi} \rho\left(dA_e, d\vec{\omega}_i, d\vec{\omega}_o\right) \cdot L_i\left(dA_e, d\vec{\omega}_o\right) \cdot cos\theta_i \cdot d\omega_i \tag{2.3}$$

Die Leuchtdichte L_o, die in Richtung $\vec{\omega}_o$ abgegeben wird, wird bestimmt durch die Emmissionsenergie L_e des Patches in Richtung $\vec{\omega}_o$ und die Summe aller eingehenden Leuchtdichte $L_i\left(\vec{\omega}_i\right)$ unter allen Raumwinkeln beschrieben durch die obere Halbkugel über dem Patch. $\rho\left(dA_e, d\vec{\omega}_i, d\vec{\omega}_o\right)$ steht für die Bidirektionale Reflexionsverteilungsfunktion.

[1] D* = beliebig viele diffuse Flächen hintereinander

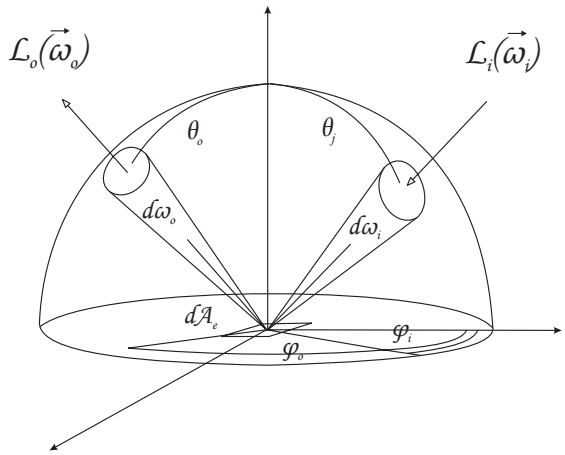


Abbildung 2.2: Graphische Darstellung der RE von Kajiya[Mül01a]

Zusammenfassung der unterstützten Lichtwege der vorgestellten Methoden:

	Scanline Rendering	RayTracing	**Radiosity**	PhotonMapping
LA	nein	nein	ja	nein
LDA	ja	ja	ja	ja
LSA	nein	nein	nein	nein
LDSA	nein	ja	nein	ja
LD*A	nein	nein	**ja**	ja
LDS*A	nein	nein	nein	ja

Globale Beleuchtungsmodelle

Zur Lösung dieser Rendering Equation existieren verschiedenste Lösungsansätze, wie z.B. Distributed Ray Tracing, Path-Tracing, Bi-Directional Ray Tracing, Photon Mapping und **Radiosity**. In Kapitel 2.3 wird erläutert wie die Radiositygleichung aus der Rendering Equation hergeleitet wird. [Mül01a]

2.2 Grundlagen von Radiosity

2.2.1 Radiometrie & Photometrie

Radiometrie ist die Wissenschaft des messbaren Austauschs von elektromagnetischer Strahlungsenergie. Dieser Strahlungsaustausch kann mithilfe von physikalischen Einheiten bzw. radiometrischen Einheiten, beschrieben und mit speziellen Equipment gemessen werden. Die Computergrafik besitzt die photometrischen Einheiten als Pendant zu den radiometrischen Einheiten der Physik. Diese Elemente beschreiben die Wahrnehmungspsychologie, um subjektive Eindrücke, die durch Beleuchtung entstehen, vorherzusagen. Die **Photometrie** beschäftigt sich mit der Wirkung des elektromagnetischen Spektrums auf ein visuelles Empfängersystem unter Verwendung physiologischer Aspekte. Zu jeder radiometrischen Einheit existiert ein photometrisches Pendant.

Eine beliebige radiometrische Größe $X(\lambda)$ wird in ihr photometrisches Pendant X überführt durch Gewichtung mit der $V(\lambda)$-Kurve [2] und K_m [3]:

$$X = K_m \int_{380nm}^{780nm} X(\lambda) \cdot V(\lambda)\, d\lambda \tag{2.4}$$

Strahlungsmenge $Q(\lambda)$ **/ Lichtmenge** Q Die Strahlungsmenge gibt den gesamten Energieverlust des sichtbaren elektromagntischen Spektrums einer Lichtquelle an. Q berechnet sich aus der $Q(\lambda)$ und der $V(\lambda)$-Gewichtung:

$$Q = K_m \int_{380nm}^{780nm} Q(\lambda) \cdot V(\lambda)\, d\lambda \tag{2.5}$$

Strahlungsfluss $\Phi(\lambda)$**, Lichtstrom** Φ Der Quotient aus der Strahlungsmenge $Q(\lambda)$ und der Zeit t ergibt den Strahlungsfluss $\Phi(\lambda)$. Dieser ist ein Maß für die Leistung einer Lichtquelle:

$$\Phi = \frac{dQ}{dt} \tag{2.6}$$

Photometrische Grundgrößen

Strahlstärke $I(\lambda)$ **/ Lichtstärke** I Der Quotient aus dem in einer bestimmten Richtung ausgesandten Strahlungsfluss und dem durchstrahlten Raumwinkel beschreibt die Strahlstärke $I(\lambda)$:

$$I = \frac{d\Phi}{d\omega} \tag{2.7}$$

Raumwinkel ω Der Raumwinkel ω ist eine Erweiterung des zweidimensionalen Winkels im Bogenmaß und wird beschrieben durch das Verhältnis der bedeckten Kugelkalotte zum Quadrat des Kugelradius:

$$\omega = \frac{A_k}{r^2} \tag{2.8}$$

Strahldichte $L(\lambda)$**, Leuchtdichte** L Das photometrische Pendant der Strahldichte, die Leuchtdichte beschreibt die Helligkeitswahrnehmung des menschlichen Auges. Die Leuchtdichte ist der Quotient der Lichtstärke durch den sichtbaren Anteil der Senderfläche:

$$L = \frac{dI}{dA \cdot cos\theta} \tag{2.9}$$

Bestrahlungsstärke $E(\lambda)$**, Beleuchtungsstärke** E Die Bestrahlungsstärke $E(\lambda)$ ist der auftretende Strahlungsfluss Φ pro Flächenelement dA_e:

$$E = \frac{d\Phi}{dA_e} \tag{2.10}$$

[2] Hellempfindlichkeitsgrad
[3] K_m = photometrisches Strahlungsequivalent $\left(683 lm \cdot W^{-1}\right)$

Spezifische Lichtausstrahlung (Radiosity) $B(\lambda)$**,** B Die Spezifische Lichtausstrahlung $B(\lambda)$ ist der abgegegebene Strahlungsfluss Φ pro Flächenelement dA_s der leuchtenden Fläche:

$$B = \frac{d\Phi}{dA_s} \tag{2.11}$$

2.2.2 Reflexions-Funktion

Verschiedene Materialien, wie z.B. Holz, Marmor oder Glas, haben unterschiedliche Lichtreflexionseigenschaften und werden damit verschieden wahrgenommen. Das Konzept der Reflexion beschreibt die Eigenschaften des reflektierten Lichts von einem Material.

BRDF Die bidirektionale Reflexionsverteilungsfunktion[4] ist grundlegend die Funktion, die man mit Reflexion direkt in Verbindung bringt. Sie beschreibt für alle Einfalls- und Ausfallswinkel die ausgestrahlte Leuchtdichte im Verhältnis zur eingestrahlten Beleuchtungsstärke.

BRDF und ihre Eigenschaften

$$f_r(d\vec{\omega}_i, d\vec{\omega}_o) = \frac{dL_o(d\vec{\omega}_o)}{dE_i(d\vec{\omega}_i)} \tag{2.12}$$

Damit die BRDF physikalisch plausibel ist, müssen die drei folgenden Eigenschaften erfüllt sein:

1. die BRDF ist nicht negativ
2. die BRDF erfüllt den Energieerhaltungssatz - *Licht wird reflektiert und nicht erzeugt*
3. die BRDF erfüllt die Helmholtzreziprozität - *Der Lichtstrom ist vorwärts wie rückwärts identisch*: $f_r(d\vec{\omega}_i, d\vec{\omega}_o) = f_r(d\vec{\omega}_o, d\vec{\omega}_i)$

Die Einheit der BRDF ist sr^{-1} und nimmt Werte im Bereich $[0, \infty[$ an.

Der Reflexionsgrad [5] Ähnlich zur BRDF ist der Reflexionsgrad ρ definiert. Anstatt des Wertebereichs $[0, \infty[$ der BRDF liegt der Wertebereich von ρ im Bereich $[0, 1]$ und ist speziell für diffuse Umgebungen geeignet. Er ist definiert durch das Verhältnis zwischen dem gesamten ausfallenden Lichtstrom zum gesamten eingestrahlten Lichtstrom:

$$\rho = \frac{\Phi_o}{\Phi_i} = \frac{\int_{2\pi} L_o(d\vec{\omega}_o) \cdot cos\theta_o \cdot d\omega_o}{\int_{2\pi} L_i(d\vec{\omega}_i) \cdot cos\theta_i \cdot d\omega_i} \tag{2.13}$$

[4] Bidirectional Reflectance Distribution Function
[5] Reflectance

Diffuse Lambert-Reflexion Die diffuse Reflexion sagt aus, dass das Licht in alle Richtungen unabhängig von der Einfallsrichtung gleich gestreut wird, mit anderen Worten, die BRDF ist konstant. Bei der Lambert-Reflexion werden die angeleuchteten Materialien zu Lambertstrahlern. Das bedeutet, dass die reflektierte Leuchtdichte proportional zur einfallenden Beleuchtungsstärke ist und sie ist in alle Raumrichtungen gleich und konstant :

$$\begin{aligned} L_o\left(d\vec{\omega}_o\right) &= \int_{2\pi} f_r\left(d\vec{\omega}_i, d\vec{\omega}_o\right) \cdot L_i\left(d\vec{\omega}_i\right) \cdot cos\theta_i \cdot d\omega_i \\ L_o &= f_{r,d} \cdot \int_{2\pi} L_i\left(d\vec{\omega}_i\right) \cdot cos\theta_i \cdot d\omega_i \\ L_o &= f_{r,d} \cdot E \end{aligned} \tag{2.14}$$

Der Reflexionsgrad für die diffuse Lambert-Reflexion berechnet sich durch Einsetzen in die Gleichung für ρ:

Zusammenhang zwischen BRDF, Reflexionsgrad und diffuser Lambert-Reflexion

$$\rho = \frac{\Phi_o}{\Phi_i} = \frac{\int_{2\pi} L_o\left(d\vec{\omega}_o\right) \cdot cos\theta_o \cdot d\omega_o}{\int_{2\pi} L_i\left(d\vec{\omega}_i\right) \cdot cos\theta_i \cdot d\omega_i} \tag{2.15}$$

$$= \frac{L_o \cdot \int_{2\pi} cos\theta_o \cdot d\omega_o}{E}$$

$$= \frac{L_o \cdot \pi}{E} \tag{2.16}$$

Durch Einsetzen der BRDF-Gleichung $L_o = f_{r,d} \cdot E$ folgt daraus für ρ:

$$\rho = f_{r,d} \cdot \pi \tag{2.17}$$

Daraus ergibt sich ein Relexionsgrad ρ für diffuse Lambertstrahler mit:

$$\rho = \frac{\Phi_o}{\Phi_i} = \frac{B}{E} \tag{2.18}$$

Der Reflexionsgrad ist, unter der Annahme der diffusen Lambert-Reflexion, dimensionslos und unabhängig von den Einfalls- und Ausfallswinkeln. Eine Lambertoberfläche erscheint aus allen Blickwinkeln gleich hell. Die BRDF eines diffusen Lambertstrahlers ist für alle Einfalls- und Ausfallswinkel konstant und hat die Einheit sr^{-1}:

$$f_r\left(d\vec{\omega}_i, \vec{\omega}_o\right) = \frac{\rho}{\pi} \tag{2.19}$$

2.3 Die Radiositygleichung

2.3.1 Formfaktoren

Um bei einem Strahlungsaustausch zwischen zwei Flächen die Strahlung ermitteln zu können, einem Patch an ein anderes abgegeben wird, ist etwas nötig, diesen Anteil zu berechnen bzw. zu definieren. Ein Formfaktor zwischen zwei Flächen gibt diesen Anteil

an. Beruhend auf den beiden Radiosityannahmen, erstens der komplett diffusen Szene und zweitens der konstanten Leuchtdichten auf einem Patch [6], ermittelt man den Formfaktor zwischen Patch s und Patch e mit:

$$F_{se} = \frac{1}{A_s} \cdot \int_{A_s} \int_{A_e} \frac{cos\theta_s \cdot cos\theta_e \cdot dA_s \cdot dA_e}{\pi \cdot d^2} \tag{2.20}$$

Formfaktor als Maß für Strahlungsaustausch zweier Flächen

Ein Formfaktor beschreibt den Anteil der von einem Patch an ein anderes Patch abgegebenen Energie. Mithilfe zweier Schleifen jeweils über die Empfänger- und Senderfläche, wird pro Element der Flächen, die Distanz und die Winkel zwischen den Normalen und dem Distanzvektor berechnet, die Werte aufsummiert und anschließend durch die Fläche des Senders dividiert. Formfaktoren besitzen folgende Eigenschaften:

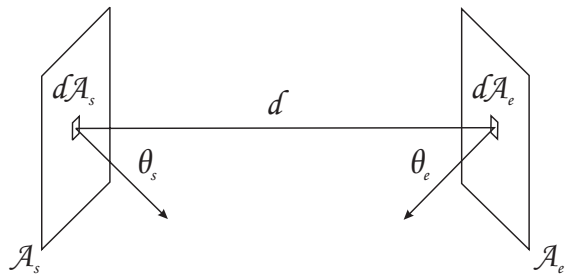


Abbildung 2.3: Graphische Darstellung des Formfaktor Integrals[Mül01d]

- Wertebereich von $[0; 1]$
- ist abhängig von der Geometrie der beiden Flächen
- Reziprozität: $F_{es} \cdot A_e = F_{se} \cdot A_s$
- keine Selbstbeleuchtung: $\forall\,(s)\, F_{ss} = 0$
- geschlossene Umgebungen: $\forall\,(s) \sum_{e=1}^{\#Flaechen} F_{se} = 1$
- offene Umgebungen: $\forall\,(s) \sum_{e=1}^{\#Flaechen} F_{se} \leq 1$

Visibilität zwischen zwei Patches

Bis jetzt wird bei der Formfaktorberechnung lediglich darauf geachtet, dass der Anteil der Energie von einem Patch zu einem anderen Patch, im nicht verdeckten Fall, berechnet wird. Sollten sich zwei Patches nicht „sehen“, da ein Objekt zwischen ihnen liegt, wird der Fall nicht betrachtet und der Formfaktor trotzdem berechnet. Daher wird das Formfaktor-Integral um eine Visibilitätsabfrage erweitert. Kann ein Patch seine Energie nicht an ein bestimmtes anderes Patch abgeben, so ist der Visibilitätsterm $V\,(dA_s, dA_e)$ und somit der Formfaktor F_{se} null. Ein solcher Formfaktor wird auch „vollständig“ genannt:

$$F_{se} = \frac{1}{A_s} \cdot \int_{A_s} \int_{A_e} \frac{cos\theta_s \cdot cos\theta_e \cdot dA_s \cdot dA_e}{\pi \cdot d^2} \cdot V\,(dA_s, dA_e) \tag{2.21}$$

[6] constant radiosity assumption

$V(dA_s, dA_e)$ ist der Sichtbarkeitsterm, der entweder 0 bei Verdeckung oder 1 bei keiner Verdeckung annimmt.

Nusselt-Analogon Das Nusselt-Analogon beschreibt einen Formfaktor zwischen einer infinitesimalen kleinen Fläche dA_s zu einer Fläche A_e[7][Fec96]:

$$F_{dA_s A_e} = \int_{A_e} \frac{cos\theta_s \cdot cos\theta_e}{\pi \cdot d^2} \cdot V(dA_s, A_e) \cdot dA_e \tag{2.22}$$

Nusselt-Analogon

Die Fläche A_e wird auf die Oberfläche der Einheitshalbkugel mit Mittelpunkt auf der infinitesimal Fläche A_s projeziert. Die so entstandene Fläche auf der Halbkugel wird auf die Grundfläche des Halbkreises projeziert. Der resultierende Flächeninhalt wird durch den Flächeninhalt des Einheitskreises dividiert und liefert so das Verhältnis zwischen der projizierten Fläche und dem Einheitskreis.

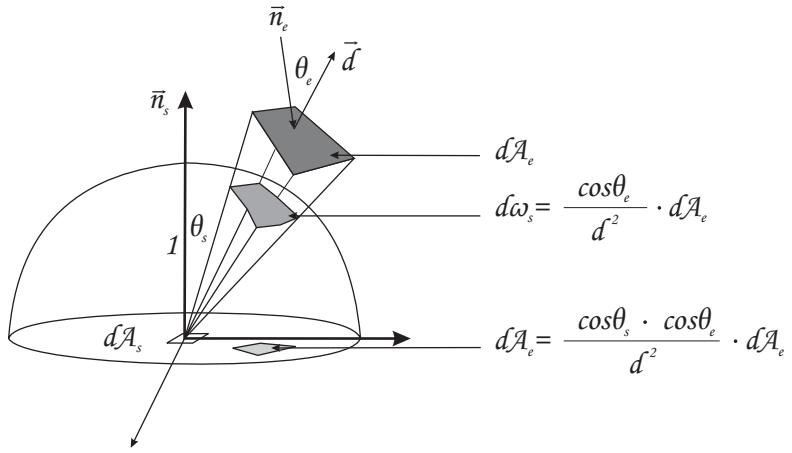


Abbildung 2.4: Veranschaulichung des Nusselt-Analogon[Mül01d]

Prisma-Formfaktor Das Prisma-Verfahren berechnet, wie das Nusselt-Analogon, Formfaktoren der ersten Formfaktorvereinfachung, also Punkt zu Fläche Formfaktoren. Über den Sender wird ein Prisma gespannt, mit dessen Seiten- und Winkelinformationen es ermöglichen, den Formfaktor mit der Formel [Gla95]

Prisma-Formfaktor

$$F_{dA_s A_e} = -\frac{1}{2\pi} \sum \vec{n_s} \circ \vec{\gamma_i} \tag{2.23}$$

zu berechnen. Die Vektoren $\vec{\gamma_i}$ stehen jeweils senkrecht auf den Prismenflächen und haben die Länge, die gleich dem Betrag des Winkels γ ist. $\vec{n_s}$ ist die normierte Normale der Senderfläche.

Die Vektoren $\vec{\gamma_i}$ werden wie folgt berechnet: Die auf den Pyramidenseiten senkrecht

[7] Punkt-zu-Fläche Formfaktor

stehenden normierten Vektoren $\vec{g_i}$ werden durch Division des Kreuzprodukts der Kantenvektoren r_{i+1} und r_i durch dessen Länge erhalten:

$$\vec{g_i} = \frac{\vec{r_i} \times \vec{r_{i+1}}}{|\vec{r_i} \times \vec{r_{i+1}}|} \tag{2.24}$$

Der Winkel γ_i wird bestimmt mit:

$$\gamma_i = arccos \frac{\vec{r_i} \circ \vec{r_{i+1}}}{|\vec{r_i}| \cdot |\vec{r_{i+1}}|} \tag{2.25}$$

Der gesuchte Vektor $\vec{\gamma_i}$ ergibt sich aus dem Produkt des Vektors $\vec{g_i}$ mit dem Skalar γ_i:

Berechnung des Prisma-Formfaktors

$$\vec{\gamma_i} = \gamma_i \cdot \vec{g_i} \tag{2.26}$$

Mithilfe der Vektoren $\vec{\gamma_i}$, die nun berechnet werden können, kann der Prisma Formfaktor ermittelt werden. Die Grafik 2.5 zeigt eine Skizze zur Anwendung der Prisma-Methode.

