UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Hierarchisches Radiosity unter Berücksichtigung von Texturen

Studienarbeit

im Studiengang Computervisualistik

vorgelegt von

Sören Kewenig

Betreuer: Dipl.-Inform Oliver Abert
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Dezember 2006

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Studienarbeit von Sören Kewenig (Matr.-Nr. 202120648)

Thema: **Hierarchisches Radiosity unter Berücksichtigung von Texturen**

Nachdem bereits 1988 Progressive Refinement von Cohen et al. entwickelt wurde, waren die ersten Schritte in Richtung Globale Beleuchtung für das Scanline-Rendering getan. In den folgenden Jahren wurden verschiedene weitere Verfahren entwickelt um eine möglichst realitätsgetreue Beleuchtung einer virtuellen 3D-Welt zu erreichen, so z.B. auch das bis heute aktuelle hierarchische Radiosity, das von Hanrahan et al. bereits 1991 vorgestellt wurde.

Das Ziel dieser Studienarbeit ist die Erarbeitung und Implementierung eines Radiosity-Systems mit hierarchischer Beschleunigung. Der Schwerpunkt bei der Entwicklung liegt dabei nicht primär auf der Geschwindigkeit des Systems, sondern auf der korrekten Umsetzung sowie der möglichst realistischen Beleuchtung einer 3D-Szene. So soll das System nicht nur eine weit reichend korrekte Beleuchtung, sondern auch eine realistische Nicht-Beleuchtung in Form von Schatten darstellen können.

Als Besonderheit soll das bis dahin entwickelte Radiosity-System von einfachen Oberflächen auf Texturen übertragen werden, so dass in den Lichtaustausch auch die Beschaffenheit von etwaigen Texturen einberechnet werden.

Die Umsetzung soll unter Windows mit OpenGL und der Programmiersprache C++ geschehen. Die Schwerpunkte sind dem nach wie folgt:

1. Wissensaufbau in Bezug auf Radiosity-Systeme
2. Implementierung eines hierarichischen Radiosity-Systems für Dreiecke/Quads
3. Erweiterung des vorhandenen Systems um Schattenwurf
4. Entwicklung von texturbasierten Radiosity sowie Einbindung in das System
5. Dokumentation des Systems und der erzielten Ergebnisse

Koblenz, den 20.06.2006

Betreuer: Dipl.-Inform. Oliver Abert

– Prof. Dr. Stefan Müller –

Erklärung

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

Ort, Datum Unterschrift

Danksagung

Dieser Abschnitt ist meiner Freundin Elke gewidmet, die sich trotz ihrer Vordiplomsprüfungen, die Zeit genommen hat, diese Studienarbeit in einem nächtlichen Marathon Korrektur zu lesen.

Sören Kewenig

Inhaltsverzeichnis

1. Motivation

Zur Erzeugung von künstlichen Bildern, die den Eindruck der Realität vermitteln sollen, reicht die realistische Modellierung von Szenen oder Objekten alleine nicht aus. Damit eine Szene wirklich realistisch wirkt, ist die wichtigste Komponente das Licht. Nicht nur die Wahl der Lichtquelle, also Punktlicht oder Spotlicht, oder die dazugehörigen Parameter sind entscheidend, sondern viel mehr wie ein Rendering-System Licht simuliert. Die Qualität der Simulation entscheidet letztlich über die Qualität des Endresultats.
So können die einfachsten Objekte nur durch eine realistische Beleuchtung ästhetisch aussehen, wie z.B. die Cornell Box, wohingegen Objekte die sehr reich an Details sind, aber falsch oder schlecht beleuchtet werden, irgendwie fehl am Platz wirken. Das Ziel eines jeden Rendering-Systems sollte es daher sein, ein möglichst realistisches Beleuchtungsmodell umzusetzen.