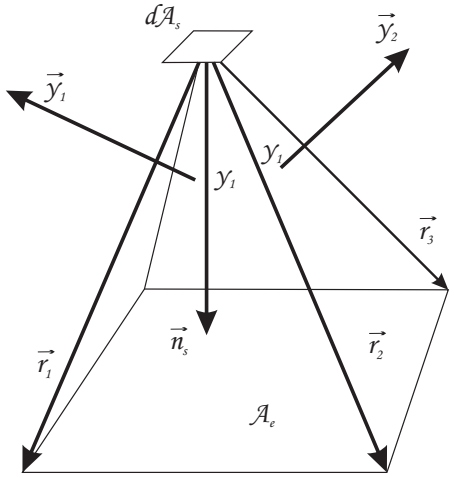


Abbildung 2.5: Skizze zur Prisma-Formfaktorberechnung[Mül01d]

Zweite Formfaktor-Vereinfachung Mithilfe der zweiten Formfaktor-Vereinfachung lassen sich Punkt-zu-Punkt Formfaktoren auf einfache Weise berechnen. Ein Distanzvektor von einem Punkt (x_s, y_s, z_s) des Senderpatches dA_s zu einem Punkt (x_e, y_e, z_e) des Empfängerpatches dA_e wird aufgespannt und an diesen Punkten die normierten Normalen $\vec{n_s}$ und $\vec{n_e}$ bestimmt. Aus der Gleichung

$$F_{dA_s dA_e} = \frac{\left(\vec{d} \circ \vec{n_s}\right) \times \left(-\vec{d} \circ \vec{n_e}\right)}{\pi \cdot d^4} \cdot \Delta A_e \tag{2.27}$$

mit

Zweite Formfaktor-Vereinfachung

$$d \cdot cos\theta_s = \vec{d} \circ \vec{n}_s \tag{2.28}$$

$$d \cdot cos\theta_e = -\vec{d} \circ \vec{n}_e \tag{2.29}$$

ergibt sich die endgültige Berechnungsformel:

$$F_{dA_s dA_e} = \frac{cos\theta_s \cdot \cos\theta_e}{\pi \cdot d^2} \cdot \Delta A_e \tag{2.30}$$

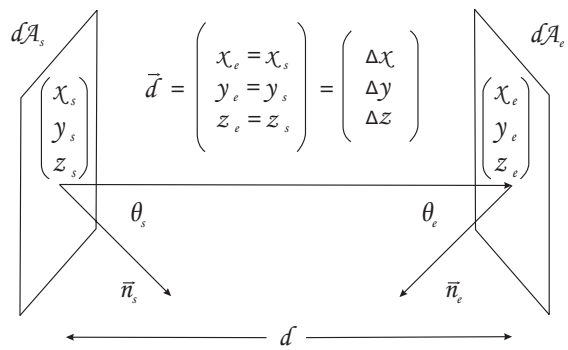


Abbildung 2.6: Skizze zur zweiten Formfaktorvereinfachung[Mül01d]

2.3.2 Radositygleichung

Die Herleitung der Radositygleichung, ausgehend von der Rendering Equation, erfolgt unter der Annahme der finiten Elemente. Das bedeutet, dass die Szene in kleine Patches unterteilt wird. Die Patches, uniforme Elemente [8], werden so klein gewählt, dass die Radiosity auf den Patches als konstant angenommen wird. Des Weiteren wird die Szene als komplett diffus angenommen, da Radiosity nur in der Lage ist, diffuse Reflexionen berechnen zu können. Mittlerweile existiert auch die Möglichkeit, Radiosity in Verbindung mit diffusen, spekularen und transluzenten Flächen einzusetzen.
Die aus der Rendering Equation abgeleitete Radositygleichung wird beschrieben durch die Formel:

$$B_e = E_e + \rho_e \sum_{s=1}^{n} B_s \cdot F_{es} \tag{2.31}$$

Aufbau der Radositygleichung

- B_e = Ausgestrahlte Radiosity
- E_e = Eigenemission
- ρ_e = Reflexionsgrad
- B_s = Eingestrahlte Radiosity

[8] Die Radiosity ist auf dem Element überall gleich

- F_{es} = Formfaktor (Empfängerfläche zur Senderfläche)

und ist graphisch in der Abbildung 2.7 verdeutlicht. Die ausgestrahlte Radiosity eines Patches e wird berechnet aus der Eigenemission des Patches und der durch ρ bestimmten Radiosity, die durch die Umgebungspatches eingestrahlt wird. Die eingestrahlte Radiosity wird durch den Formfaktor zwischen der Empfängerfläche und der Senderfläche gesteuert.

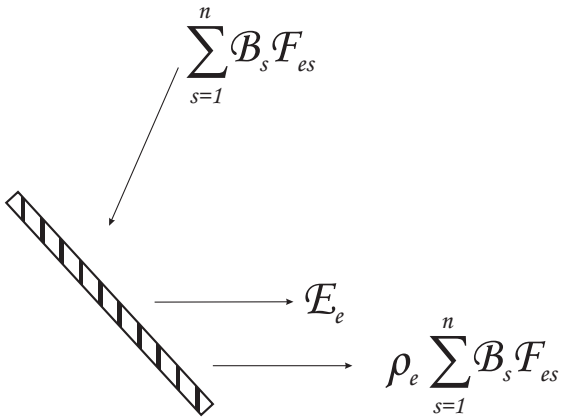


Abbildung 2.7: Skizze zur Radiositygleichung[Mül01d]

Lösungsmöglichkeiten der Radiositygleichung Folgende Möglichkeiten zur Lösung der Radiosity-Gleichung können verwendet werden:

Lösungen für die Radiosity-gleichung

- **Radiosity** (finite Elemente)
- Photon Mapping
- Path-Tracing
- Bidirectional Raytracing

Aus der Radiositygleichung entsteht durch Umformen das Radiosity Gleichungssystem, welches es zu lösen gilt:

$$M \cdot \vec{B} = \vec{E} \tag{2.32}$$

Vektoren $\vec{B}$ und $\vec{E}$ sind n-dimensional für n Patches in der Szene. Die Matrix M hat die Ausdehnung $N \times N$. Vektor $\vec{B}$ ist der Lösungvektor, den es zu bestimmen gilt.
In der Matrix M werden die Formfaktoren und ihre zugehörigen ρ-Werte gespeichert. Um die in der Matrix verwendeten n^2 Formfaktoren nicht alle berechnen zu müssen, wird sich der drei Eigenschaften der Formfaktoren bedient:

Eigenschaften der M-Matrix

1. keine Selbstbeleuchtung: $\forall\,(s)\, F_{ss} = 0$

2. Reziproke Eigenschaft: $F_{es} \cdot A_e = F_{se} \cdot A_s$
3. Geschlossene Umgebung: $\forall\,(s) \sum_{e=1}^{\#Flaechen} F_{se} = 1$

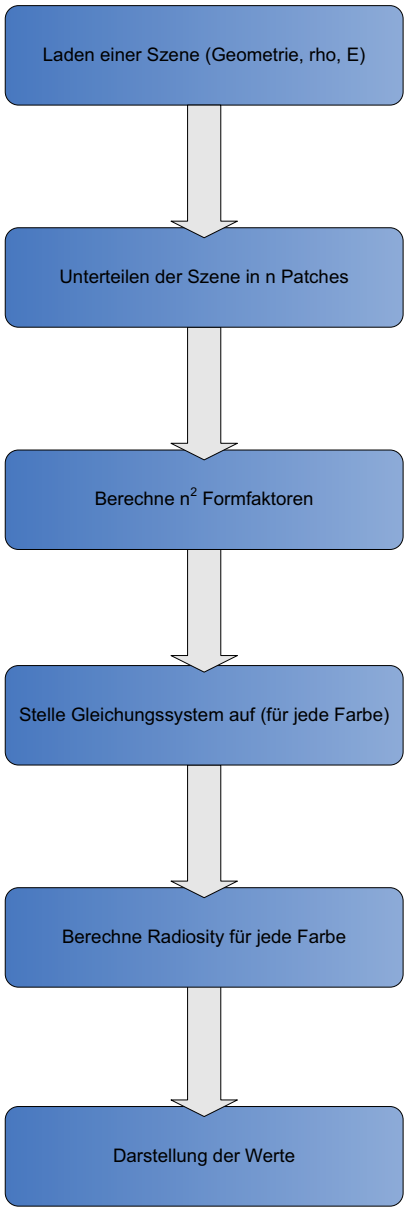
Somit ist die Mindestanzahl der zu berechnenden Formfaktoren:

$$\tfrac{1}{2}\,(N-1)\,(N-2)$$

Neben der Möglichkeit, die Berechnung der Formfaktoren zu beschleunigen, ist es wichtig, die Konvergenzkriterien für die M-Matrix zu erfüllen: Alle Werte, die nicht auf der Diagonalen liegen, sind **negativ**, alle übrigen Werte **positiv**. Die Matrix M muss des Weiteren diagonal-dominant sein:

$$|m_{ee}| > \sum_{s=1;s\neq e}^{n} |m_{es}|\,(1 \leq e \leq n) \tag{2.33}$$

2.3.3 Allgemeiner Radiosity Algorithmus



1. Alle Szenenelemente werden geladen und gleichzeitig zwei Vektoren, $\vec{\rho}$ und $\vec{E}$, aus den Patchinformationen der Szene erstellt. $\vec{\rho}$ beinhaltet die Reflexionskoeffizienten und $\vec{E}$ die Bestrahlungsstärke, bzw. die Initiale Radiosity.

2. Ausführen einer subdivide-Methode zur Unterteilung der Szenenelemente in kleine Patches, damit gewährleistet wird, dass ein genauerer Lichtausstausch auf kleineren Flächen stattfindet.

3. Berechnung der Formfaktoren zwischen allen Patches.

4. Die $\vec{B}$ und M Matrizen werden erstellt. $\vec{B}$ soll die aus dem Gleichungssystem errechneten Radiosities für die Patches enthalten. M beeinhaltet die Werte aus ρ und alle n^2 Formfaktoren. Zu beachten: Matrix M muss die Kriterien aus 2.3.2 erfüllen. Zu lösen gilt es: $M \cdot \vec{B} = \vec{E}$

5. Lösen des Gleichungssystems mit direkten oder indirekten Methoden.

6. Zuweisen der neuen Radiositywerte den Patches und Clippen der Werte auf den darstellbaren Bereich[a].

[a] Tonemapping

3 Radiosity-Algorithmen

3.1 Direkte & Indirekte Methoden

Damit das Gleichungssystem $M \cdot \vec{B} = \vec{E}$ bestimmt werden kann, wird sich den direkten oder indirekten Verfahren gewidmet. Direkte Methoden sind sehr geläufig zum Lösen von Gleichungssystemen, eignen sich aber wenig bei großen Gleichungssystemen. Methoden, wie z.B. die Matrix-Invertierung $\vec{B} = M^{-1} \cdot \vec{E}$ oder dem Gauss-Eleminationsverfahren, können zwar für das Radiosityproblem eingesetzt werden, sind aber wegen ihrer hohen Komplexität von $O\left(n^3\right)$ zu rechenaufwendig und eignen sich daher nur zur Berechnung von kleinen Szenen.
Indirekte Methoden, auch iterative Methoden genannt, nutzen gegenüber den direkten Methoden eine Schätzung, um zu einer Lösung zu kommen. Durch Anwendung möglichst kostengünstiger Rechenoperationen wird die Schätzung bei jeder neuen Iteration verbessert, bis sich eine Schätzung ergibt, die sich von dem zuvor berechneten Ergebnis kaum oder gar nicht mehr unterscheidet. Dieser Zustand wird als **konvergierter Zustand** bezeichnet. [CW93] Vorteile gegenüber den direkten Methoden ergeben sich in der geringeren Komplexität bei schwach besetzten Koeffizientenmatrizen, einer höheren Unempfindlichkeit gegen Rundungsfehler und benötigen weniger Speicherplatz.

Lösen der Radiosity-gleichung mit indirekten Methoden

Ausgehend von dem gegebenen Gleichungssystem:

$$M \cdot \vec{B} = \vec{E} \tag{3.1}$$

mit dem unbekannten Vektor $\vec{B}$, einem Startvektor $\vec{B}^{(0)}$ und einem Fehlervektor $\vec{e}^{(0)}$, der definiert ist als die Differenz des Lösungsvektors mit dem Startvektor, kann der Lösungsvektor B berechnet werden:

$$e^{(0)} = B - B^{(0)} \tag{3.2}$$

Sofern sich B nicht berechnen lässt, kann e nicht direkt bestimmt werden und ein Residuum-Vektor $r^{(0)}$ wird berechnet:

$$r^{(0)} = M \cdot B^{(0)} - E \tag{3.3}$$

Der Residuum-Vektor kann ohne Wissen über B bestimmt werden. Nach jedem Iterationsschritt werden $r^{(t)}$ und $B^{(t)}$ durch die neuen besseren Schätzungen $r^{(t+1)}$ und $B^{(t+1)}$ ersetzt. Wird in weiteren Iterationen der Residuum-Vektor r null, so ist auch der Fehler-Vektor e null und das Ergebnis ist richtig. Ausschlaggebend für eine schnelle Konvergenz des Gleichungssystems ist die Wahl des Initial-Vektors $B^{(0)}$. Je näher er an der Endlösung liegt, desto schneller konvergiert die Methode. $B^{(0)}$ kann mit verschiedenen Werten, Nullen oder Zufallszahlen, initialisiert werden, es hat lediglich Auswirkungen auf die Geschwindigkeit der Konvergenz. Bei der Radiosity-Methode wird als Initial-Vektor E gewählt, somit ist $B^{(0)} = E$.

3.2 Relaxationsmethoden

Die iterativen Methoden berechnen von einer Iteration zur nächsten einen Lösungsvektor, bei dem sich alle Komponenten ändern. Relaxationsmethoden verändern jeweils nur eine Komponente des Lösungsvektors, das bedeutet, dass jede Iteration ein Element des Residuum-Vektors gleich null gesetzt wird. Somit wird ein Wert $B_i^{(t)}$ so bestimmt, dass $r_i^{(t+1)} = 0$ und alle anderen Werte $r_j^{(t)}$ mit $j \neq i$ sich erhöhen und sich damit eine Verbesserung im Gesamten ergibt. Zwei solcher Relaxationsmethoden sind das **Jacobi**- und das **Gauss-Seidel**-Verfahren.

3.2.1 Jacobi-Verfahren

Das Jacobi-Verfahren wird Gesamtschrittverfahren genannt, da pro Iteration alle Komponenten des Lösungsvektors verändert werden. Somit ist das Jacobi-Verfahren ein iteratives Verfahren. Als Startvektor, wie oben beschrieben, wird der Emissionsvektor E benutzt. Anschließend wird der Residuum-Vektor $\vec{r}$ mit der Formel $\vec{r} = \vec{E} - M \cdot \vec{B}$ berechnet und auf den Lösungsvektor, den Radiosityvektor $\vec{B}$, addiert. Somit fliesst der neue Radiosityvektor $\vec{B}$ in der nächsten Iteration in die Berechnung mit ein. Mit erhöhter Anzahl an Iterationen sinken auch die sichtbaren Farbänderungen, die durch das Gleichungssystem berechnet werden, daher wird ein Abbruchkriterium eingeführt, wie z.B. einer maximalen Anzahl an Iterationen oder die Residuen werden auf $> \epsilon$ untersucht. Solang die Residuen noch oberhalb der Schwelle ϵ sind, wird die Berechnung fortgesetzt.

Die Fullmatrix-Methoden **Jacobi** und **Gauss-Seidel**

Algorithm 1 JacobiIteration()

```
1: B⃗ = E⃗;
2: while not converged do
3:     for all e do
4:         r⃗ = E⃗ − M · B⃗;
5:         B⃗ = B⃗ + r⃗
6:     end for
7: end while
```

Abbildung 3.1: Pseudocode des Jacobi-Algorithmus

3.2.2 Gauss-Seidel-Verfahren

Im Gegensatz zum Jacobi-Verfahren ist das Gauss-Seidel-Verfahren ein Einzelschritt-Verfahren[1]. Das Verfahren startet wie das Jacobi-Verfahren, indem der Lösungsvektor $\vec{B}$ mit dem Emissionsvektor $\vec{E}$ initialisiert wird. Das Verfahren arbeitet analog zum Jacobi-Verfahren, aktualisiert die berechneten Radiositywerte aber direkt innerhalb der Iteration. Durch diesen Vorteil konvergiert das Verfahren schneller als das Jacobi-Verfahren.

[1] Relexationsverfahren

Algorithm 2 GaussSeidelIteration()

1: $\vec{B} = \vec{E}$;
2: **while** not converged **do**
3: **for** all e **do**
4: $B_e = E_e - \sum_{s \neq e}^{n} m_{es} \cdot B_s$;
5: **end for**
6: **end while**

Abbildung 3.2: Pseudocode des Gauss-Seidel-Algorithmus

3.3 Progressive Refinement

Die Algorithmen Jacobi und Gauss-Seidel werden Fullmatrix-Methoden genannt, da ihr Kernalgorithmus die komplette M Matrix erstellt, um das Radiositygleichungssystem zu lösen. Dieser Umstand bringt aber zwei große Probleme mit sich: Zum einen ist der Speicherverbrauch für diese Matrix enorm, wenn die Szene sehr groß ist, und zweitens ist eine erhebliche Rechenleistung erforderlich, damit das System schnell konvergiert. Progressive Refinement[2] ist eine Lösung für das Problem. Während bei den Fullmatrix-Methoden Gather-Algorithmen[3] eingesetzt werden, diese Algorithmen sammeln die Radiosity von allen anderen Patches ein, ist Progressive Refinement die umgekehrte Methode und verteilt die Radiosity eines Patches an alle anderen. Das Ziel von Progressive Refinement

Grundlagen des **Progressive Refinement** Algorithmus

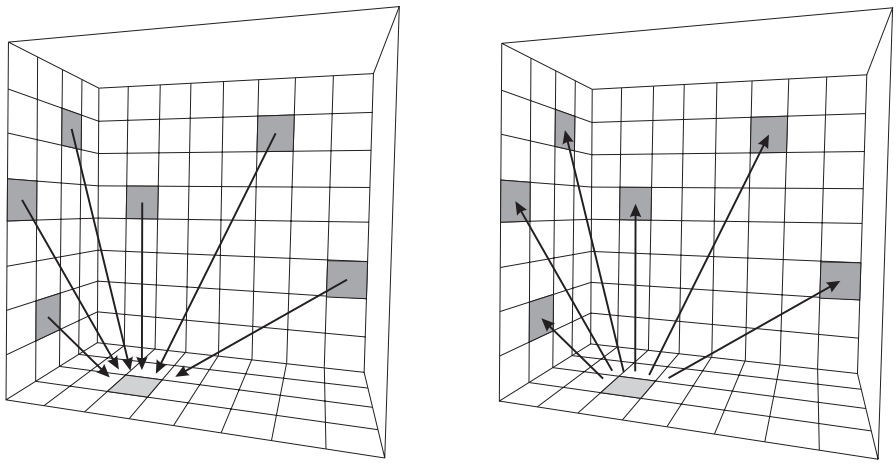


Abbildung 3.3: Skizze Gather- und Shootalgorithmus

liegt in der schnelleren Konvergenz und dem direkten Darstellen der Resultate nach einer Iteration, im Gegensatz zu den FM-Methoden. Daher versucht die Methode zu Beginn des Algorithmus in der ersten Iteration den größten Fortschritt aller Durchgänge zu machen, der möglich ist.

[2] Schrittweise Verfeinerung
[3] to gather = einsammeln

Physikalisch interpretieren lässt sich die PR-Methode wie folgt: Jedes Patch i besitzt eine Radiosity B_i, die zuvor für das Patch berechnet wurde und eine unshot-Radiosity ΔB_i, die angibt, wie viel Radiosity von dem Patch noch nicht versendet wurde. Zu Beginn des Algorithmus werden die beiden Werte auf die Emissionswerte der Patches gesetzt. Im nächsten Schritt wird das Patch mit dem größten unversendeten Lichtstrom, $\Delta B_i \cdot A_i$, gesucht und seine Energie Δrad an alle anderen Patches j, mit Bedingung $i \neq j$, verteilt. Die Energie Δrad wird bei dem Empfängerpatch sowohl auf B_j als auch auf ΔB_j aufaddiert. Nachdem die Energie an alle anderen Patches abgegeben wurde, wird nun die unshot-Radiosity Δrad des Patches i auf null gesetzt, $\Delta B_j = 0$. Der Pseudocode Algorithmus 3 verdeutlicht den Ablauf des Algorithmus.

Algorithm 3 ProgressiveRefinement()

```
 1: for alle i do
 2:     B_i = E_i;
 3:     ΔB_i = E_i;
 4: end for
 5: while nicht konvergiert do
 6:     nimm i mit ΔB_i · A_i = max;
 7:     for jedes Element j do
 8:         Δrad = ΔB_i · ρ_j F_ji;
 9:         ΔB_j = ΔB_j + Δrad;
10:         B_j = B_j + Δrad;
11:     end for
12:     ΔB_i = 0
13:     Darstellung der Ergebnisse
14: end while
```

3.4 Hierarchisches Radiosity

Grundlagen des **Hierarchischen Radiosity** Algorithmus

Damit computergenerierte Szenen photorealistischer wirken, wird die normale Patchaufteilung in einer Szene verfeinert. In Bereichen, in denen eine höhere Patchauflösung erforderlich ist, z.B. in Schattenregionen, unterteilt man die Patches in mehrere Unterpatches. Mit einer solchen Unterteilung steigt auch der Berechnungsaufwand von Radiosity in der Szene, daher kann das Patch nicht nur verfeinert werden, sondern es können auch mehrere Patches zu einem Patch zusammengefasst werden. Man führt eine Hierarchie in die Patch-Datenstruktur ein. Mit der einfachen Unterteilung der Patches in einer Szene steigt der Aufwand der Berechnungen an. Eine Hierarchie hat nun den Vorteil, dass man die Bereiche, die unterteilt werden sollen, bestimmen und verfeinern kann, während die anderen Patches unverändert bleiben. Daher entsteht ein erkennbarer Geschwindigkeitsvorteil der Hierarchiestruktur mit gleichzeitiger Möglichkeit, Teile der Szene zu unterteilen.