1.1 Die Rendering Equation

Alle Beleuchtungsmodelle die in der Computergraphik existieren, lassen sich auf die Rendering Equation zurückführen. Einfache Modelle, wie z.B. Gouraud Shading, beachten für die Beleuchtung von Objekten nur die Menge des einfallenden Lichts, dass von einer bestimmten Richtung einfällt, und berechnet daraus die Helligkeit für einen Punkt. Die Reflektion eines Teils des eingefallenen Lichts wird dabei nicht beachtet. Diese Systeme werden auch als lokale Beleuchtungssytemen bezeichnet.
Für eine wirklich realistische Beleuchtung muss beachtet werden, wieviel Licht von einem Flächenelement[1] dA_e in eine beliebige Richtung ω_o ausgestrahlt wird. Da Licht nicht einfach als Licht betrachtet wird, sondern in mehreren Größen, wie Leuchtdichte, Beleuchtungsstärke, etc. zerlegt werden kann, wird in dem folgenden Abschnitt von der Leuchtdichte gesprochen.
Der Anteil der Leuchtdichte die von einem Flächenelement dA_e ausgesendet wird, ergibt sich dabei zum Einen über die Leuchtdichte L_e, die vom Flächenelement dA_e selbst ausgesendet wird, die sogenannte Eigenemission. Der andere Teil errechnet sich über die Leuchtdichte L_i, die auf das Flächenelement dA_e von allen beliebigen Winkeln ω_i einfällt und in die Richtung ω_i reflektiert wird.

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

Für alle Einfallswinkel kann dann über die BRDF (Bidirectional Reflectance Distribution Function) f_r die ausgestrahlte Leuchtdichte ins Verhältnis gesetzt werden zur Beleuchtungsstärke. Die BRDF kann dabei die Leuchtdichte für jeden Einfalls- bzw. Ausfallswinkel bestimmen (4D), wobei dann für jeden Einfallswinkel ω_i geprüft wird wie hoch die Leuchtdichte ist, die in alle Winkel ω_o reflektiert wird.

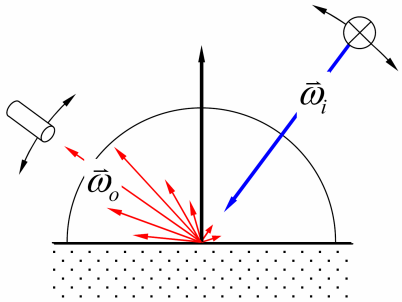


Abbildung 1: Licht, das über einen Einfallswinkel einfällt, kann in mehrere Ausfallswinkel reflektiert werden.[2]

[1] Ein Flächenelement entspricht einem Punkt, besitzt aber eine Normale, da er ja nur Teil einer Fläche ist.
[2] Das Bild wurde aus Quelle [Müll05] kopiert.

Wenn die Auswertung zusätzlich noch für jede Wellenlänge oder sogar noch für jeden Punkt auf der Oberfläche erfolgen soll, steigt die Komplexität auf 5D bzw. 7D an. Da eine Auswertung der obigen von Kajiya [Kaji86] vorgestellten Gleichung sehr zeitaufwendig wäre, ist mit unterschiedlichen Ansätzen versucht worden die BRDF anzunähern. So sind einfachere Modelle wie z.B. Cook-Torance [Cook81] oder Phong [Phon73] entstanden.
Bekannte Lösungsansätze für komplexere Modelle sind zum Einen das Raytracing in Verbindung mit Photon-Mapping oder die hier vorgestellten Radiosity-Systeme.

2. Radiosity

Das Radiosity-Verfahren wurde im Jahre 1983 von Cindy Goral et al. entwickelt. Es besitzt seine Ursprünge in der Thermodynamik, wo jenes verwendet wurde, um den Austausch von Wärmestrahlung zu berechnen.
Die mathematischen Grundlagen des heutigen Radiosity-Algorithmus wurden bereits in den 20er Jahren von Yamauti und Buckley erarbeitet, und diese Ideen schließlich im Jahre 1934 von Higbie in dem Buch „Lighting Calculations“ zu einem ersten Algorithmus weiterverarbeitet. Trotz des immensen Aufwands dieses Verfahrens begaben sich Moon und Spencer 1940 an ein Experiment, in dem sie einen leeren Raum in Abschnitte einteilten und mit der Arbeit von Higbie die Farbwerte für diese Flächen berechneten. Dies geschah damals in mühsamer Handarbeit, indem die Helligkeit dieser Flächen berechnet und anschließend einzelne Papierstücke ausgeschnitten und aneinandergefügt wurden.
Seit der Einführung des Radiosity-Verfahrens in die Computergraphik sind immer wieder neue Varianten des Algorithmus entwickelt worden (Kapitel 5).