Warum Hierarchisches Radiosity?

Die zuvor vorgestellte Full-Matrix Methode ist zu speicheraufwendig und die Progressive-Refinement Methode benötigt zu viele Relaxationen, um indirektes Licht bis zur Konvergenz zu berechnen. Des Weiteren macht es keinen Sinn, von einem Sender oder

Empfänger aus alle Patches der „Welt“ zu betrachten. Diesen Problemen und den Herausforderungen stellt sich die Hierarchische Radiosity Methode.[Mül01b]
1986 schlug Cohen et al [CGI86] in den ersten hierarchischen Lösungsansätzen eine 2-Level-Hierarchie vor. Die Szene wird in Patches unterteilt, welche wiederum in kleine Elemente aufgeteilt werden können. Oberflächen, die als Lichtquelle agieren, werden grob in Patches unterteilt, während die beleuchteten Oberflächen in eine feinere Struktur, in Elemente, unterteilt werden. Die Abbildung 3.4 vedeutlicht den Vorgang. In diesem Ka-

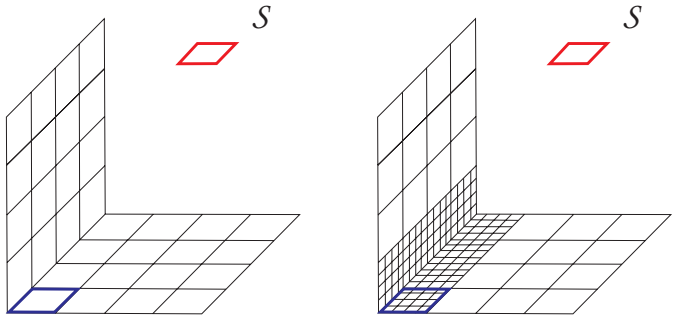


Abbildung 3.4: Skizze zur 2-Level Hierarchie[Mül01b]

pitel wird von einer Multilevel-Hierarchie ausgegangen, die später im Jahre 1990/1991 von Hanrahan et al. beschrieben wurde[HSA91]. Das erlaubt Sendern und Empfängern, auf verschiedenen Detailstufen zu operieren. In einer Multilevel-Hierarchie kann deshalb Energie zwischen den Patches auf jeder beliebigen hierarchischen Ebene ausgetauscht werden. Diese Vorgehensweise führt dazu, dass weniger Gather-Operationen durchgeführt werden müssen. Der Aufwand, gemessen an der Anzahl der Formfaktorberechnungen, sinkt auf $O(n)$ bei einer vorgegeben Fehlertoleranz. n bezeichnet die Anzahl der Blätter im Quadtree, in welchem die Hierarchie abgespeichert wird.

Multilevel-Hierarchie

Pat Hanrahan bezog das Radiosity Problem auf ein ähnliches Problem, dem N-Body-Problem. Dieses beschäftigt sich mit der Interaktion der Gravitation zwischen n Partikeln [HSA91], die mit ihrer Energie die anderen $n-1$ Partikel beeinflussen, das bedeutet, es entstehen n^2 Interaktionen. Implementationen zum Lösen dieses Problems zeigten auf, dass die Energie, die von einem Partikel auf eine entfernte Gruppe mehrerer Partikel wirkt, in einer einzelnen Interaktion berechnet werden kann. Ein Beispiel: Ein System mit drei Galaxien mit großer Anzahl von Partikeln, die weit entfernt voneinander sind, können aufgrund der hohen Distanz als jeweils ein Objekt angesehen und entsprechend die Interaktion zwischen drei Objekten berechnet werden. Das Radiosity-Problem ist ähnlich dem des N-Body-Problems, in dem die Anzahl der Interaktionen mit $\frac{1}{r^2}$ abnehmen. Die Interaktionen können mithilfe der Formfaktor-Algebra zusammengefasst werden. Der größte Unterschied dennoch ist, dass Radiosity mit großen Flächen, die in kleinere unterteilt werden, die Berechnung beginnt. Wobei hingegen das Partikel-Problem direkt auf kleinen Objekten, die zusammengefasst werden, arbeitet. Die Anwendung des N-Body-Problems auf Radiosity zeigt die Abbildung 3.5. Der Hierarchische Radiosity Algorithmus erstellt keine explizite Matrix M, so wie die vorher beschriebenen Algorithmen. Stattdessen werden, während die Unterteilung der Patches rekursiv erfolgt, Links

N-Body-Problem

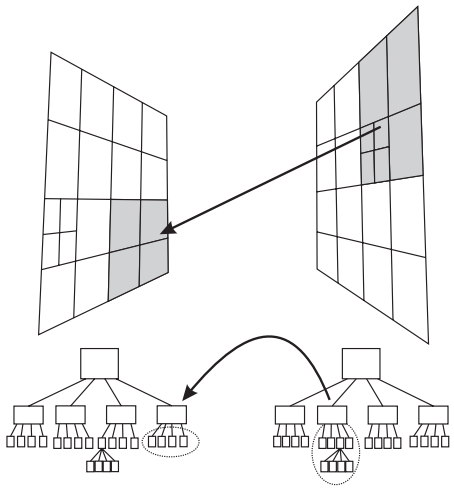


Abbildung 3.5: Radiosity mit Multilevel-Hierarchie, Datenstruktur Quadtree[Mül01b]

zwischen den Knoten des Quadtrees und jeder Oberfläche erstellt. Ein Link repräsentiert eine Beziehung zwischen zwei Mengen von Elementen. Die Elemente der beiden Mengen sind die Blätter der Unterbäume unterhalb des Knotens, an welchem der Link endet. Jeder Link repräsentiert somit eine Untermenge von Formfaktoren aus der originalen $n \times n$ Matrix M. Ein Link zwischen zwei Knoten auf der niedrigsten Stufe des Baumes entspricht einem einzigen Eintrag in der M Matrix. Links auf höheren Ebenen repräsentieren daher eine Gruppe von mehreren Formfaktoren.
Zur Erstellung der Links kann zunächst die komplette $n \times n$ Matrix mit n^2 Formfaktoren aufgebaut und nach Gruppen gesucht werden, in denen die gleichartigen Formfaktoren zusammengefasst werden. Im Endeffekt würde diese Methode die Zeit verkürzen, in der das Gleichungssystem gelöst würde, aber dem Hauptproblem, der Reduzierung der Berechnung von Formfaktoren, entgegen sprechen. Daher wird eine Methode genutzt, die vorhersagt, ob eine Gruppe von Formfaktoren vom Betrag her gleich ist, auch kohärent genannt, bevor die eigentliche Berechnung der Formfaktoren durchgeführt wird. Wird eine Gruppe als kohärent bestimmt, so wird für sie ein einziger Formfaktor berechnet, der diese Gruppe repräsentiert. Diese Methode zur Vorhersage der Kohärenz in einer Gruppe nennt sich Orakel-Funktion. Auf diese Funktion wird später näher eingegangen. Der grundlegende Ablauf des hierarchischen Radiosity Algorithmus geschieht wie im Algorithmus 4 beschrieben.

Allgemeiner HR-Algorithmus

Algorithm 4 HierarchicalRadiosity()

```
1: InitialLinking
2: while nicht konvergiert do
3:     Refine
4:     Gather
5:     PushPull
6: end while
```

Zunächst werden Links zwischen den einzelnen Root-Patches der Szene erstellt, dieser Schritt ist das *Initial Linking*. Die Links werden in einer eigenen Liste gespeichert, die für jeden Root-Patch angelegt wird. Diese Links werden *Gather*-Links genannt, da später anhand der Links die Radiosity eingesammelt wird. Die nachfolgenden Schritte werden solange ausgeführt, bis die Lösung des Gleichungssystems konvergiert ist. Die *Refine*-Methode prüft alle Links, entscheidet, ob dieser verfeinert werden muss und führt dieses gegebenenfalls durch. Nachdem die Links verfeinert wurden, wird nun die Liste der *Gather*-Links durchlaufen und die Radiosity für das Patch eingesammelt. Damit die Radiosity innerhalb der Hierarchie konsistent bleibt, muss noch der *PushPull*-Schritt durchgeführt werden.

3.4.1 Initial Linking

Die Szene besteht zu Beginn des Algorithmus aus Root-Patches, die zunächst alle miteinander mithilfe eines Links verbunden werden. Die Skizze 3.6 zeigt ein Beispiel für die Verlinkung von drei Patches. Dabei gibt es keine Eigenverlinkung. Zur Erstellung

Initial Linking als Preprocess

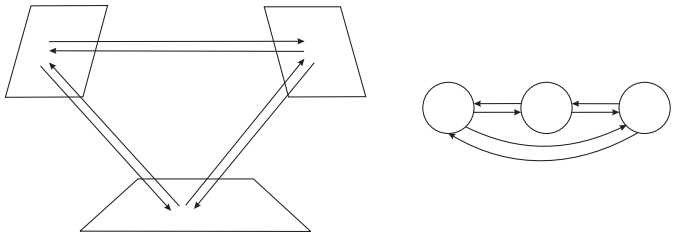


Abbildung 3.6: Schritt 1 HR: Initiales Verlinken[Mül01b]

der Links sind zwei Datenstrukturen notwendig, die grundlegend für den hierarchischen Radiosity Algorithmus sind. Die Datenstruktur *Quadnode* speichert die Informationen eines Knotens im Quadtree, die Struktur *Link* beinhaltet die Informationen eines Links zwischen zwei Knoten im Quadtree:

```
struct Quadnode{
  float              Bg;       // gather radiosity
  float              Bs;       // shooting radiosity
  float              E;        // emission
  float              area;     // Fläche des Patches
  float              ρ;        // Reflexionsgrad
  struct Quadnode**  children; // Liste mit vier möglichen Kindern
  struct Link*       L;        // Pointer auf ersten Gather-Link
};

struct Link{
  struct Quadnode*   q;        // gather node
  struct Quadnode*   p;        // shooting node
  float              Fqp;      // Formfaktor zwischen q und p
  float              vis;      // Visibilitästerm zwischen q und p
  struct Link*       next;     // nächster Gather Link von q
};
```

3.4.2 Refine

Die Refine-Methode untersucht alle Links und überprüft, ob diese verfeinert werden müssen. Die Vorgehensweise der Methode beschreibt der Algorithmus 5 im Anhang. Die Methode benötigt vier Hilfsmethoden. Die erste Methode *bool OracleRefine(e, s, vis, FF)* testet, ob ein Link anhand seiner Parameter verfeinert werden sollte. Nachdem bestimmt wurde, dass der Link verfeinert wird, entscheidet die Methode *OracleSubdivide(e, s)*, ob der Empfänger oder der Sender verfeinert werden muss. Des Weiteren stehen die Methoden *deleteLink(L)* und *Subdivide()* zur Verfügung. Erstere löscht den angegebenen Link, zweitere führt die eigentliche Verfeinerung des Patches durch.

Refine - Unterteilung von Patches

Zuerst entscheidet *OracleRefine()*, ob der Link verfeinert wird. Falls nicht, wird der nächste Link getestet, falls aber doch entscheidet *OracleSubdivide()*, ob der Empfänger oder Sender verfeinert wird. Anschließend wird der Refine-Methode übergebene Link mit *deleteLink()* gelöscht und der entsprechende Patch unterteilt. Nachdem die Subdivide-Methode das Patch verfeinert hat, ist es wichtig neue Links zwischen den Patches zu bilden. Die Abbildung 3.7 zeigt die Durchführung: Zuerst Löschen des direkten Links zwischen den Patches mit anschließender Unterteilung und Setzen von vier neuen Links zwischen den Kindern und des nicht-verfeinerten Patches.

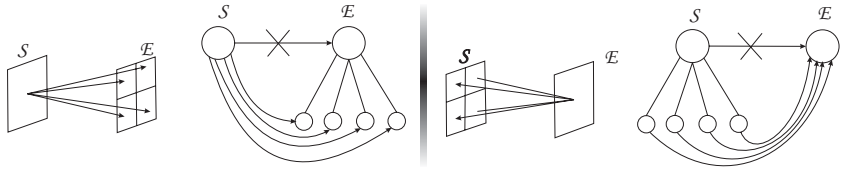


Abbildung 3.7: Schritt 2: Refine[Mül01b]

3.4.3 Gather

In einer Schleife für alle Gather-Links eines Patches wird die Radiosity Energie B_s eingesammelt und auf die Gather Energie B_g aufaddiert: $B_e+ = B_s \cdot \rho_e \cdot F_{es} \cdot vis$. Der Algorithmus 8 und die Abbildung 3.8 zeigen diesen Vorgang.

Gather - Einsammeln von Radiosity anhand von Gatherlinks

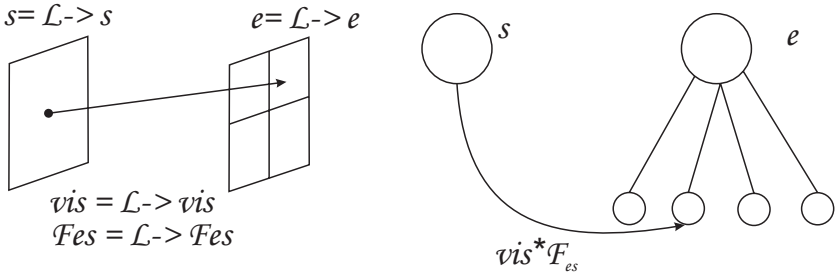


Abbildung 3.8: Schritt 3: Gather[Mül01b]

3.4.4 PushPull

Nach dem Einsammeln der Radiosity über verschiedene Ebene der Hierarchie hinweg, ist es nötig, die Radiosity innerhalb der Hierarchiestruktur konsistent zu halten. Zum einen muss für die Darstellung die Radiosity der Blätter und zum anderen die Radiosity innerhalb der Knoten des Quadtrees konstistent sein, damit der Radiosityaustausch stimmt. Deswegen wird die empfangene Radiosity B_g aus dem Gather-Schritt an die Kindknoten weitergegeben: **Push**: $B_i+ = B$ und anschließend der flächengewichtete Mittelwert aller Kinderradiositywerte zurückgeholt: **Pull**: $B = \frac{1}{A}\sum B_i \cdot A_i$, veranschaulicht in der Abbildung 3.9. Die nachfolgende Abbildung 3.10 zeigt eine hierarchische Struktur zum

PushPull - Konsistenz der Daten innerhalb der Hierarchie

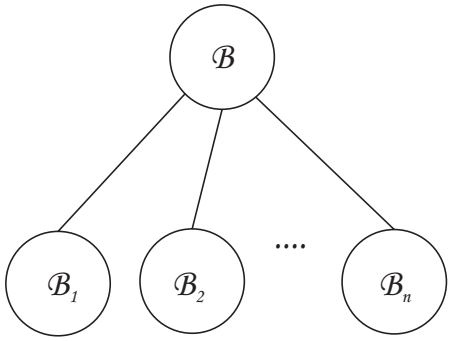


Abbildung 3.9: Schritt 4: PushPull[Mül01b]

Austausch von Radiosity in einer Szene. Die Linien zwischen den Patches zeigen die auf verschiedenen Ebene der Hierarchie angelegten Links. Die unterschiedlichen Farben der Linien sagen etwas über die Sichtbarkeit eines Links zwischen zwei Patches aus (dunkel blau: vollständig verdeckt, weiß: vollständig sichtbar, grün: teilweise verdeckt).

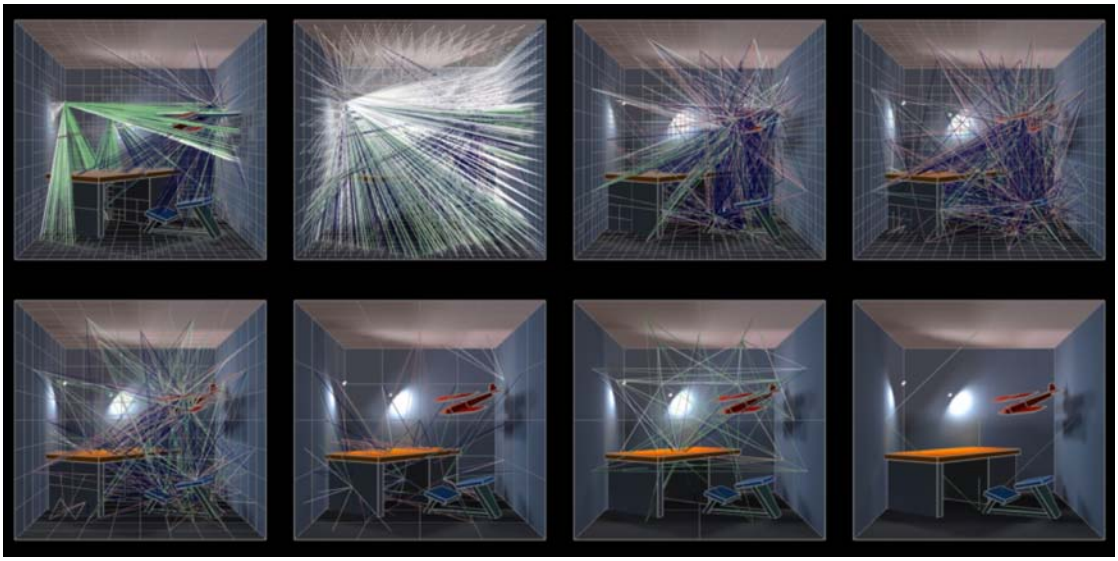


Abbildung 3.10: Darstellung der HR-Links auf verschiedenen Hierarchieleveln[Mül01b]

4 GPU Programmierung

Das Kapitel GPU Programmierung beschäftigt sich mit der Vorstellung von Sprachen zum Programmieren von Grafikkarten. Speziell wird hier die Sprache Cg von NVidia angesprochen, da die Umsetzung der Diplomarbeit auf der Verwendung von Cg beruht. Die sogenannten Shadersprachen werden in ihrer Grundfunktion erläutert und mit Beispielen dargestellt. Im Weiteren wird dazu übergegangen, zu beschreiben, wie es möglich ist, die Shadersprachen, ihrem Zweck widersprechend, für allgemeine Berechnungsaufgaben zu nutzen.

Shadersprachen

Shadersprachen sind für den prinzipiellen Einsatz im Bereich Computergrafik ausgelegt. Sie erledigen spezielle Aufgaben, wie z.B. das Darstellen von Objekten, das Kolorieren sowie die Bewegung von Objekten, um nur ein paar Einsatzgebiete zu nennen. Die wohl bekanntesten und aktuellsten Shadersprachen sind:

- OGLSL - OpenGL Shading Language
- HLSL - Highlevel Shading Language (von Microsoft)
- Cg - C For Graphics (NVidia & Microsoft)

Im Zuge der Vorbereitungen auf die Diplomarbeit schien der Einsatz der Cg - Shading Language am geeignesten. Cg lässt sich mithilfe von OpenGL einfach ansprechen und einbinden. Des weiteren unterstützt Cg direkt die Fähigkeiten der NVidia Geforce 6 Grafikkarte, die als Grundlage der Programmierungen diente. Der Cg-Kompiler ist für den Einsatz mit dem NV40 Chipsatz optimiert.

4.1 NVidias Cg Shading Language

Was ist **Cg**?

Cg als Shadingsprache ermöglicht mithilfe der Grafikkarte die Struktur, das Erscheinungsbild und die Bewegung von Objekten zu kontrollieren und beschleunigt darzustellen. Es ist die Schnittstelle, die die Einstellung der Parameter für die zuvor beschriebenen Aktionen mit der enormen Geschwindigkeit und dem Leistungsumfang der modernen Grafikkarten vereint. Die ersten Shadingsprachen waren ähnlich der Assembler-Programmiersprache und wurden hardwarenah programmiert, was wenig Komfort in der Programmierung zur Folge hatte. Cg erleichtert das Programmieren, da die Programmierer nun nicht mehr ausschließlich hardwarenah programmieren müssen. Als Kooperationsarbeit zwischen Microsoft und NVidia wurde Cg für OpenGL und DirectX auf verschiedenen Plattformen entworfen. Es ist somit im Gegensatz zu Microsofts Eigenproduktion HLSL plattformunabhängig. Shadingsprachen im Allgemeinen unterscheiden sich gegenüber von normalen Programmiersprachen wie C++ und Java darin, dass sie

für den speziellen Einsatz im Computergrafikbereich konzipiert sind. Dennoch sind mit dem eingeschränkten Funktionsumfang der Shadersprachen in Cg auch komplexe physikalische Simulationen berechenbar, so genannte NON-Shading Operationen. Der wohl größte Unterschied zu gewöhnlichen Programmiersprachen ist das Datenflussmodell der Shadingsprachen. Bei diesem Modell existiert eine feste Reihenfolge von Verarbeitungsschritten, die auf die Eingabedaten angewendet wird. Dieses Modell verspricht im höchstens Maße Parallelität und daher enorme Geschwindigkeitsvorteile. Eingabedaten für Cg-Programme können zum einen Vertex-Daten und zum anderen Fragment-Daten sein. Unter Vertex-Daten werden z.B. x,y,z und w Koordinaten eines Punktes, verstanden. Fragment-Daten sind ähnlich den Pixeldaten, besitzen aber mehr Eigenschaften, z.B. Überblendungseigenschaften oder Nebel.[FK03] Wird ein Vertex erzeugt, so durchläuft

Grundlagen der GPU-Architektur

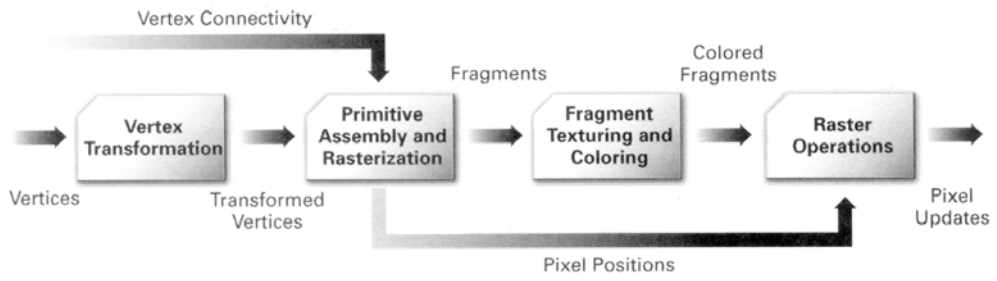


Abbildung 4.1: Skizze der Grafikpipeline mit Vertex und Fragmentdaten[FK03]

es automatisch das geladene Cg-Programm. Für die beiden Datentypen, Vertices und Fragmente, gibt es jeweils zwei Arten der Verarbeitung: den Vertex-Shader sowie den Fragment-Shader. Die Abbildung 4.1 zeigt den Datenaustausch und die typische Verarbeitung in einer Grafikpipeline.
CPUs sind für den „general purpose“-Einsatz gedacht und somit universell einsetzbar. Grafikprozessoren sind auf ihr Einsatzgebietes optimiert und in ihrem Bereich äußerst schnell. Daher wird versucht, die Geschwindigkeit der GPUs für „general purpose“-Anwendungen zu nutzen. Dies widerspricht aber der Architekur der GPUs und es werden spezielle Methoden erforderlich, um die GPUs so nutzen zu können. Das Datenflussmodell der GPUs ist ein Streamingmodell. Zu Beginn einer Verarbeitungspipeline mit mehreren Prozessen werden Daten, die dann sequentiell diese durchlaufen und fertig verarbeitet am Ende der Pipeline herauskommen, eingegeben. Das Problem des Streamings ist, dass die Daten während der Verarbeitung keine Abhängigkeiten haben. Somit sind z.B. keine Pixelnachbarinformationen verfügbar und diese müssen, falls benötigt, speziell zur Verfügung gestellt werden. Die Parallelisierung der Prozesse in der Pipeline macht die GPU so effektiv und geschieht auf zwei Arten. Schon beim Zugriff auf Dateneinheiten, z.B. rgba-Wert, im Shader werden diese parallel in einem Schritt erfasst. Diese Eigenschaft nennt sich SIMD[1]. Die zweite Parallelisierung erfolgt durch die Anzahl der verwendeten Pipelines. In momentanen Grafikprozessoren sind es 16. Shader übernehmen innerhalb der Pipeline nur Teilaufgaben, wie in Abbildung 4.2 dargestellt. Zur

Eigenschaften moderner Grafikkarten

[1]Single Instruction Multiple Data

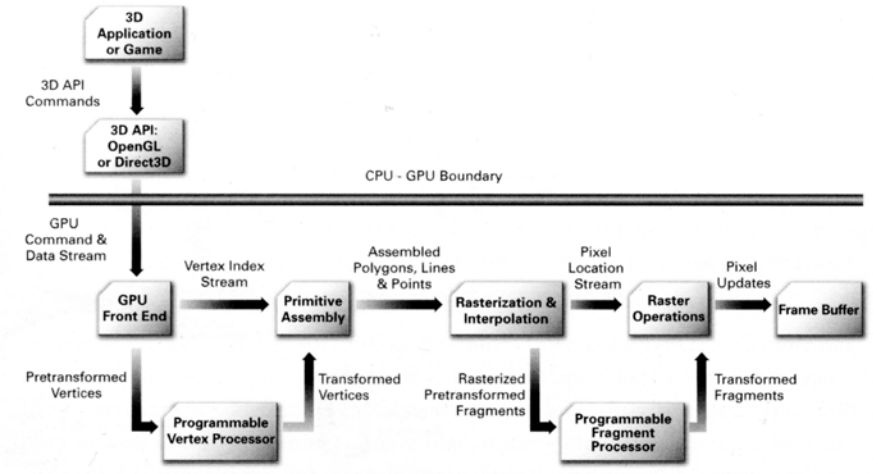


Abbildung 4.2: Skizze einer programmierbaren Pixelpipeline[FK03]

Umsetzung von Radiosity auf der Grafikhardware werden Fragmentshader benötigt, da der Leistungsumfang von Vertexshadern nicht ausreicht, um die komplexen Berechnungen durchzuführen. Das folgenden Kapitel beschreibt die Voraussetzung zur Nutzung der GPU für allgemeine Berechnungen.

4.2 CPU zu GPU

Die Programmierung auf der GPU erfordert einiges Umdenken von der CPU - Programmierung. Es gibt keine vorgegebenen oder selbst zu definierenden Datenstrukturen. Um Berechnungen auf der Grafikkarte durchführen zu können reicht es nicht, eine Datenstruktur an den Shader als Textur zu übergeben und dem Shader die Berechnung zu überlassen. Zunächst muss beachtet werden, dass die Grafikkarte nur auf Vertices oder Fragmenten arbeitet. Wie werden die Daten in Fragmente übergeben, wenn Fragmentshader genutzt werden? Dazu werden zunächst die Begriffe Textur und Fragment Programm in diesem Kontext erläutert.

GPU für allgemeine Zwecke nutzen (**general purpose**)

4.2.1 Textur als Array

Grafikprozessoren kennen lediglich Texturen. Texturen dienen als einziger Datenspeicher. Mithilfe der uniform-Funktion der Cg-Shader lassen sich auch Daten direkt an den Shader übergeben. Mit speziellen Texturerweiterungen, zum Beispiel GL_TEXTURE_ RECTANGLE_NV, lassen sich auch float Werte in einer Textur speichern. Das Texturformat überstützt zu den standardmäßigen OpenGL-Texturformaten, wie GL_TEXTURE_2D, auch Größen, die nicht $2^n x 2^n$ entsprechen. Von Vorteil ist das Speichern der Berechnungsdaten in der STL-Datenstruktur *vector* und das erste Objekt des *vectors* als Adresse auf die Daten im Arbeitsspeicher zu nutzen:

Textur als Datenspeicher

```
vector<float> array;
```

```
glTexImage2D(GL_TEXTURE_RECTANGLE_NV, 0, GL_FLOAT_RGBA32_NV,
                     texSize, texSize, 0, GL_RGBA, GL_FLOAT, &array[0])
```

4.2.2 Fragment Programm als Computational Kernel

Nachdem die Eingabe durch eine Textur geregelt ist, wird auch der Algorithmus benötigt, der auf den Daten arbeiten soll. Der Programmkern, Computational Kernel, ist ein Fragment-Shader. Der Shader stellt den Verarbeitungskern dar, der auf der CPU durch einen Algorithmus präsentiert wird. Fragment Shader warten mit einer großen Bibliothek an mathematischen Funktionen auf und können auf Texturen zugreifen. Ihre Berechnungen beschränken sich im Wesentlichen auf SIMD-Werte, z.B. einen xyzw- oder rgba-Wert. Natürlich kann anstelle eines rgba-Wertes auch jeder andere Wert eingegeben und ausgegeben werden. Die Ausgabe ist nicht auf den Wertbereich 0 bis 255 beschränkt.

Fragmentshader als Berechnungsalgorithmus

4.2.3 Eins-zu-Eins Pixel zu Texel Mapping

Ein Fragment Programm wird, nachdem es definiert wurde, auf alle vorhandenen Fragmente oder Texel angewendet. Dazu muss gewährleistet werden, dass die richtigen Daten in der Textur auch in diesem Fragment zur Verfügung stehen. Das bedeutet, die Pixeldaten müssen den Texturdaten entsprechen. Des Weiteren müssen alle Daten der Textur bearbeitet werden.

1. Orthographische Projektion - Die orthographische Projektion erlaubt keine Verzerrungen bei der Darstellung von Daten:

   ```
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(0, 1, 0, 1);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   ```

2. Data-Dimensioned Viewport - Der Viewport gibt die Anzahl der Pixel an, die gerendert, bzw. bearbeitet werden sollen und wird in der Größe der Textur gewählt.

   ```
   glViewport(0, 0, texSize, texSize);
   ```

4.2.4 Viewport-Sized Quad als Data Stream Generator

Die Berechnung der Daten findet, wie schon gesagt, mithilfe der Fragmente statt. Damit die Fragmente nach der Rasterisierung verfügbar sind, muss zunächst Geometrie gezeichnet werden. Diese wird in Form eines Quads der Größe des Viewports gezeichnet[2].

[2] auch screenfilling quad/polygon

```
glBegin(GL_QUADS);
  glVertex2f(0.0, 0.0);
  glVertex2f(1.0, 0.0);
  glVertex2f(1.0, 1.0);
  glVertex2f(0.0, 1.0);
glEnd();
```

4.2.5 Copy-To-Texture als Feedback

Nachdem die Berechnung auf der GPU stattgefunden hat, können die Ergebnisse ausgelesen werden, um sie in der nächsten Iteration nutzen zu können oder um sie verarbeitet darzustellen. Die Daten werden aus dem Framebuffer ausgelesen und in einer Textur gespeichert. Dieser Vorgang nennt sich Copy-To-Texture. Damit werden die Daten aus dem Framebuffer wieder zugänglich und können auch auf der CPU weiterverarbeitet werden:

```
glBindTexture(GL_TEXTURE_RECTANGLE_NV, Texture);
glCopyTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, 0, 0, 0, 0, 0,
                                                        Width, Height);
```

Der Nachteil der Copy-To-Texture - Methode ist, dass sie langsam ist, da die Daten erst in den Framebuffer geschrieben werden und anschließend wieder ausgelesen werden. Abhilfe schafft die nachfolgende Methode.

4.3 RenderTexture

Mark Harris stellt die Klasse RenderTexture zur Verfügung, die die Render-To-Texture Methode, sowie die Copy-To-Texture Methode unterstützt. Diese Methode kann verschiedenste Daten, int, float für unterschiedliche Texturen, z.B. RGBA32, verarbeiten. Die Verwendung der Klasse wird der Headerdatei der Renderklasse entnommen. Sie ist unter dem Link [3] verfügbar.
Die Renderklasse unterstützt das Rendern von berechneten Daten direkt in eine Textur anstatt in den Framebuffer, welches höhere Performanz verspricht. Wahlweise können die Daten aus der Renderklasse auch mit:

```
float data[512*512*4];