2.1 Beleuchtungsmodell und Herleitung

Auch die Radiosity-Verfahren stützen sich auf der oben vorgestellten Gleichung, wobei zwei Randbedingungen eine Umsetzung erst ermöglichen. Die „Constant Radiosity Assumption“ ist die erste Randbedingung und besagt, dass die Radiosity auf einem Flächenelement dA_e als konstant angenommen werden kann. In der zweiten Annahme wird davon ausgegangen, dass die Umgebung, also alle Flächen der Szene, vollständig diffus ist. Das Wort „diffus“ drückt in diesem Kontext aus, dass die Leuchtdichte einer Fläche unter jedem Winkel konstant ist und somit gleich hell erscheint. Eine Fläche die diese Eigenschaft besitzt nennt man auch einen diffusen Lambert-Strahler, da diese Fläche in alle Richtungen die gleiche Leuchtdichte ausstrahlt.
Da also angenommen wird, dass die Szene nur aus solchen Lambert-Strahlern besteht, kann die ursprüngliche BRDF als konstant angenommen werden. Das Verhältnis des einfallenden Lichtstroms zum ausfallenden Lichtstrom braucht dann nur noch über den Reflektionsfaktor ρ des Elements dA_e bestimmt werden. Dieser kann entweder als spektraler Reflektionsgrad für Wellenlängen oder als einfacher Reflektionsgrad betrachtet werden. Der Wertebereich liegt in beiden Fällen zwischen null und eins.

$$f_r(d\vec{\omega}_i, d\vec{\omega}_o) = \frac{\rho}{\pi}, \text{ mit } \rho = \frac{\Phi_o}{\Phi_i}$$

Für das Verhältnis von einfallendem zu ausfallendem Lichtstrom beschreibt der untere Teil gerade die Beleuchtungsstärke E. Es ergibt sich somit für ρ:

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

Wird die BRDF nun als konstant angenommen, kann diese vor das Integral der Rendering Equation (Kapitel 1) gezogen werden und somit ergibt sich:

$$L_o = f_{r,d} \cdot \int_{2\pi} L_i(d\vec{\omega}_i) \cdot \cos\theta_i \cdot d\omega_i \Leftrightarrow L_o = f_{r,d} \cdot E$$

Wird nun das Ergebnis von ρ aus der oberen Gleichung nach E aufgelöst und in die untere Gleichung eingesetzt, kann man die Leuchtdichte rausstreichen und man erhält für ρ:

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

Dieses eingesetzt in die Rendering Equation an Stelle der BRDF, erhält man einen konstanten Faktor, der nur noch von dem definierten Reflektionsgrad ρ des Materials einer Fläche dA_e abhängt. Dieser kann, da er konstant ist, vor das Integral gezogen werden.

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

Nach dem Einsetzen von $\Phi = L \cdot A \cdot \pi$ und einer Multiplikation mit π ergibt sich die Radiosity-Gleichung, wobei als Sendergröße die Radiosity B und als Empfängergröße die Beleuchtungsstärke E gelten. Im letzten Schritt wird das Integral diskretisiert, da die Strahlung in einer endlichen Szene immer nur von endlich vielen Flächen n versendet werden kann. Der Anteil der Strahlung B, die von einem Sender ausgesendet wird und beim Empfänger ankommt, wird dabei von dem Formfaktor F_{es} beschrieben. Näheres dazu im folgenden Kapitel.

$$B(dA_e) = E(dA_e) + \rho(dA_e) \int_{2\pi} L_i(dA_e, d\vec{\omega}_i) \cdot \cos\theta_i \cdot d\omega_i$$