glReadPixels(0, 0, texSize, texSize, GL_RGBA, GL_FLOAT, data);
```

in ein Array geschrieben werden.

[3] http://www.markmark.net/misc/rendertexture.html

5 Radiosity auf der GPU

Das Kapitel Radiosity auf der GPU stellt die beiden Verfahren Full-Matrix und Progressive Refinement aus den Papern [CHH03] und [CjHL04] vor und die Umsetzung der Algorithmen auf der GPU wird erläutert und diskutiert. Der Algorithmus Full-Matrix wurde selbst implementiert und wird zusätzlich zu dem Paper präsentiert. Der Algorithmus ist aber nicht allgemein gehalten (wie der Hierarchische Algorithmus auf der GPU) und dient nur zu Demonstrationszwecken, wie die Umsetzung aussehen könnte.

5.1 Full-Matrix-Methode

Die Full-Matrix-Methode ist die Grundlage der ersten Radiosityberechnungen für computergenerierte Szenen. Die Umsetzung dieser Methode bringt einige Probleme mit sich. Das größte Problem ist der Umfang der zu berechnenden Daten. Es müssen n^2 Formfaktoren, bei n Flächenelementen, berechnet werden, was verdeutlicht, dass die Formfaktorberechnung den Flaschenhals in der Methode darstellt. Der Aufwand ist nicht das Problem, doch aber die Limitierung durch den Speicher auf der Grafikkarte. Aus Performanz und Speicherplatzgründen wird diese Technik aber nur testweise implementiert und mit der Geschwindigkeit der CPU-Version verglichen[CHH03].
Mithilfe der Nutzbarkeit von Floating-Point-Texturen ist es möglich, Matrix Operationen wie die der FM-Methode, auf der GPU durchzuführen. Die Matrix-Radiosity spiegelt die globale Beleuchtung in einer geschlossenen Umgebung mit einem linearen Gleichungssystem wieder:

$$B_i = E_i + \rho_i \sum_{j=1}^{N} F_{ij} B_j \tag{5.1}$$

Die Zusammensetzung der Formel wird in Kapitel 3 beschrieben. Durch Umformungen wird im Endeffekt eine Lösung für die Gleichung:

Full-Matrix auf der GPU

$$M \cdot B = E \tag{5.2}$$

gesucht, wobei B den zu lösenden Radiosityvektor angibt. Carr et al. entschieden sich für die Umsetzung der Jacobi-Methode, obwohl die Gauss-Seidel-Methode, da sie weniger Speicher benötigt und schneller konvergiert, geeigneter erscheint. Der Vorteil der Jacobi-Methode liegt in der sichtbaren Parallelität des Algorithmus, der eher dem Streaming-Modell der GPU entspricht. Aus der Jacobi-Formel:

$$B_i^{(k+1)} = E_i - \sum_{j \neq i} M_{ij} \frac{B_j^{(k)}}{m_{ii}} \tag{5.3}$$

wird durch Umformung und Vermeidung der $j \neq i$-Bedingung, da sie nicht in Fragment-Programmen umgesetzt werden kann, die folgende Gleichung:

$$B^{(k+1)} = B^{(k)} + E - diag\,(M)^{-1}\,MB^{(k)} \tag{5.4}$$

Als Datenspeicher wird eine RGB-Matrix verwendet, sodass mit einem Texturzugriff im Fragmentshader alle drei Werte mit einem Aufruf geladen werden können. Die eigentliche Berechnung arbeitet jeweils auf einem Teil des Matrix-Vektor Produktes. Die Anzahl der pro Durchgang berechneten Werte hängt ab von der maximalen Anzahl an Shaderinstruktionen, die während eines Durchgangs möglich sind. Die Implementation auf der Geforce FX sieht die Berechnung von 254 Elementen pro Durchgang vor, da 1024 Instruktionen pro Durchgang ausgeführt werden und die Berechnung vier Instruktionen für sich beansprucht. Daraus folgt, dass eine Jacobi Iteration bei einer N-Patch Szene $N/254$ Durchgänge benötigt. Die Abbildung 5.1 zeigt das Resultat der Berechnung. Die Formfaktorberechnung wurde zuvor auf der CPU ausgeführt, da diese bei der Szene nicht so stark ins Gewicht fällt. Die berechneten Radiositywerte werden in einer 1D-Textur

Full-Matrix-Umsetzung auf die GPU nach Carr[CHH03]

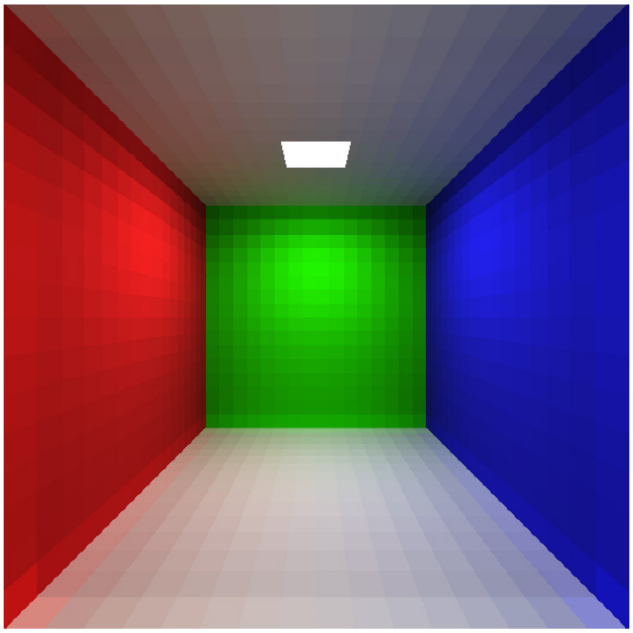


Abbildung 5.1: Radiositylösung der Cornellbox-Szene[CHH03]

gespeichert und mithilfe von Texturkoordinaten auf die Patches gemapt. 1D-Texturen verhindern die Möglichkeit, die Daten linear zu interpolieren. Geschwindigkeitstests des Algorithmus zeigten, dass der Algorithmus auf der GPU dennoch langsamer ist als die Gauss-Seidel-Implementierung auf der CPU. Weitere Informationen hierzu sind unter [CHH03] zu finden.

Eigene FM-Implementation

Die Einarbeitung in die Cg-Shadingsprache in Verbindung mit OpenGL erforderte viel Arbeit und daher war die Implementierung der FM-Methode eine gute Übung für die spätere Implementierung der Hierarchischen Methode auf der GPU. Die Abbildung 5.2 zeigt die Skizze für den Ablauf der Full-Matrix-Methode auf der GPU. Dort wird explizit gezeigt, welche Operationen auf der CPU und welche auf der GPU ausgeführt werden. Zuerst wird die Szene mit Patches generiert und eventuell noch ein Subdivide-Algorithmus zur Verfeinerung des Patches aufgeführt. Nachdem die Daten der Patches zur Verfügung stehen, können die vier Eingabetexturen für die Formfaktorberechnung erstellt werden. Die Berechnung der Formfaktoren wird mit der zweiten Formfaktorvereinfachung durchgeführt und gibt als Ergebnis Punkt-zu-Punkt Formfaktoren aus. Dazu werden die folgenden Daten benötigt:

- Mittelpunkt des Senderpatches (p1.xyz)
- Normale des Senderpatches (n1.xyz)
- Größe des Senderpatches (s)
- Mittelpunkte des Empfängerpatches (p2.xyz)
- Normale des Empfängerpatches (n2.xyz)

Diese Daten werden, wie in der Abbildung 5.3 erkennbar, in vier Texturen gespeichert, um den Zugriff auf alle 13 Werte pro Texel zu ermöglichen. Die Berechnung der Formfaktoren übernimmt der folgende Shader aufgeführt im Anhang 10. Damit die Formfaktoren im Render-To-Texture Modus berechnet werden, wird die zuvor initialisierte Rendertextur aktiviert und nach der Berechnung deaktiviert. Anschließend werden die Daten aus der Render-Textur mit `glReadPixels()` ausgelesen und in einem Array gespeichert, um die nachfolgende Erstellung der M-Matrix aus den Formfaktor- und ρ-Daten zu vereinfachen. Die M-Matrix wird zusammen mit den Vektoren B und E dem Shader zum Lösen des Gleichungssystem übergeben. Das Resultat ist eine 1D-Textur mit den Radiosity-Werten. Damit die Lösung konvergiert, wird sie in die nächste Iteration übergeben, bis die Berechnung durch Konvergenz abgeschlossen ist. Mithilfe der Texturkoordinaten bezogen auf die 1D-Textur werden die Farbwerte dann beim Zeichnen der Patches entsprechend gesetzt. Das Resultat der Berechnung einer Cornell-Box Szene zeigt die Abildung 5.3. Die Umsetzung der FM-Methode erfolgte ebenfalls durch Implementierung der Jacobi-Methode mit dem Ziel:

Eigene Umsetzung der **Full-Matrix**-Methode auf der GPU

$$r = E - M * B$$
$$B = B + r$$

zu lösen. Die Berechnung der Radiosities B erfordert drei Shaderdurchgänge. Der Quellcode der Shader befindet sich im Anhang unter 10.2.1. Der erste Shader multipliziert jeweils die Elemente aus einer Zeile der M-Matrix mit den Werten des B-Vektors, zu sehen in Abbildung 5.4 als Shader 1. Der zweite Shader multipliziert alle im ersten Shader berechneten Werte einer Zeile der MB-Matrix zusammen. Die erste Überlegung ging

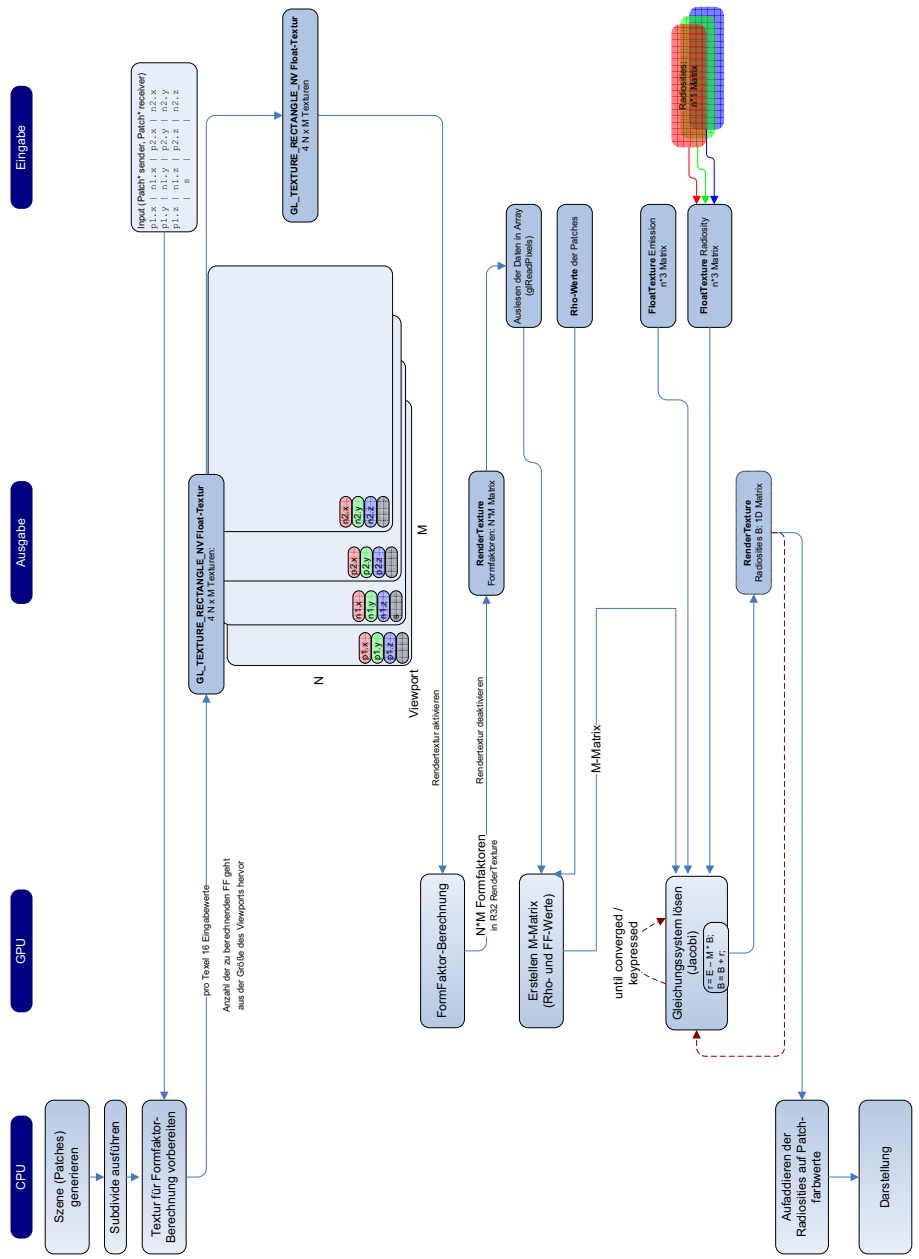


Abbildung 5.2: Entwurf eines FM Algorithmus auf der GPU

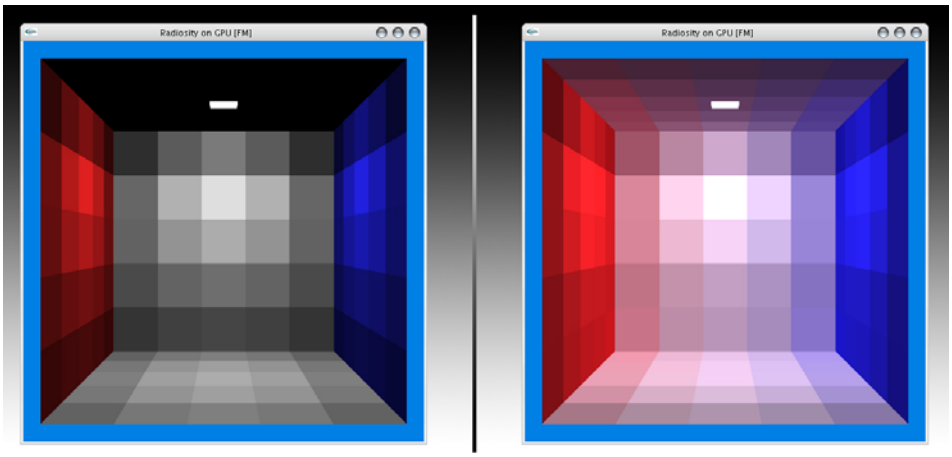


Abbildung 5.3: Anfangs- und konvergierte Szene der FM-Methode auf der GPU

dahin, eine simple Schleife über alle Objekte zu verwenden, wie Shader 2 in Abbildung 5.4 verdeutlicht. Ein besserer Weg ist, den GPU-Algorithmus Reduce aus dem Buch GPUGems[ea04] zu verwenden. Anstatt eines Reduce der Daten wird eine Multiplikation der Daten einer Zeile der MB-Matrix durchgeführt. Der letzte Schritt berechnet das Residuum $r = E - MB$ und addiert es auf die Radiosity B. Die neue Radiosity B ist erforderlich für den nächsten Iterationsschritt und wird diesem zur Verfügung gestellt, dargestellt als Shader 3 in Abbildung 5.4. Die drei Shader werden hintereinander ausgeführt und stellen eine Jacobi-Iteration dar. Für jede weitere Iteration müssen die Shader mit dem neuen Radiosityvektor B durchlaufen werden.

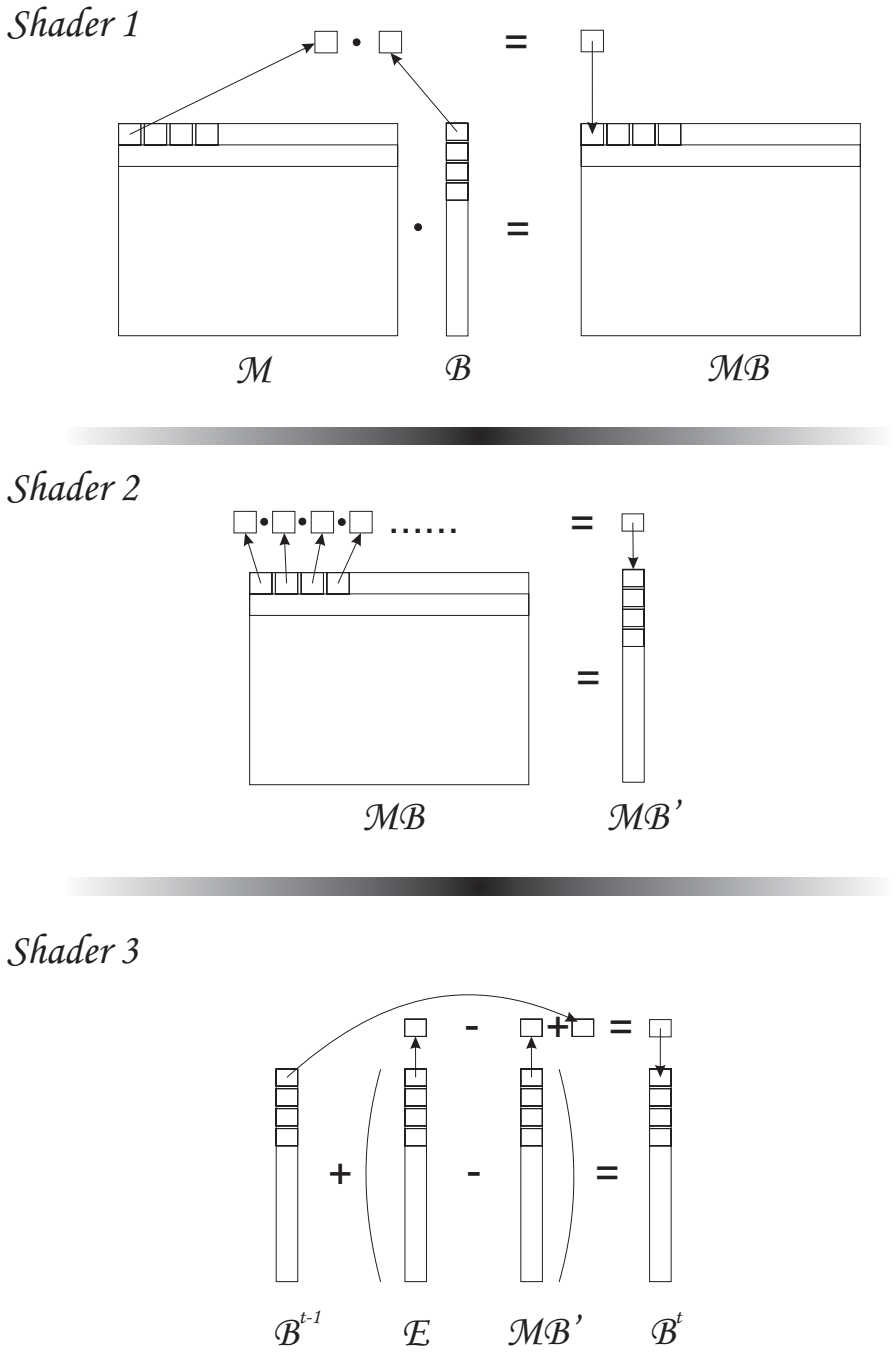


Abbildung 5.4: Full-Matrix GPU-Methode als drei Shader Modell

5.2 Progressive-Refinement-Methode

Die absolute Führung in puncto Radiosity auf der Grafikhardware besitzt das Paper von Greg Combe, Mark J. Harris und Anselmo Lastra [CjHL04]. Dieses Paper beschreibt die Berechnung einer Radiositylösung mit Progressive Refinement auf der GPU. Mark Harris ist der Autor der RenderTexture Klasse. Der Vorteil der PR-Methode liegt darin, dass sie sich gut auf die parallele Datenverarbeitungsstruktur der GPU abbilden lässt. Die PR-Methode speichert keine explizite Radiositymatrix und kann auch während der laufenden Verarbeitung schon Berechnungsergebnisse anzeigen. Dabei legten die Autoren Wert darauf, keine Vorberechnungen machen zu müssen und vermieden das Zurücklesen von Daten. Mithilfe der RenderTexture-Klasse ist es möglich, die berechneten Radiositywerte direkt in eine Textur zu schreiben und direkt zum Rendern zu nutzen.

Progressive Refinement auf der GPU nach Coombe[CjHL04]

Abbildung 5.5: Beispielszene berechnet mit Progressive Refinement auf der GPU[CjHL04]

Die Implementierung sieht eine konventionelle patchorientierte Radiosity auf einem uniform unterteilten Arbeitsbereich vor. Die Werte für die Radiosity- und Emissionsenergien werden assoziiert mit den Patches in Texturen gespeichert. Der Algorithmuskern arbeitet in Iterationen für alle empfangenden Polygone und testet jedes Element auf Sichtbarkeit. Diese „Gather“-Operation nutzt die SIMD-Fähigkeit der Fragmentprogramme voll aus.

Der Pseudcode der Progressive Refinement Implementation:

```
set initial shooter energy E
while not converged
  render scene from shooter's POV
  store as item Buffer B
  for every receiving polygon
    for every texel
      if texel is visible in B
        compute form factor FF
        deltaE = ρ·FF·E
        add deltaE to residual
        add deltaE to radiosity
  shooter's energy E=0
  select next shooter
```

5.2.1 Sichtbarkeit

Coombe benutzt eine erweiterte Hemicube-Methode zur Bestimmung der Visibilität von einem Sender. Ein Vertex-Programm hilft, die Vertices zu verzerren, damit die Visibilitätsberechnungen mithilfe einer stereographischen Projektion durchgeführt werden können. Der Vorteil, ist die einmalige Iteration über die Geometrie, statt einer fünffachen Iteration beim Hemicubeverfahren. Zuerst wird eine stereographische Map aus der Sicht des Senderpatches generiert und als Item-Buffer abgespeichert. Nun werden alle Patches, die Energie von dem Sender erhalten, orthographisch in einen Framebuffer der Größe seiner Radiositytextur projeziert. So wird es möglich, bei einer eins-zu-eins Korrespondenz der Fragmente des Framebuffers und der Radiositytexels, direkt auf diese zuzugreifen. Die Rückprojektion des Radiositytexels in den Viewport des Shooters erfolgt durch ein Fragment Programm. Da die stereographische Projektion genutzt wird, erfolgt der Zugriff auf den Item-Buffer durch Vektor Normalisierung. Ist nun die ID im Item-Buffer gleich der ID des Polygontexels so wird angenommen, dass das Texel vom Sender aus sichtbar ist. Ein Hardware-Verdeckungsalgorithmus hilft, die Anzahl der zu testenden Polygone zu senken. Während eines Sichtbarkeitstestes wird der Verdeckungsalgorithmus auf jedes Polygon angewendet, nur die Polygone, die den Test bestehen, werden bearbeitet. Dies verspricht teilweise einen Vorteil von 50% an Polygonen, die nicht bearbeitet werden müssen.

5.2.2 Formfaktoren

Nachdem die Visibilität der Patches bestimmt wurde, müssen nun die Formfaktoren berechnet werden, damit der Anteil der Radiosity, den sie erhalten, berechnet werden kann. Der Standard Fläche-zu-Fläche Formfaktor geht davon aus, dass die Differentialflächen klein in Bezug auf die Distanz zwischen den Flächen ist. Ist dies nicht der Fall, so entstehen Artefakte bei nah aneinander liegenden Flächen, wie z.B. in Ecken. Eine Reduzierung dieser Fehler wird erzielt, indem eine Kreisannäherung zur Berechnung des FF benutzt wird. Die Abbildung 5.6 verdeutlicht die Idee. Das Polygon j wird unterteilt und mit einem Kreis angenähert. Die Annäherung berechnet einen Formfaktor

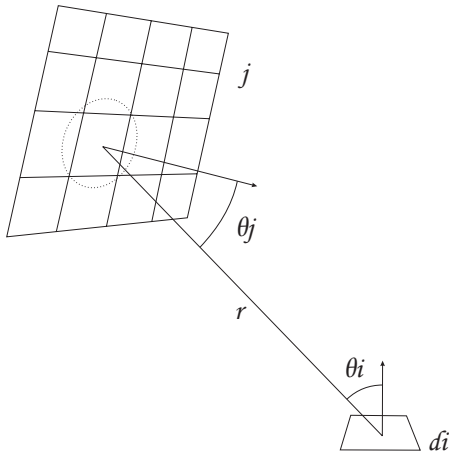


Abbildung 5.6: FF zwischen Differntialfläche di und Polygon j[CjHL04]

zwischen einem finiten Flächenelement j und einer Differentialfläche di, wobei j in m flächenorientierte Kreise unterteilt wird. Es gilt

$$F_{j \to di} = A_j \sum_{i=1}^{m} \frac{cos\theta_j cos\theta_i}{\pi r^2 + \frac{A_j}{m}} \tag{5.5}$$

zu lösen. Diese Aufgabe erledigt das Fragmentprogramm im Pseudocode:

```
deltaE computeFF()
  r = ShootPos - TexelPos;
  dist2 = dot(r, r);
  r = r * rsqrt(dist2);
  cosi = dot(TexelNormal, r);
  cosj = dot(ShootNormal, r);
  top = max(cosi * cosj, 0);
  bot = π * dist2 + ShootDeltaArea;
  F_ij = IsVisible() ? top/bot : 0.0;
  deltaE = TexelColor * ShootE() * F_ij;
```

5.2.3 Wahl des nächsten Senders

Die schnelle Konvergenz des Progressive Refinement Algorithmus liegt in der Wahl des Senders mit der größten unshot Radiosity. Dieses Patch wird durch einen einfachen z-Buffer Algorithmus gefunden. Zunächst wird ein ein-Pixel Framebuffer aufgesetzt. Ein Fragment Programm liest die $(1x1)$ Mipmap-Level der Residuuentextur aus und erhält den Durchschnitt aller Texel. Dieser Vorgang wird für jedes Polygon durchgeführt. Die durchschnittlichen Residuumwerte werden mit der Basisauflösung der Textur und der Fläche des Polygons multipliziert. Der Kehrwert des Wertes wird als Tiefenwert und die ID des Polygons als Farbe gerendert. Der z-Buffer-Test wählt nun automatisch das Polygon mit der höchsten unshot/residuen Energie. Die Information wird aus dem Framebuffer zurückgelesen.
Eine Möglichkeit, den Prozess des Energieverteilens zu beschleunigen, ist es, nicht auf

dem Level der Basisauflösung Energie zu verschiessen, sondern auf einer niedrigeren Auflösung. Coombes Algorithmus verarbeitet 16 Texel pro Schuss. Die CPU bleibt zuständig für die Handhabung der Senderpositionen auf jedem Polygon.

5.2.4 Adaptive Unterteilung

Zusätzlich zur uniformen Unterteilung wurde eine adaptive Unterteilung für den PR-Algorithmus programmiert, da sie für bessere Ergebnisse bei der Darstellung der Szene sorgt. Bereiche mit viel Informationen werden zu wenig und Bereiche mit wenig Informationen zu viel abgetastet, daher die adaptive Unterteilung. Die verwendete Subdivide-Methode unterteilt die Geometrie der Szene hierarchisch und erstellt dabei einen Quadtree. Die Radiosity des unterteilten Patches wird bilinear gefiltert an die Blätter der Bäume weitergegeben. Die Radiosity wird mit der GPU auf den Texeln der Blättertexturen berechnet. Nach der Berechnung werden die Blättertexturen untersucht und bei Bedarf die Patches unterteilt. Unterteilt wird aufgrund der Gradienten der Radiositywerte. Die Gradienten der Radiosities werden durch ein Fragmentprogramm berechnet und auf Diskontinuitäten untersucht. Die Anzahl der diskontinuierten Fragmente wird bestimmt und bei einer bestimmten Schwelle wird das zugehörige Patch unterteilt.
Die adaptive Unterteilung ermöglicht es, Sender auf einer feineren Hierarchie zu finden. Es wird der gleiche z-Buffer Algorithmus zum Finden das Patches verwendet, wie zuvor beschrieben. Der einzige Unterschied besteht darin, dass die Energie eines Knotens verwendet wird, nicht die des ganzen Patches.

5.2.5 Darstellung

Zur Darstellung der Radiositytexturen eines Quadtrees traversiert die CPU durch den Baum und unterteilt diesen rekursiv. Sobald ein Blatt erreicht ist, wird ein Viereck mit dem Radiositywert des Blattes als Farbe gezeichnet. Wenn alle Blätter gezeichnet sind, so werden alle Patches der Szene mit der adaptiven Radiositymethode texturiert. Die Abbildung 5.7 zeigt eine 260.000 Elemente Szene berechnet mit dem Progressive Refinement Alorithmus.

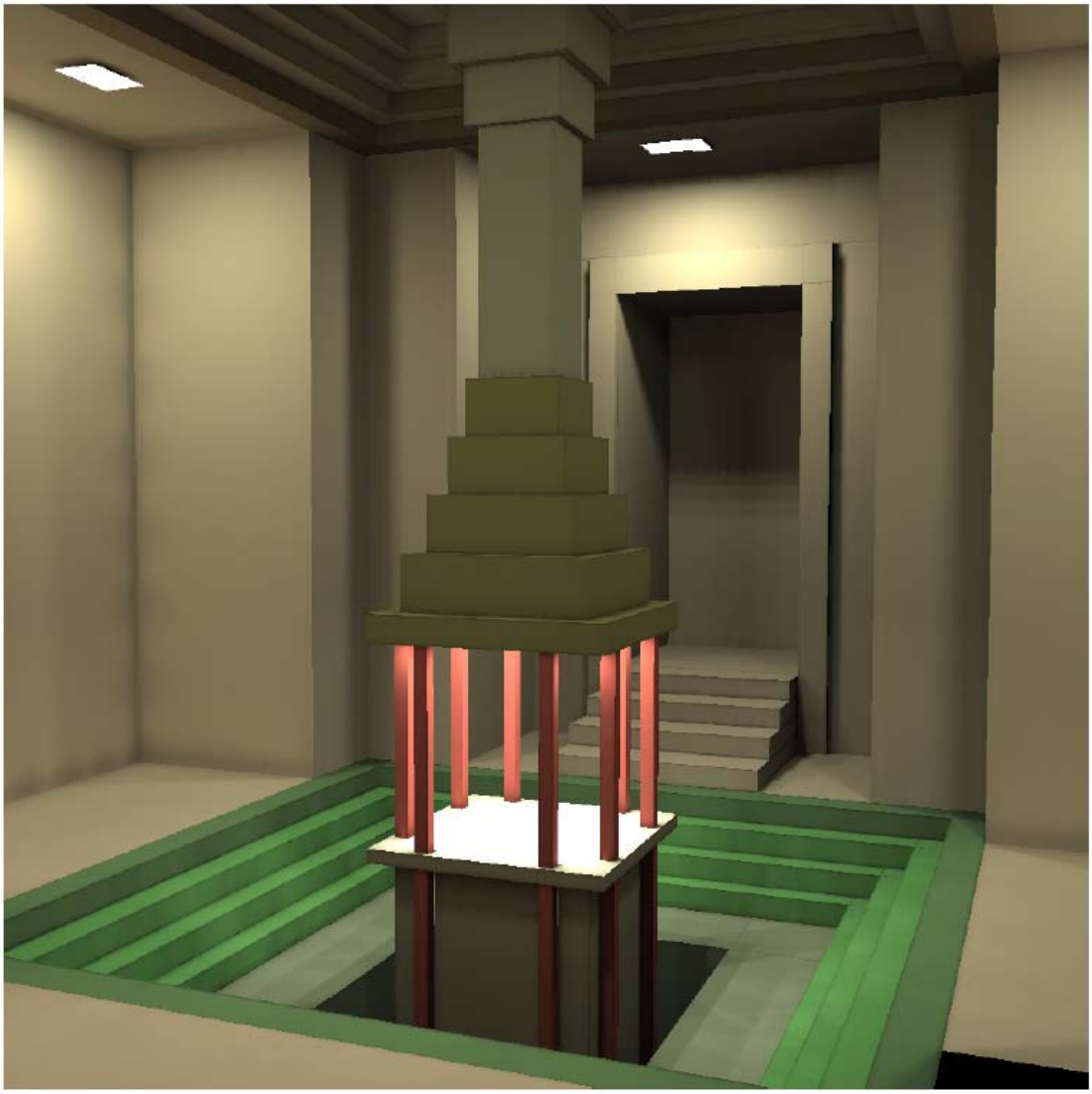


Abbildung 5.7: Szene mit 260.000 Elementen berechnet mit PR auf der GPU[CjHL04]

6 Hierarchisches Radiosity auf der GPU

Die Hauptaufgabe der Diplomarbeit bestand darin, neben dem Studium und der Einarbeitung in Radiosityalgorithmen, sowohl auf der CPU als auch auf der GPU, ein hierarchisches Radiositysystem auf der GPU zu implementieren. Dazu musste eine Lösung gefunden werden, wie die drei Grundmethoden des hierarchisches Radiosityalgorithmus, **Refine**, **Gather** und **PushPull** umzusetzen sind. Die Herausforderung dabei war, die Datenstrukturen, die die Basis darstellen, auf der GPU zugänglich zu machen. Zum einen ist das die Datenstruktur der Patches, den Szenenelementen, und zum anderen die Datenstruktur der Links, die Verbindungen zwischen den Patches. Die Patch-Datenstruktur ist ausgehend von Wurzelpatches wie ein Quadtree aufgebaut. Jedes Patch kann vier Kinder besitzen, die wiederum Kinder besitzen können und so fort. Die Klasse **Patch** repräsentiert dabei ein Patch mit den folgenden Eigenschaften, den Eckpunkten, der Größe, der Radiosity, der Normalen und der Link-Datenstruktur, um nur einige wichtige Merkmale der Klasse zu nennen. Die Link-Datenstruktur ist eine Liste von Elementen der Klasse **Link**. Sie stellt eine Verbindung zwischen zwei Patches dar, dem Senderpatch und dem Empfängerpatch, dem sogenannten Gather-Link. Diese Klasse besitzt einen Zeiger auf das Senderpatch und den Formfaktor zwischen den beiden Patches.
Die Überlegungen zur Umsetzung des Algorithmus auf der GPU führten zu einem hybriden System. Dies hat den Vorteil, dass die beim Ausführen des Algorithmus auftretenden, wachsenden und nicht bekannten Datenmengen, besser verarbeitet werden können. Die Aktualisierung der Patch-Hierarchie, der Kinderstellung, sowie der Aktualisierung des Links eines Patches, erfordern ein Auswerten der berechneten Daten auf der CPU. Die Umsetzung einer Patch-Hierarchie samt Linkstruktur auf der Grafikkarte widerspricht der steifen Programmierumgebung der GPU. Daraus resultiert das folgende Datenberechnungsmodell, das Anwendung in dieser Arbeit findet:

1. Ermitteln zu berechnender Daten
2. Erstellung einer Textur mit den Daten
3. Berechnung auf den Daten durch den Shader (pro Fragment)
4. evtl. Multipass-Berechnung durch mehrere Shader hintereinander
5. Zurücklesen der Daten
6. Auswerten der Daten und Postprocessing

Die größte Schwierigkeit bei der Umsetzung des Systems ist es, ein adaptives System zu generieren, welches die Möglichkeit bietet, verschiedenste Szenen bearbeiten zu können. Die Abbildung 6.1 zeigt ein Modell, das verdeutlicht, wie hierarchisches Radiosity umgesetzt werden kann.

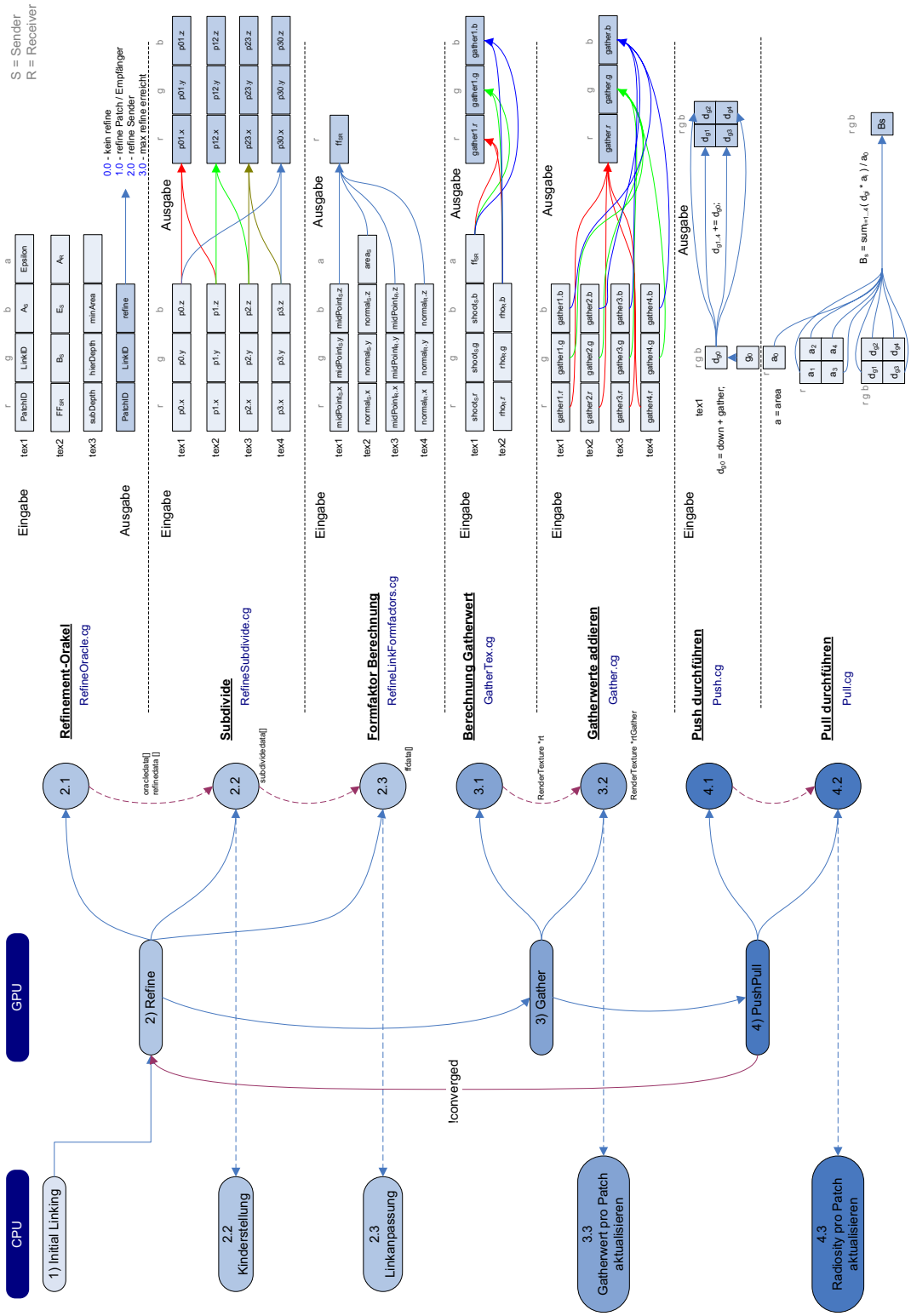


Abbildung 6.1: Modell zur Umsetzung von HR auf der GPU

Der Algorithmus startet zunächst mit der Generierung der Szene mithilfe von Patches und verlinkt anschließend alle Patches untereinander. Nachdem der Vorgang abgeschlossen ist, wird der nächste Schritt, das Refine, auf der GPU ausgeführt. Alle im folgenden beschriebenen Shader sind, mit Quellcode-Dokumentation, im Anhang zu finden.

6.1 Refine

Mit einer Schleife über alle Wurzelpatches werden die Links jedes einzelnen Patches mit dem Refinement-Orakel auf Verfeinerung untersucht. Als Bewertungsmaßstab wird das BFA-Kriterium genutzt. Die Radiosity des Senderpatches B_s multipliziert mit dem Formfaktor FF_{se} des Link und der Empfängerfläche A_e wird gegen eine Schranke ϵ getestet. Als Eingabe für den Shader, der als Refinement-Orakel fungiert, werden folgende Daten benötigt:

- $PatchID$ - Integer-ID das aktuellen Patches
- $LinkID$ - Integer-ID des aktuellen Links
- A_s - Senderfläche
- ϵ - Schranke, unterhalb derer nicht unterteilt wird
- FF_{se} - Formfaktor Sender zum Empfänger
- B_s - Radiosity des Senderpatches
- E_s - Emmissionsenergie des Senderpatches
- A_e - Empfängerfläche
- $subDepth$ - maximale Hierarchietiefe
- $hierDepth$ - aktuelle Hierarchietiefe
- $minArea$ - minimale Flächengröße

Die elf Werte werden in drei Texturen gespeichert, wie in Punkt 2.1 in der Abbildung 6.1 zu sehen ist. Zur Orientierung innerhalb der hierarchischen Patch-Datenstruktur werden die PatchID und LinkID gespeichert. Diese Werte werden lediglich durch den Shader durchgereicht und zusammen mit dem Refine-Ergebnis ausgegeben. Somit entsteht ein rgb-Rückgabewert, der als Rotwert die PatchID, als Grünwert die LinkID und als Blauwert den Refinewert speichert. Dabei kann der Refinewert die vier folgenden Werte annehmen: 0.0 für keine Refine, 1.0 für Patch refine, 2.0 für Sender refine und 3.0 falls die maximale Hierarchietiefe erreicht ist. Anschließend werden die Orakeldaten der Links eines Patches zurückgelesen und ausgewertet, welche Links für die weitere Verarbeitung vorbereitet werden müssen und gesondert gespeichert.
Anhand der Daten des Refinement-Orakels und eines Tests, ob Kinder, auf der jeweiligen Seite eines Links, auf der Kinder erstellt werden sollen, schon vorhanden sind, werden die

Eckpunkte p0, p1, p2 und p3 der zu unterteilenden Patches in vier Texturen gespeichert. Die Mittelpunktberechnung übernimmt wiederum ein Shader, Punkt 2.2 in Abbildung 6.1. Das Ergebnis des Shader sind jeweils die Mittelpunkte der Konstellationen p01, p12, p23 und p30. Durch diese Daten können auf der CPU die Kinderpatches erstellt und die Hierarchie angepasst werden.
Nun wird sich wieder der Ergebnisdaten des Refinement-Orakels gewidmet, da nun die Links unter Berücksichtigung des Refine-Wertes bearbeitet werden. Damit neue Links erstellt werden können, müssen Formfaktoren der Patches berechnet werden, zwischen denen ein Link gebildet wird. Die Berechnung der Formfaktoren übernimmt wieder ein Shader. Die Anwendung zeigt Punkt 2.4 in Abbildung 6.1. Zur Berechnung des Formfaktors zwischen Sender und Empfänger werden folgende Daten benötigt:

- $midPoint_s.xyz$ - Mittelpunkt des Senders
- $normal_s.xyz$ - Normale des Senders
- $area_s$ - Fläche des Senders
- $midPoint_e.xyz$ - Mittelpunkt des Empfängers
- $normal_e.xyz$ - Normale des Empfängers

Der Rückgabewert des Shaders ist ein einziger float-Wert, der Formfaktor FF_{se} zwischen Sender und Empfänger. Nachdem sowohl die erforderlichen Kinder und Formfaktoren erzeugt wurden, lassen sich nun auch die Links erstellen und der Link-Datenstruktur des jeweiligen Patches hinzufügen. Rekursiv werden anschließend die neu erzeugten Kinder der Refine-Methode unterzogen.

6.2 Gather

Die Gather-Methode sammelt für jedes Patch anhand der Gather-Links, die in der Link-Datenstruktur gespeichert sind, die ρ- und formfaktorgewichtete Radiosity der Sender ein, addiert diese und fügt sie zur Radiosity des Patches hinzu. Mithilfe einer Schleife über alle Gather-Links werden die folgenden Daten eingesammelt

- $shoot_s.rgb$ - Radiosity des Senderpatches
- ff_{se} - Formfaktor Sender > Empfänger
- $rho_e.rgb$ - Rhowert des Empfängers

und in zwei Texturen gespeichert. Sie dienen als Eingabe des Shaders, der die Gatherwerte bilden soll, Punkt 3.1 in Abbildung 6.1. Pro Fragment wird jeweils ein Gatherwert wie folgt berechnet

```
gather = ( rho * ( shoot * ff ) );
```

und für das Fragment gespeichert. Die Anzahl der zu bearbeitenden Fragmente spiegelt die Anzahl der Links des Patches wieder. Die Gatherwerte werden mithilfe einer Instanz der RenderTexture und dem Render-To-Texture Modus in eine Textur geschrieben, so dass sie im nächsten Shader direkt als Eingabe dienen. Der zweite Teil des Gather-Prozesses hat nun die Aufgabe die berechneten Gatherwerte der Textur zu addieren. Hier bietet sich der Einsatz des Reduce-Algorithmus an, so wie er im Buch GPU Gems[FK03] beschrieben wurde. In dem Buch wurde der Algorithmus verwendet, um den maximalen Wert der in einer Textur gespeicherten Werte zu ermitteln. Der Code des Gather-Algorithmus:

```
cgGLBindProgram(GatherProgram);
cgGLEnableProfile(fpProfile);
cgGLSetTextureParameter(GatherTex, rt->GetTextureID());
cgGLEnableTextureParameter(GatherTex);

for (int i=texSize/2; i>0; i = i/2) {
  glViewport(0, 0, i, i);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluOrtho2D(0, i, 0, i);
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();

  glBegin(GL_QUADS);
    glTexCoord2f (-1.0, -1.0); glVertex2f (0.0, 0.0);
    glTexCoord2f (i*2-1, -1.0); glVertex2f (i, 0.0);
    glTexCoord2f (i*2-1, i*2-1); glVertex2f (i, i);
    glTexCoord2f (-1.0, i*2-1); glVertex2f (0.0, i);
  glEnd();

  glFlush();

  glCopyTexSubImage2D(GL_TEXTURE_RECTANGLE_NV, 0, 0, 0, 0, 0, i, i);
}
```

Zuerst wird der Shader, hier GatherProgram, geladen und die Textur mit den Gatherwerten, hier die RenderTexture *rt, dem Shader zur Verfügung gestellt. Ausgehend von einer 2^n-Textur werden nun bei jedem Schleifendurchlauf jeweils vier Fragmente auf ein Fragment reduziert. Das setzt voraus, dass die Texturkoordinaten, der Viewport und die orthographische Projektion auf die Anzahl der Ergebnisse eingestellt sind, also pro Iteration ein Viertel der zu berechnenden Daten. Der Shader zur Addition aller Werte:

```
gather1 = f3texRECT(gatherTexture, texCoords);
gather2 = f3texRECT(gatherTexture, texCoords + float2(0,1) );
gather3 = f3texRECT(gatherTexture, texCoords + float2(1,0) );
```

```
gather4 = f3texRECT(gatherTexture, texCoords + float2(1,1) );