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

3. Der Formfaktor

Damit ein Lichtaustausch zwischen zwei Flächen stattfinden kann, muss zuerst ermittelt werden, wie hoch der Anteil der vom Sender versendeten Strahlung ist, der beim Empfänger ankommt. Um dies zu bestimmen, enthält die eben vorgestellte Radiosity-Gleichung mehrere Faktoren. Der wichtigste dieser Faktoren ist der Formfaktor, auf dem das gesamte Radiosity-Verfahren aufbaut. Dieser Formfaktor besteht aus einem Doppelintegral über die Sender- sowie die Empfängerfläche und enthält alle geometrischen Größen von der Entfernung d zweier Flächen, über die Flächengrößen der Senderfläche A_s bzw. Empfängerfläche A_e, bis hin zu der Stellung und der sich damit ergebenden sichtbaren Fläche beider Strahlungspartner. Eine korrekte Bestimmung des Formfaktor würde bedeuten, dass für jeden Punkt, z.B. auf der Empfängerfläche, die Strahlungsmenge bestimmt werden muss, die von jedem Punkt der Senderfläche ausgesendet wird. Es müsste also für jeden „Strahl", über den Strahlung ausgetauscht wird, der innere Teil des Integrals ausgewertet werden. Da diese analytische Lösung des Formfaktors aufgrund des Doppelintegrals zwar möglich, aber sehr aufwendig, ist wird in der Praxis eine vereinfachte Variante genutzt.

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

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

3.1 Erste Formfaktor-Vereinfachung

Damit Radiosity-Simulationen in der Computergraphik genutzt werden können, muss ein Kompromiss zwischen der Genauigkeit und der Effizienz eines Systems getroffen werden. Dieser Kompromiss ist die erste Formfaktor-Vereinfachung, deren Fehler vorhanden aber annehmbar ist.
Hierbei wird von der Annahme ausgegangen, dass einer der beiden Strahlungspartner, in diesem Fall die Senderfläche, sehr viel kleiner ist als die Empfängerfläche. Die Empfängerfläche wird somit zu einem Flächenelement reduziert, so dass die Strahlung, die von diesem Element ausgeht, als konstant angesehen werden kann. Durch diese Annahme ist es möglich, den nun konstanten Faktor A_s vor das verbleibende Intergral zu ziehen.

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

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

Da aber auch die Bildung eines Integrals über eine Fläche sehr aufwendig ist, müssen auch hier Abstriche hingenommen werden. Denn ausgehend vom obigen Beispiel muss bei der Diskretisierung des verbleibenden Integrals die Anzahl der Punkte auf der Senderfläche festgelegt werden, für die die obige Gleichung gelöst werden soll. Damit dies nicht nötig ist, greift an dieser Stelle die „Constant-Radiosity-Assumption". Denn dadurch, dass die Radiosity pro Flächenelement als konstant angenommen wird, reicht es, den Formfaktor zu einem Punkt der Senderfläche zu bestimmen. Wollte man aber das obige Integral korrekt auswerten, müsste man sehr viele infinitesimal kleine Flächen erzeugen, die dann jeweils eine andere Radiosity besitzen dürften.

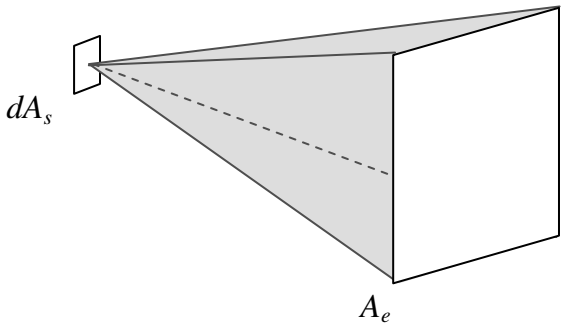


Abbildung 2: Der „Raumwinkel“ zwischen dem Sender dA_s und dem Empfänger A_e.

Damit der Fehler trotz dieser Verfeinfachung minimal bleibt berechnet man den Formfaktor, wobei man die Fläche mit dem geringeren Flächeninhalt als infinitesimal klein annimmt, und dann den gewünschten Formfaktor über die Reziprozität ermittelt.

3.2 Zweite Formfaktor-Vereinfachung

Kommt es bei der Bestimmung des Formfaktors eher auf die Geschwindigkeit des Systems und nicht so auf die Genauigkeit an, kann die zweite Formfaktor-Vereinfachung verwendet werden. Bei dieser geht man noch einen Schritt weiter und nimmt an, dass sowohl die Sender-, als auch die Empfängerfläche infinitesimal klein sind. Durch diese Annahme ist es möglich, auch das letzte Integral zu beseitigen, da nun nur noch der Strahlungsaustausch zwischen zwei Punkten berechnet wird, und somit den Berechnungsaufwand auf ein Minimum reduziert wird.
Diese Formfaktor-Vereinfachung sollte allerdings nur für Flächen verwendet werden, deren Entfernung-zu-Fläche-Wert für parallele Flächen bei ca. 1.5 liegt. Ist der d/A-Wert kleiner, steigt der Fehler sehr schnell an und so kann, bei einem zu geringem Abstand zwischen Sender und Empfänger, der Fehler so groß werden, dass der Formfaktor den normalen Wertebereich sogar überschreitet. Ohne ein Clipping des Formfaktors würde es dann unweigerlich zu einer „Lichtbombe“ kommen, da der Empfänger mehr Strahlung aufnehmen würde als der Sender ausstrahlt.

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

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

Mit einer weiteren Umformung können aus der Formfaktor-Vereinfachung sogar die aufwendigen Cosinusberechnungen eliminiert und durch Skalarprodukte ersetzt werden.

$$d \cdot \cos\theta_s = \vec{d} \circ \vec{n}_s \text{ bzw. } d \cdot \cos\theta_e = \vec{d} \circ \vec{n}_e$$

$$F_{dA_s dA_e} = \frac{(\vec{d} \circ \vec{n}_s) \cdot (-\vec{d} \circ \vec{n}_e)}{\pi \cdot d^4} \cdot \Delta A_e$$

3.3 Die Formfaktor-Eigenschaften

Der Formfaktor selbst ist eine geometrische Größe, weshalb er dimensionslos ist. Der Wertebereich reicht dabei von null, es wird keine Strahlung ausgetauscht, bis eins, es wird alle Strahlung ausge-

tauscht. Zu den weiteren Eigenschaften des Formfaktors gehört die Reziprozität, die sich durch die Multiplikation der Flächenverhältnisse A_e/A_s oder A_s/A_e ergibt:

$$F_{A_e dA_s} = \frac{A_s}{A_e} \cdot F_{dA_s A_e}, \text{ bzw. } F_{A_s dA_e} = \frac{A_e}{A_s} \cdot F_{dA_e A_s}$$

Betrachtet man den Formfaktor in Verbindung mit dem Visibilitätsterm, können drei Fälle auftreten in denen der Formfaktor null wird. Soll der Formfaktor zu sich selbst berechnet werden, d.h. eine Selbstbeleuchtung stattfinden, wie es theoretisch bei gewölbten Flächen passieren könnte, ist das Ergebnis als null definiert, da eine Selbstbeleuchtung nicht vorgesehen ist. Im zweiten Fall wird der Formfaktor null, falls Sender- und Empfängerfläche unendlich weit voneinander entfernt sind, und somit der Abstand *d* wesentlich größer ist als der Term des Zählers. Im letzten Fall muss ein Objekt Sender und Empfänger vollständig verdecken, so dass der Visibilitätsterm null ergibt.

3.4 Der Formfaktor-Fehler

Abhängig von der Stellung zwischen Sender und Empfänger können je nach Formfaktor-Vereinfachung unterschiedlich große Fehler entstehen. Wie man unten stehender Graphik entnehmen kann, verhalten sich die Ergebnisse der ersten als auch der zweiten Formfaktor-Vereinfachung nur bei größeren Abstand-zu-Fläche-Werten wie die Ergebnisse der Ausgangsgleichung. Ein System das auf möglichst hohe Genauigkeit und Performanz abzielt, wird wohl abhängig von dem *d*/*A*-Wert entscheiden, welche Berechnung genutzt wird.
Für alle anderen Systeme könnte eine Hybride zwischen der ersten und zweiten Formfaktor-Vereinfachung gewählt werden. Für das in dieser Studienarbeit entwickelte System wurde ausschließlich die erste Formfaktor-Vereinfachung verwendet, da eine Optimierung zu einer Hybride aus Zeitgründen nicht mehr realisierbar war.