// Rückgabewert: Addition der vier Gather-Werte
return float3(gather1 + gather2 + gather3 + gather4);
```

Nachdem alle Fragmente bearbeitet wurden, muss die Ergebnistextur in die aktuelle Textur geschrieben werden, damit sie für den nächsten Iterationsschritt als Eingabe zur Verfügung steht. Die Schleife läuft so lange bis der Viewport nur noch ein Pixel groß ist. In dem entsprechenden Fragment befindet sich dann der Wert aller addierten Gatherwerte. Der Wert wird zurückgelesen und der Gatherwert des zugehörigen Patches aktualisiert.

6.3 PushPull

Die Umsetung der PushPull-Methode auf der GPU wird in die zwei Teilalgorithmen Push und Pull unterteilt und getrennt voneinander betrachtet. Die CPU-Version ist eine Verzahnung der beiden Methoden, die rekursiv über die Kinderpatches ausgeführt werden. Dabei wird die Pull-Methode direkt bei abgeschlossener Push-Methode eines Teilbaumes ausgeführt und nicht nachdem die komplette Push-Methode abgeschlossen ist. Die Betrachtung der beiden Methoden als Teilprobleme ermöglicht die Umsetzung auf der Grafikkarte.
Die Voraussetzung für den Algorithmus auf der Grafikkarte ist eine bestimmte Datenrepräsentation. Der Teilbaum des Quadtrees eines Rootpatches wird auf Texturen abgebildet und zwar so, dass für jede Hierarchiestufe eine Textur angelegt wird. Zudem befindet sich ein Vaterknoten immer direkt über seinen Kindern, somit ist gewährleistet, dass mithilfe der Texturkoordinaten der Kinder, in der Textur einer Hierarchiestufe darüber, auf den Vater, bzw. auf die dort gespeicherten Werte, zugegriffen werden kann.
Die für das PushPull benötigen Daten sind:

- down-Wert (gesammelte Gather-Energie der Väterknoten)
- gather-Wert (Gather-Energie der Knoten)
- area-Wert (Flächengrößen der Knoten)

Die erforderlichen Texturen pro Berechnungschritt
für **Push**:

- Textur der down-Werte der Väterknoten
- Textur der gather-Werte der Knoten

für **Pull**:

- Textur der down+gather - Werte (RenderTextur)
- Textur der Flächengrößen der Knoten

- Textur der Flächengrößen der Väterknoten
- Textur mit 0/1-Wert - Kind vorhanden nein/ja

Diese Daten müssen in Form mehrerer Texturen zur Verfügung gestellt werden. Die Ergebnisse werden wiederum in Render-Texturen gespeichert, um sie in der nächsten Iteration direkt als Eingabe nutzen zu können ohne die Daten zurücklesen zu müssen. Das Zurücklesen der Daten wird erst zum Ende des Pullschrittes erforderlich. Der Ablauf von PushPull wird in den Punkten 4.1 und 4.2 in Abbildung 6.1 gezeigt.
Die **Push**-Methode benötigt die zuvor genannten Eingabetexturen down und gather. Die initiale down-Textur für den Wurzelknoten ist lediglich ein Nullvektor $(0, 0, 0)$, da kein Vater vorhanden ist. Mithilfe des Nullvektors und des Gatherwertes des Wurzelknotens wird der Wert d_{g0} erzeugt, der auf die Gatherwerte der Kinder addiert wird. Der Viewport wird dabei auf die Größe der Textur auf der momentan befindlichen Hierarchiestufe eingestellt. Nach der Bearbeitung des Wurzelknotens ist der d_{g0}-Wert auf die vier Kinder zu addieren, also auf eine $2x2$ Textur mit den Gatherwerten der Kinder. Pro Fragmentbearbeitung wird ein Push-Durchgang ausgeführt. Anschließend wird das Ergebnis in einer $2x2$ RenderTextur gespeichert, die als Eingabe zur Berechnung der Kinder in der $4x4$ großen Textur dient. Bei einer maximalen Texturgröße von $2048x2048$ kann die Hierarchie in der Tiefe 11 noch bearbeitet werden. Nachdem alle Hierarchielevel bearbeitet wurden, existieren für jedes Level jeweils eine Textur mit Gatherwerten. Nun müssen diese Werte flächengewichtet an die Väterknoten weitergegeben werden.
Diese Aufgabe übernimmt die **Pull**-Methode. Innerhalb des Pull-Shaders wird zunächst anhand der binären Textur geprüft, ob das aktuelle Patch ein Blatt ist oder nicht. Ist es ein Blatt, so wird der zugehörige Gatherwert aus der Render-Textur als Ergebnis gespeichert. Sofern aber Kinder vorhanden sind, werden die Gatherwerte $d_{g1..4}$ und die Flächengrößen $a_{1..4}$ der vier Kinder, sowie die Flächengröße a_0 des Patches eingelesen. Der Radiositywert für das Patch ergibt sich somit aus:

$$B_s = \frac{\sum_{i=1}^{4} d_{gi}}{a_0} \tag{6.1}$$

Der Vorgang wird ebenfalls über alle Hierarchiestufen beginnend von dem untersten Level zu den Wurzelpatches durchgeführt. Pro Fragmentberechnung wird hier ebenfalls ein Pull-Schritt durchgeführt.

7 Ergebnisse & Bewertung

Dieses Kapitel gibt einen Überblick über die Ergebnisse der Diplomarbeit, sowohl visuell als auch aus der Sicht des Programmierers und bewertet kritisch den Ansatz zur Umsetzung von Hierarchischem Radiosity auf der GPU.

7.1 Ergebnisbilder

Die Ergebnisbilder zeigen die Cornellbox in verschiedenen Ausführungen. Dabei variieren die Beleuchtungs- und Patcheigenschaften, sowie die Auflösungsstufen. Die Abbildung

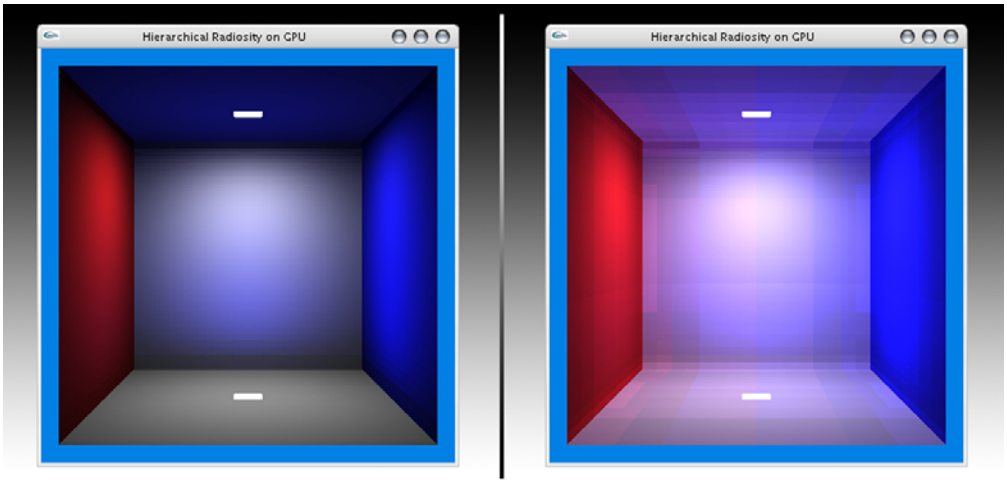


Abbildung 7.1: Szene mit 75.000 Polygonen berechnet mit HR auf der GPU

7.1 zeigt eine Szene, die zu Beginn aus sieben Wurzelpatches besteht, den fünf Wänden und zwei Lichtquellen. Die Lichtquelle an der Decke strahlt weißes Licht, die Lichtquelle auf dem Boden blaues Licht, aus. Das linke Bild in der Abbildung zeigt die Szene nach der ersten Iteration, also nach dem Austausch der direkten Lichtenergie. Das rechte Bild zeigt den Zustand nach sieben Iterationen, dem konvergierten Zustand, nachdem keine visuellen Änderungen mehr eintraten. Sehr gut erkennen lässt sich der Strahlungsaustausch auf unterschiedlichen Leveln durch die unterschiedlich hellen Flächen der Szene. Eine bilineare Filterung wurde nicht eingesetzt, lediglich die Tatsache, dass die Szene ca. 75.000 Patches nach der siebten Iteration umfasst, lässt es bilinear gefiltert erscheinen. Die maximale Hierarchietiefe liegt in der Szene bei neun, das Refine-ϵ bei 0.0003 und die Mindestflächengröße bei 0.0002. Eine ähnliche Szene mit verändeter Beleuchtung zeigt

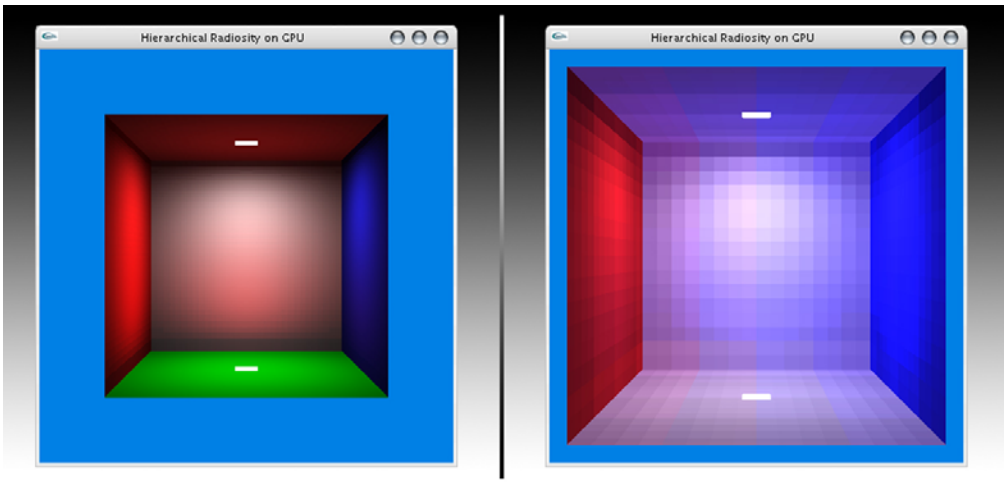


Abbildung 7.2: Szene mit 75.000 und 1.700 Polygonen

die linke Grafik in Abbildung 7.2. Diese wurde mit den gleichen, zuvor genannten, Parametern berechnet. Die rechte Grafik ist eine Variante mit verringerter Hierarchietiefe von vier und einem Refine-ϵ von 0.003. Die Abbildung 7.3 zeigt die Szene aus der Abbil-

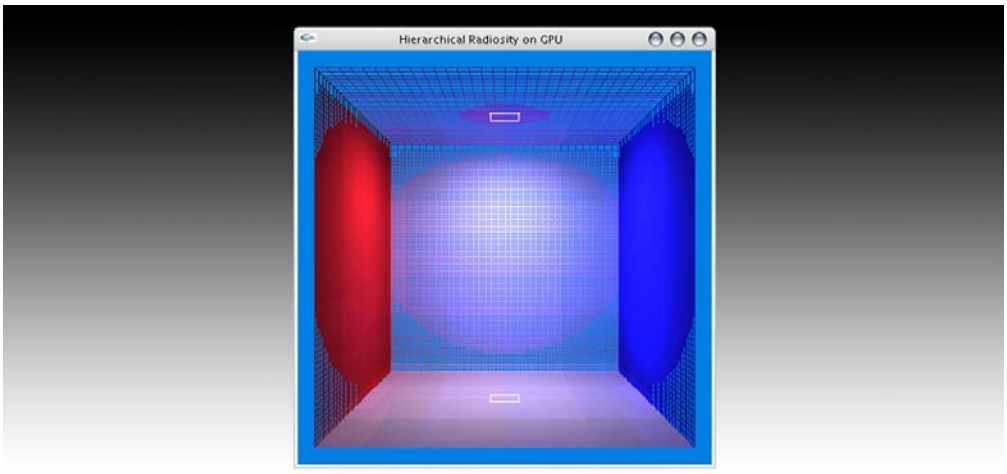


Abbildung 7.3: Wireframe-Ansicht der Szene aus Abbildung 7.1

dung 7.1 im Wireframe-Modus. In diesem kann man die unterschiedliche Unterteilung der Szene durch die Subdivide-Methode mithilfe des BFA-Kriteriums beobachten.

7.2 Berechnungszeiten

Das Unterkapitel Berechnungszeiten zeigt die verschiedenen Laufzeiten der Algorithmen auf der CPU und der GPU auf. Die erste Tabelle zeigt einen Überblick über die Full-

Matrix und Progressive Refinement Algorithmen, sowohl auf der CPU als auch auf der GPU. Die CPU-Algorithmen wurden von der Computergrafikgruppe des FB4 der Universität Koblenz zur Verfügung gestellt. Die FullMatrix Implementation auf der GPU ist die in 5.1 vorgestellte Version.

Szene: Cornellbox, 150 Patches, bis Konvergenz

		Gesamt
CPU		
	FullMatrix[1]	0,411 s
	Progressive Refinement[2]	0,415 s
GPU		
	FullMatrix[3]	0,098 s

Das Ergebnis dieses Tests ist, dass die Berechnung der Gleichung $E = MB$ auf der GPU beschleunigt ablaufen kann. Die Berechnung einer Szene mit 150 Patches setzt eine $150x150$ große M-Matrix voraus, die noch gut in einer Textur gespeichert werden kann. Ab einer Größe von $2048x2048$ wird es problematisch, da hier die maximale Texturgröße liegt. Die Berechnung müsste auf mehrere Texturen aufgeteilt werden. Die Progressive Refinement-Methode ist im Grunde schneller als die FullMatrix-Methode der CPU, benötigt aber zu viele Iterationen um die Konvergenz zu erreichen, hier 500 Iterationen. Der nächste Test beruht auf der Implementation der hierarchischen Radiosity Modelle. Da diese Modelle die Konvergenz sehr schnell erreichen, wurden hier zwei Messpunkte gewählt, zunächst die 1. Iteration, die für das Verteilen der direkten Energie steht, und die 7. Iteration, als letzte Iteration zum konvergierten Zustand. Dabei wurden die Zeiten, jeweils zum Berechnen einer 1.700 und einer 75.000 Patch großen Cornellbox-Szene gemessen.

Szene: CB, 1.700 Patches, bis Konvergenz, 1. Iteration

	Refine	**Gather**	**PushPull**	**Gesamt**
CPU HR	0,122 s	0,01 s	0,012 s	0,144 s
GPU HR	0,572 s	0,258 s	0,018 s [4]	0,848 s

Szene: CB, 1.700 Patches, bis Konvergenz, 7. Iteration

CPU HR	0,007 s	0,015 s	0,01 s	0,032 s
GPU HR	0,172 s	0,323 s	0,01 s [4]	0,505 s

[1] Zweite FF-Vereinfachung
[2] Prisma-FF, 180 Patches, 500 Iterationen
[3] Zweite FF-Vereinfachung, Algorithmus aus 5.1
[4] CPU-Version

Szene: CB, 75.000 Patches, bis Konvergenz, 1. Iteration				
CPU HR	1,368 s	0,123 s	0,051 s	1,542 s
GPU HR	22,036 s	10,669 s	0,053 s [5]	32,758 s
Szene: CB, 75.000 Patches, bis Konvergenz, 7. Iteration				
CPU HR	0,099 s	0,161 s	0,048 s	0,308 s
GPU HR	4,249 s	11,035 s	0,045 s [5]	15,329 s

Der erste Test mit der Cornellbox mit 1700 Patches ziegt, dass die HR-Methode auf der GPU um Schnitt dreimal langsamer ist als die CPU-Version. Sehr viel stärker wird der Unterschied in den darauffolgenden Tests. Das Phänomen, das z.B. bei der Methode Refine auftritt ist, dass nach der ersten Iteration der Großteil der Szene unterteilt ist und in den nachfolgenden Iterationen nur noch wenig verfeinert wird, spiegelt sich in sowohl in dem CPU- als auch dem GPU-Algorithmus wieder. Daher kann auch ein Rückgang der Berechnungszeiten verzeichnet werden. Auffällig bei der GPU-Version ist dennoch, dass ein Grundlevel an Bearbeitungszeit erhalten bleibt und nicht gegen Null sinkt. Der Test mit der 75.000 Patch großen Szene fällt mit durchschnittlich 40-facher Bearbeitungszeit sehr hoch aus. Um Aufschluss über die hohen Kosten zu erlangen ist es sinnvoll, einen Profiler das Projekt analysieren zu lassen um herauszufinden, welche Methodenaufrufe welche Kosten verursachen.

7.3 Profiling

Ein Profiler ermöglicht es einem Programmierer, seinen Quellcode auf Geschwindigkeitsmerkmale hin zu überprüfen. Bei der GPU-Programmierung ist es nicht so interessant, wie häufig die C++-Methoden aufgerufen werden, sondern welche Komponenten, in Bezug auf die Programmierung auf der Grafikkarte, wie stark benutzt werden und wie häufig. Mithilfe dieser Daten kann man anschließend die Methoden zur Berechnung von Daten optimieren und anpassen. Der verwendete Profiler in einer Testversion stammt von der Firma Automated AQ mit dem Namen AQTime in der Version 4.60. Die nachfolgende Tabelle zeigt die Methodenaufrufe, die am meisten Rechenzeit, bei einer fast 60 Sekunden langen Berechnung, benötigt haben. Die Angaben sind in Sekunden. Die Hit Counts zeigen die Häufigkeit der Methodenaufrufe. Die interessantesten sind die ersten fünf Methoden, glTexImage2D, glReadPixels, glBindTexture, wglMakeCurrent und glBegin.

[5] CPU-Version

Routine Name	Time	Time with Children	Shared Time	Hit Count
glTexImage2D	32.26	32.26	100	576647
glReadPixels	19.4	19.4	100	231473
glBindTexture	17.23	17.23	100	1275370
wglMakeCurrent	10.33	10.33	100	121863
glBegin	3.55	3.55	100	1768180
_heap_alloc	1.17	1.17	100	2578005
free	1	1	100	2326010
cgGLEnableTex...	0.82	6.22	13.15	576754
glVertex3f	0.74	0.74	100	7072284
glEnd	0.67	0.67	100	1768180
cgGLBindProgram	0.6	1.01	59.71	231580
glGetIntegerv	0.59	0.59	100	576764
<Root>	0.47	107.63	0.44	1
cgGLSetTex...	0.4	0.81	49.78	576754
cgGLDisableTex...	0.35	0.69	50.75	637572
glViewport	0.33	0.33	100	174737
operator new	0.32	2.07	15.27	2577928
cgiInitialize	0.32	0.32	100	4664286
...				
...				

- **glTexImage2D** - wird immer dann aufgerufen, wenn eine Textur mit Daten für die Berechnung auf der GPU erstellt wird.
- **glReadPixels** - liest die berechneten Daten aus der Render-Textur zurück in ein Array zum Postprocessing auf der CPU
- **glBindTexture** - bindet die angegebene Textur als aktuelles Texturziel / Texturobjekt
- **wglMakeCurrent** - setzt den angegebenen Renderkontext aktuell (wird von der Klasse RenderTexture verwendet)
- **glBegin** - Startkommando zur Eingabe von Eckpunkten einer Primitiven

7.4 Zusammenfassung der Ergebnisse

Die Umsetzung des Hierarchisches Radiosityalgorithmus auf der GPU bringt grundlegend einige Probleme mit sich. Zum einen ist die rekursive Bearbeitung der Kinder und Kindeskinder auf der GPU nur umsetzbar, wenn die Daten zurück in ein Array geschrieben werden, damit sie für den nächsten Rechenschritt ausgewertet werden können. Anderenfalls müssten enorme Datenmengen im Speicher der GPU jeder Zeit zur Verfügung stehen, die untereinander in Beziehung stehen. Ein weiteres Problem ergibt sich aus der

Menge der zu berechnenden Daten. Der schnellste Texturtyp zum Berechnen von Daten ist eine Textur mit $2^n x 2^n$ Maßen, alternativ lassen sich auch nxm- oder nxn-Texturen benutzen. Das Problem dabei ist, dass eine Textur in den meisten Fällen immer etwas größer ist, als die Anzahl der Daten, die berechnet werden müssen. Man berechne z.B. 13 Formfaktoren auf der GPU, so kann die Textur mit den Maßen $2x7$ oder $4x4$ gewählt werden. In beiden Fällen gibt es einmal 1 und das andere Mal 3 Fragmente, die es zu bearbeiten gilt, obwohl keine Daten vorliegen. Da eine Textur aber in der Größe des Viewports vorliegen muss, werden die restlichen Bereiche mit Nullen aufgefüllt. In der späteren Nachbearbeitung der Daten werden die überflüssigen Daten nicht berücktsichtigt, sie sind aber dennoch durch die Berechnung im Shader gelaufen und verursachen höhere Bearbeitungszeiten.
Die Schwierigkeit, die sich aus der Bearbeitung eines nicht uniformen oder nicht eingeschränkt adaptiven Gittermodells einer Szene auf der GPU ergibt, liegt in den verschieden großen Datenmengen, die pro Patch auftreten können, z.B. die Datenstruktur der Links eines Patches, die unterschiedlich stark benutzt wird. Daher können die Texturen für die Daten der Patches statisch gesetzt werden, die sie im Laufe der Berechnung vielleicht mehr Daten aufnehmen müssen. Eine Abhilfe würde hier eine Vorhersage, der Ausnutzung der Datenstrukturen eines Patches während der Berechnung bzw. eine feste Größe, für die maximale Anzahl an Daten, zu Beginn des Algorithmus, schaffen.
Die Ergebnisse des Profilings bestätigen die soeben angesprochenen Angaben. Die Methode, die am meisten Rechenzeit kostet, ist die `glTexImage2D` -Methode, welche Daten eines Arrays in eine Textur zu konvertiert. Das Auswerten der Daten der vorhergehenden Berechnung erfordert das Erstellen neuer Texturen. Im Schnitt werden für die Berechnung eines Patches sieben Texturen erstellt. Die Methode `glReadPixels` mit dem zweithöchsten Zeitfaktor liest die Daten der Texturen zurück. Auffallend bei der Methode ist, dass sie im Vergleich zur `glTexImage2D` -Methode nur halb so viele Aufrufe hat. Dies lässt sich darin begründen, dass einige Texturen ohne Zurücklesen in den nächsten Berechnungsschritt als Render-Textur einfliessen können. Die dritte Methode im Zeitranking ist `glBindTexture`.Sie benötigt vergleichsweise viel Zeit, da sie sehr häufig aufgerufen wird. Sie bindet eine Textur als aktuelles Texturobjekt, pro Patchberechnung werden im Schnitt 17 Texturen gebunden. Der Unterschied im Binden und Erstellen einer Textur liegt in der Verwendung der RenderTexture-Klasse, da diese nur einmal erstellt wird, aber mehrmals verwendet wird. Dazu werden die Methoden `BeginCapture` und `EndCapture` der Render-Texture Klasse benötigt, die die Daten des aktuellen OpenGL-Kontextes speichert. Nachdem die `EndCapture` -Methode aufgerufen wurde, kann mit `glBindTexture` die Render-Textur als Eingabetextur für einen Shader dienen. Daraus resultiert auch der vierthöchste Wert der Zeitmessung, der zur Methode `wglMakeCurrent` gehört. Diese Methode wird benötigt, wenn die Methoden `BeginCapture` und `EndCapture` aufgerufen werden. Die Methode `wglMakeCurrent` wechselt zu dem angegebenen Render-Kontext. Die RenderTexture-Klasse verwendet zwei Kontexte, zum einen der Kontext zum „Einlesen“ von Daten und zum anderem zum „Bereitstellen“ der aufgenommen Daten als Textur.

8 Fazit

Diese Diplomarbeit zeigt, dass eine Umsetzung von hierarchischem Radiosity auf der GPU generell möglich, wenn auch nicht unbedingt sinnvoll, ist. Die enorme Rechengeschwindigkeit der GPU kann nicht voll ausgeschöpft werden, da die Daten regelmässig zurückgelesen werden müssen, um sie auf der CPU auszuwerten. Dies entspricht nicht dem generellen Berechnungs- und Architekturmodell von heutigen Grafikkarten. Um diesem Modell zu entsprechen, müssten die Daten komplett im Speicher der Grafikkarte liegen. Zudem stehen die Daten untereinander in Beziehung. Dieses ist ebenfalls schwer umzusetzen, da bei der Berechnung von Daten auf der GPU im „general purpose“ -Fall davon ausgegangen wird, dass die Daten keine oder nur sehr wenige Beziehung zu anderen Daten aufweisen. Die Beziehungen der Daten untereinander könnten mit weiteren zusätzlichen Texturdaten ermöglicht werden, würden dann aber dazu führen, dass Redundanz innerhalb der Datenbestände der Texturen auftritt, die wiederum den begrenzten Speicher der Grafikkarte stark belastet. Einen Ansatz zur Berechnung von Radiosity ohne nennenswertes Rücklesen von Daten zur CPU wird von Greg Coombe in seinem Paper [CjHL04] und im Kapitel 5.2 vorgestellt. Diese Umsetzung von Radiosity auf der Grafikkarte erlaubt interaktive Raten, wobei die Szenendaten komplett im Speicher der Grafikkarte liegen.
Ein weiteres Problem bei der Umsetzung sind die starken Veränderungen der Szene und der Hierarchie, die bei der Durchführung auftreten. Ideal sind daher Datenstrukturen, wie z.B. die Klasse <vector> aus dem STL-Paket von C++. Auf der GPU gibt es lediglich Texturen fester Größe als Datenspeicher. Generell ist zu sagen, dass die GPU die Berechnungen sehr schnell durchführt, die enorme Geschwindigkeit aber nicht umgesetzt werden kann, da das Konzept des Erstellens von Texturen, Berechnen durch einen Shader und Zurücklesen der Daten, den Trumpf „Geschwindigkeitsvorteil“ gegenüber der CPU nicht ausspielen kann.
Sehr interessant wäre ein Test auf aktueller Hardware, die aber leider bis zum Abgabetermin der Diplomarbeit nicht verfügbar war.

9 Ausblick

Dieses Kapitel befasst sich mit möglichen Verbesserungen und Ergänzungen.

Um die Berechnungszeiten für das hierarchische Radiosity auf der Grafikkarte zu senken, kann der Algorithmus optimiert werden, indem versucht wird, gleiche Berechnungen auf mehreren unterschiedlichen Daten, z.B. Formfaktorberechnung der Kinder von mehrerern Patches, durchzuführen. Dies würde einen größere Ausnutzung der Texturen und weniger häufiges Zurücklesen der fertigen Daten bedeuten. Diese Optimierung müsste umfassend getestet werden, um festzustellen, ob sie eine Verbesserung der Verarbeitungszeit mit sich bringt.
Des Weiteren gibt es die Möglichkeit innerhalb eines Float-Wertes mehrere Werte eines kleineren Datentyps zu speichern. Die Shadersprache Cg unterstützt dieses im Gegensatz zu OpenGL von Hause aus. Ein Datentyp mit geringerer Präzision ist z.B. half, der mit 16 Bit eine halb so große Präzision besitzt wie der float-Datentyp. Eine Umsetzung dieser Idee müsste in OpenGL implementiert und ebenfalls umfassend getestet werden.
Eine weitere Möglichkeit zur Optimierung liegt darin, so viele Werte wie möglich zu berechnen und erst dann zurück zu lesen, wenn es erforderlich ist bzw. wenn eine maximale Anzahl an Berechnungen erreicht ist. Ein Beispiel dafür wäre das Auslesen der Gatherwerte nach dem Gathern.
Die Vervollständigung des Funktionsumfangs der PushPull-Methode auf der GPU müsste abgeschlossen werden. Die Methode konnte in den Tests nicht eingesetzt werden, da der Funktionsumfang noch stark eingeschränkt war. Des Weiteren könnte ein anderes Kriterium zur Unterteilung der Patches implementiert werden, wie z.B. das Lischinksi-Kriterium, welches eine exaktere Unterteilung anhand der Gradienten der Radiositywerte eines Patches beurteilt. Das Radiositysystem besitzt in der momentanen Version noch keine Sichtbarkeits- oder Verdeckungsrechnung, um Schatten in der Szene berechnen zu können. Um eine zu starke Unterteilung der Patches nicht erforderlich zu machen, könnte das System so verändert werden, dass es bilineares Filtern zwischen den Radiositywerten der Patches zulässt, um die Lichtverteilung weicher erscheinen zu lassen.
Sehr interessant wären auch weitere Tests mit anderen Plattformen bzw. Architekturen. Die Tests wurden auf einem AMD Athlon 64 3200+, einem Gigabyte Ram und einer Geforce 6800 GT mit 256 MB Speicher ausgeführt. Da der AGP-Port nur die Möglichkeit besitzt asynchron Daten zu transferieren, er kann mehr Daten zur Grafikkarte schicken als empfangen, wäre es interessant die Geschwindigkeitstests auf einem PCI-Express System durchzuführen. Dieses System arbeitet symmetrisch und könnte das Problem des Zurücklesens der Daten um einiges verringern. Weitere Tests mit aktueller Hardware, wie z.B. dem neuen Grafikchip G70, der mit 24 Grafikpipelines aufwartet, wäre ebenfalls aufschlussreich. Eine Grafikkarte mit dem neuen G70 Ultra Chipsatz erreicht eine um

211 Prozent gesteigerte Leistung[1] und übertrifft dabei eine SLI-Version der NV40 Ultra.

[1] http://www.tweakpc.de

10 Anhang & Quellcode

10.1 Pseudocodebeispiele des hierarchischen Radiosity Algorithmus

Pseudocodebeispiele des hierarchischen Radiosity Algorithmus aus Kapitel 3.4 auf Seite 19:

10.1.1 Refine

Algorithm 5 bool Refine(L)

```
e = L->e; s = L->s;
vis = L->vis; FF = L->F_es;
if OracleRefine(s, e, vis, FF) then
    which = OracleSubdivide(e, s);
    DeleteLink(L);
    if which == 'e' then
        Subdivide(e);
        for alle Soehne e->son von e do
            Link(e->son, s);
        end for
    else if which == 's' then
        Subdivide(s);
        for alle Soehne s->son von s do
            Link(e, s->son);
        end for
    end if
    return true;
end if
return false;
```

10.1.2 RefinementOrakel

Algorithm 6 bool OracleRefine(e, s, vis, FF)

```
if ΔB_s • FF • A_e < BFA_EPSILON then
    return false;
end if
if vis == totale Verdeckung then
    return false;
end if
if vis == partielle Verdeckung then
    return true;
end if
if ΔB_s • FF • A_e < BFA_MAX then
    return false;
end if
return true;
```

10.1.3 Subdivide

Algorithm 7 OracleSubdivide(e, s)

```
if (F_es >= F_se) && (e->area > ε) then
    return ('e');
end if
if (F_es < F_se) && (s->area > ε) then
    return ('s');
end if
return ('0');
```

10.1.4 Gather

Algorithm 8 Gather(p)

```
Quadnode *q, Link *L;
p->B_g = 0;
for each gather link L of p do
    p->B_g += p->ρ • L->F_pq • L->q->B_s;
end for
for each child node r of p do
    Gather(r);
end for
```

10.1.5 PushPull

Algorithm 9 PushPull(e, B_{down})

```
if e->son == NULL then
    B_up = e->E + e->B_g + B_down
else
    B_up = 0
    for alle Soehne e->son von e do
        B_tmp = PushPull (e->son, e->B_g + B_down)
        B_up += B_tmp * e->son->area/e->area
    end for
end if
e->B_g = 0
e->B_s = B_up
return B_up
```

10.2 Radiosity GPU-Algorithmen

10.2.1 Drei Shadermodell zum Lösen der FM-Matrix

Lösen des Gleichungssystem der FM-Methode auf der GPU mit drei Shadern.

FM-Shader 1

```
float4 main(float2 tex : TEX0,
uniform samplerRECT textureM,
uniform samplerRECT textureB) : COLOR
{
float3 _matrixM, _matrixB;
float4 result;

_matrixM = f3texRECT(textureM, tex);
_matrixB = f3texRECT(textureB, float2(tex.x, 0.0));

result = float4((_matrixM * _matrixB), 1.0);

return result;
}
```

FM-Shader 2

```
float4 main(float2 tex : TEX0,
uniform samplerRECT textureMB,
uniform float numPatches) : COLOR
{
float4 result;

for(int i = 0; i < numPatches; i++){
result += texRECT(textureMB, float2(i, tex.x));
}

return result;
}
```

FM-Shader 3

```
float4 main(float2 tex : TEX0,
uniform samplerRECT textureMB,
uniform samplerRECT textureB,
uniform samplerRECT textureE) : COLOR
{
```

```
float4 result, residue;

residue = texRECT(textureE, tex) - (texRECT(textureMB, tex));
result  = texRECT(textureB, tex) + residue;

return result;
}
```

10.2.2 Shader zum Berechnen von Formfaktoren

```
float main(float2 texCoords : TEX0,
uniform samplerRECT textureFF1,
uniform samplerRECT textureFF2,
uniform samplerRECT textureFF3,
uniform samplerRECT textureFF4) : COLOR
{

// Shader zum Berechnen von Formfaktoren der zweiten Formfaktorvereinfachung (Punkt zu Punkt Formfaktor)

float ff; // Rückgabewert als float, der Formfaktor

static const float Pi = 3.14159265f; // PI ist in CG nicht definiert

// Eingabetexturen
// Eine Spalte repräsentiert Daten einer Textur zur Berechnung eines Formfaktors
//
// patch p1 = sender
// patch p2 = receiver
//
// Matrix:
// p1.x | n1.x | p2.x | n2.x
// p1.y | n1.y | p2.y | n2.y
// p1.z | n1.z | p2.z | n2.z
//      | s    |      |
// Textur:
//                   FF1  | FF2  | FF3  | FF4


// Sender Mittelpunkt
float3 p1 = f3texRECT(textureFF1, float2(texCoords));
float4 n1temp = f4texRECT(textureFF2, float2(texCoords));

// Sendergröße
   float  s = n1temp.a;

// Empfänger Mittelpunkt
float3 n1 = n1temp.rgb;

// Sender Normale
float3 p2 = f3texRECT(textureFF3, float2(texCoords));

// Empfänger Normale
float3 n2 = f3texRECT(textureFF4, float2(texCoords));

//*****************************************************
// Formfaktor Berechnung
//*****************************************************

// Verbindungvektor erstellen
```

```
float3 vectA = p2 - p1;

// Abstand der Patches bestimmen
float lenA = length(vectA);

// Vector normalisieren
vectA = normalize(vectA);

float3 cosSender = dot(vectA, n1);
float3 cosReceiver = -dot(vectA, n2);

// Senderfläche vom Receiver abgewandt?
if ((cosSender.x <= 0.0) && (cosSender.y <= 0.0) && (cosSender.z <= 0.0))
{
ff = 0.0;
}

// Der umgekehrte Fall, Receiver vom Sender abgewandt?
else if ((cosReceiver.x <= 0.0) && (cosReceiver.y <= 0.0) && (cosReceiver.z <= 0.0))
{
ff = 0.0;
}

// Formfaktor berechnen, falls cosSender.xyz && cosReceiver.xyz > 0
else {
ff = s  * cosSender * cosReceiver / (Pi * lenA * lenA);

// Zweite FF-Vereinfachung! -> Clippen notwendig?
if (ff > 1.0) ff = 1.0;
}

return float(ff);
}
```

10.2.3 Shader als Refinement-Orakel

```
float3 refineOracle(float2 texCoords : TEX0,
uniform samplerRECT refineTexture0,
uniform samplerRECT refineTexture1,
uniform samplerRECT refineTexture2) : COLOR

{
// Tempwerte zum Auslesen der Daten aus den drei Eingabe-Texturen
float4 temp0, temp1;
float3 temp2;

// Berechnungswerte
float RootPatchID, LinkID, Ae, Epsilon; // Werte der ersten Textur  (rgba)
float FFse, Bs, Es, As; // Werte der zweiten Textur (rgba)
float subDepth, hierDepth, minArea; // Werte der dritten Textur (rgb)

float refine; // Ausgabewert refine

/*
Werte für refine:
0.0 : Keine Unterteilung
1.0 : Patch wird unterteilt
2.0 : Sender wird unterteilt
3.0 : maximale Hierarchietiefe erreicht
*/
```

```
// Auslesen der Texturen
temp0 = f4texRECT(refineTexture0, texCoords);
temp1 = f4texRECT(refineTexture1, texCoords);
temp2 = f3texRECT(refineTexture2, texCoords);

// Werte aus erster Textur zuweisen
RootPatchID = temp0.r; // rootPatch ID
LinkID = temp0.g; // Link ID, des aktuell bearbeiteten Links
As = temp0.b; // Fläche des Senderpatches
Epsilon = temp0.a; // Refine Epsilon

// Werte aus zweiter Textur zuweisen
FFse = temp1.r; // Formfaktor zwischen Sender und Empfänger
Bs = temp1.g; // Radiosity des Senderpatches
Es = temp1.b; // Emission des Senderpatches
Ae = temp1.a; // Fläche des Empfängerpatches

// Werte aus dritter Textur zuweisen
subDepth = temp2.r; // maximale Tiefe für Subdivide
hierDepth = temp2.g; // aktuelle Hierarchietiefe
minArea = temp2.b; // Minimalflächengröße, unterhalb derer kein Refine stattfindet

if ( hierDepth >= subDepth ) { // Test der Hierarchietiefe auf die maximale Tiefe für Subdivide
refine = 3.0; // Falls ja, tue nichts, aber merke an, dass Tiefe erreicht (3.0)
}
else {
if ( Bs * FFse * Ae > Epsilon ) { // Subdivide durchführen (BFA-Orakel)
if ( Es > 0 || Ae > As ) { // Empfänger oder Sender? Hier: ja -> Empfänger refine
if ( Ae > minArea ) { // Testen, ob Empfängerpatch nicht minArea unterschreitet
refine = 1.0; // Nein? Dann refine!
}
else {
refine = 0.0; // wird minArea unterschritten, tue nichts
}
}
else { // Empfänger oder Sender? Hier: ja -> Sender refine
if ( As > minArea ) { // Testen, ob Senderpatch nicht minArea unterschreitet
refine = 2.0; // Nein? Dann refine!
}
else {
refine = 0.0; // Patch unterschreitet minArea -> kein refine!
}
}
}
else{
refine = 0.0; // Kein refine, falls BFA-Kriterium nicht greift
}
}

return float3(RootPatchID, LinkID, refine); // Rückgabe (PatchID, LinkID des untersuchten Link, refine-Wert)

}
```

10.2.4 Shader für Mittelpunktberechnung

```
float3 subdivide(float2 texCoords : TEX0,
  uniform samplerRECT SubdivideTex1,
  uniform samplerRECT SubdivideTex2) : COLOR

{
// Berechnungswerte
```

```
float3 point0, point1, midPoint;

// **********************    r    g     b
// Werte in SubdivideTex1: P0.x, P0.y, P0.z
point0 = f3texRECT(SubdivideTex1, texCoords);

// **********************    r    g     b
// Werte in SubdivideTex2: P1.x, P1.y, P1.z
point1 = f3texRECT(SubdivideTex2, texCoords);


//Berechnen des Mittelpunktes
// Addition der beiden Punkte
midPoint = point0 + point1;

// Bestimmen der Mitte der Addition
midPoint = 0.5 * midPoint;

// Rückgabewert: Mittelpunkt der beiden Punkte
return midPoint;
}
```

10.2.5 Shader zum Berechnen von Gatherwerten

```
float3 main(float2 texCoords : TEX0,
  uniform samplerRECT gatherTexture0,
  uniform samplerRECT gatherTexture1) : COLOR
{
// Temp-Variable zum Auslesen der Daten aus der ersten Textur
float4 shoot_ff;

// Berechungs-Variablen
float3 rho, shoot, gather;
float ff;

// Auslesen der Berechnungsdaten aus der ersten Textur und speichern in shoot_ff
shoot_ff = f4texRECT(gatherTexture0, texCoords);

// Speichern der Werte, zugehörig
shoot = shoot_ff.rgb; // Shoot-Wert des Senderpatches
ff = shoot_ff.a; // FF zwischen Sender und Empfänger

// Auslesen der rho-Werte des Empfängers und speichern in rho
rho = f3texRECT(gatherTexture1, texCoords);

// eigentlicher Gathervorgang
gather = ( rho * ( shoot * ff ) ); // rho.rgb * (Bs.rgb * linkFF.r)

return float3(gather); // Ausgabe Gatherwert
}
```

10.2.6 Shader zur Addition von Gatherwerten

```
float3 main(float2 texCoords : TEX0,
  uniform samplerRECT gatherTexture) : COLOR

{
// Vorbereiten der Gather-Variablen
float3 gather1, gather2, gather3, gather4;

// Auslesen der Berechnungsdaten aus der Textur
```

```
// gather1 ist das erste Gather-Ergebnis, welches zum Ende mit den anderen
// Gather-Werten gather2, gather3, gather4 addiert wird
gather1 = f3texRECT(gatherTexture, texCoords);
gather2 = f3texRECT(gatherTexture, texCoords + float2(0,1) );
gather3 = f3texRECT(gatherTexture, texCoords + float2(1,0) );
gather4 = f3texRECT(gatherTexture, texCoords + float2(1,1) );

// Rückgabewert: Addition der vier Gather-Werte
return float3(gather1 + gather2 + gather3 + gather4);

}
```

10.2.7 Shader zur Addition der Gatherwerte (Push)

```
float3 main(float2 texCoords : TEX0,
  uniform samplerRECT pushTexture0,
  uniform samplerRECT pushTexture1) : COLOR

{
// Vorbereiten der PushPull-Variablen
float3 down, gather;

// Auslesen der Berechnungsdaten aus der Textur
// down-Werte stehen in der pushTexture0
// gather-Werte stehen in der pushTexture1
down = f3texRECT(pushTexture0, texCoords / 2);
gather = f3texRECT(pushTexture1, texCoords / 2);

// Rückgabewert: Addition der beiden Werte
return down + gather;
}
```

10.2.8 Shader zum Berechnen der Radiosity (PULL)

```
float3 main(float2 texCoords : TEX0,
  uniform samplerRECT pullTexture0,
  uniform samplerRECT pullTexture1,
  uniform samplerRECT pullTexture2) : COLOR

{
// pullTexture0 - RenderTexture (down+gather-Werte) der Kinder, sofern vorhanden(rgb)
// pullTexture1 - Flächengrößen der Kinder des Patches (r)
// pullTexture2 - Gatherwert, Flächengröße des aktuelles Patch und Wert für "Kind vorhanden" (rgb)

// Vorbereiten der Pull-Variablen
float3 temp; // Temp-Veriable zum Auslesen der Daten aus der Textur

float3 dg1, dg2, dg3, dg4; // Gatherwerte der Kinder
float a1, a2, a3, a4; // Flächengrößen der Kinderpatches

float hasChildren; // Kinder vorhanden?
float dg0, a0; // Gatherwert und Flächengröße des Patches

float Bs; // Rückgabewert, Radiosity das Patches

   // Auslesen der Berechnungsdaten aus der Textur
temp = f3texRECT(pullTexture2, texCoords);

dg0 = temp.r; // Gatherwert des Patches auslesen
a0 = temp.g; // Gatherwert des Patches auslesen
```

```
hasChildren = temp.b; // Gatherwert des Patches auslesen

if ( hasChildren == 0.0 ) {
Bs = dg0; // Keine Kinder vorhanden, so nehme direkt den Gatherwert als Radiosity
}
else { // Kinder vorhanden?
// Gatherwerte der Kinder auslesen
dg1 = f3texRECT(pullTexture0, 2*(texCoords));
dg2 = f3texRECT(pullTexture0, 2*(texCoords + float2(0,1)) );
dg3 = f3texRECT(pullTexture0, 2*(texCoords + float2(1,0)) );
dg4 = f3texRECT(pullTexture0, 2*(texCoords + float2(1,1)) );

// Flächengröße der Kinder auslesen
a1 = f3texRECT(pullTexture1, 2*(texCoords));
a2 = f3texRECT(pullTexture1, 2*(texCoords + float2(0,1)) );
a3 = f3texRECT(pullTexture1, 2*(texCoords + float2(1,0)) );
a4 = f3texRECT(pullTexture1, 2*(texCoords + float2(1,1)) );

Bs = ( (dg1 * a1) + (dg1 * a1) + (dg1 * a1) + (dg1 * a1) ) / a0;
}

return Bs;
}
```

Literaturverzeichnis

[BDS04] Anja Borsdorf, Carsten Dachsbacher, and Marc Stamminger. Progressive radiosity auf programmierbarer grafikhardware. 2004.

[CGI86] Michael Cohen, Donald P. Greenberg, and Dave S. Immel. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics and Applications*, pages 26–35, 1986.

[CHH03] Nathan A. Carr, Jesse D. Hall, and John C. Hart. Gpu algorihms for radiosity and subsurface scattering. *The Eurographics Association 2003*, 2003.

[CjHL04] Greg Coombe, Mark j. Harris, and Anselmo Lastra. Radiosity on graphics hardware. *Graphics Interfaces 2004*, 2004.

[CW93] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Publishers, 1993.

[ea04] Randima Randy Fernando et al. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. 2004.

[Fec96] Frank Fechtner. Konzeption und implementation eines radiosity-kerns. 1996.

[FK03] Randima Fernando and Mark J-. Kilgard. *The Cg Tutorial*. Addison-Wesley, 2003.

[Gla95] A. S. Glassner. *Principles of digital Image Synthesis Vol. 1 and Vol. 2*. Morgan Kaufmann Publishers, 1995.

[HSA91] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Proceedings of SIGGRAPH*, 1991.

[Kil96] Mark J. Kilgard. The opengl utility toolkit (glut) programming interface. PDF file: glut-3.spec.pdf, 1996. API Version 3.

[KW03] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. 2003.

[Mül01a] Stefan Müller. Einführung in die photorealistische computergraphik. PDF file: 01_Einfuehrung_2.pdf, 2001. Vorlesung Photorealistische Computergrafik.

[Mül01b] Stefan Müller. Hierarchisches radiosity. PDF file: 14_HierRad_2.pdf, 2001. Vorlesung Photorealistische Computergrafik.

[Mül01c] Stefan Müller. Orakel. PDF file: 13_Orakel_2.pdf, 2001. Vorlesung Photorealistische Computergrafik.

[Mül01d] Stefan Müller. Strahlungsaustausch zwischen 2 flächen. PDF file: 04_strahlungsaustausch_2.pdf, 2001. Vorlesung Photorealistische Computergrafik.

[nVI04] nVIDIA. Cg toolkit - user's manual. PDF file: Cg_Toolkit.pdf, 2004. Release 1.2.

[nVI05] nVIDIA. nvidia gpu programming guide. PDF file: GPU_ Programming_Guide.pdf, 2005. Version 2.4.0.

[SP94] Francois X. Sillion and Claude Puech. *Radiosity & Global Illumination*. Morgan Kaufmann Publishers, 1994.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000.