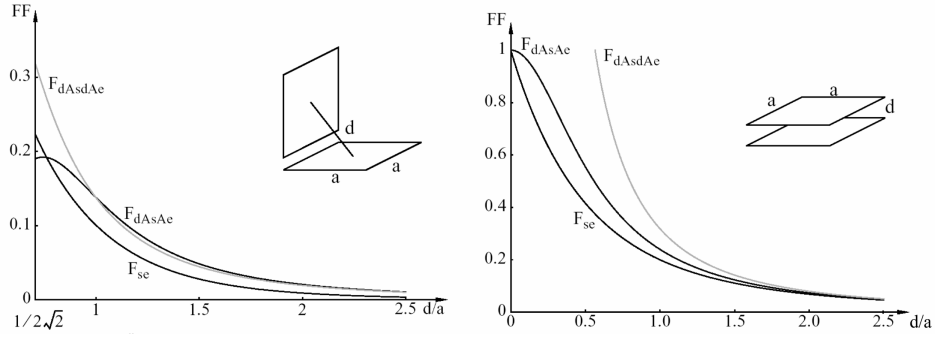


Abbildung 3: Der Fehler für die Formfaktorvereinfachungen für parallele und senkrechte Flächen.[3]

3.5 Die Formfaktor-Bestimmung

In diesem Abschnitt werden kurz einige Verfahren zur Formfaktor-Bestimmung vorgestellt. Im Detail werden zwei Verfahren behandelt: das Nusselt-Analogon und die Prisma-Lösung, wobei das Nusselt-Analogon eher als Ideengeber anzusehen ist. Eine weitere verbreitete Möglichkeit zur Berechnung des Formfaktors ist das aufwendigere Hemi-Cube-Verfahren, das von Cohen et. al. [Cohe85] vorgestellt wurde, hier aber nicht näher erläutert wird. Alle hier vorgestellten Verfahren basieren auf der ersten Formfaktor-Vereinfachung. Für die beiden anderen Bestimmungen ist eine nährere Erläuterung nicht

[3] Das Bild wurde aus Quelle [Müll05] kopiert.

nötig, da für die Ausgangsgleichung nur eine Lösung [Schr93] existiert, die in Simulationen auf Grund ihrer Komplexität nicht eingesetzt wird, sondern nur als Referenzwert dient. Für die zweite Formfaktor-Vereinfachung ist die Vorstellung eines Verfahrens ebenfalls nicht nötig, da die obige Gleichung aus Kapitel 3.2 zur direkten Bestimmung verwendet werden kann.

3.5.1 Das Nusselt-Analogon

Damit der Formfaktor zwischen einem Flächelement *e* und einer Fläche A_s bestimmt werden kann, muss der gesamte verfügbare Raumwinkel des Elements *e*, welches gerade die Beleuchtungshemisphäre ist, ins Verhältnis gesetzt werden zu dem von der Fläche A_s eingenommenen Raumwinkel. Eine Lösung hierfür wurde bereits 1928 von Nusselt [Nuss28] gefunden. Die Lösung kann nun für das Nusselt-Analogon verwendet werden um den Formfaktor zu bestimmen.

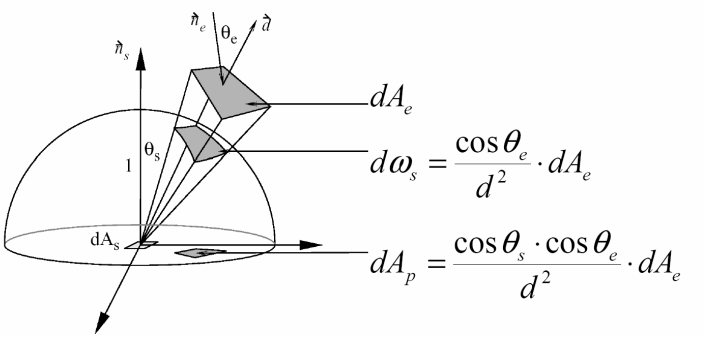


Abbildung 4: Der Formfaktor kann über mehrere Projektionen ins Verhältnis zum Einheitskreis gesetzt werden.[4]

Zur Bestimmung des Formfaktors wird zuerst die Fläche A_s auf die Einheitskugel des Elements *e* projiziert. Diese Projektion wird dann ein weiteres Mal, aber diesmal orthogonal, auf die Grundfläche der Hemisphäre projeziert. Das Verhältnis dieser Projektion zur Fläche des Einheitskreises ist dann der gesuchte Formfaktor.

3.5.2 Die „Prisma"-Lösung

Die „Prisma"-Lösung ist von dem eben vorgestellten Nusselt-Analogon abgeleitet. Bei dieser Lösung wurde die Projektion der Fläche ins Verhältnis zur Fläche des Einheitskreises gesetzt. In diesem Verfahren wird der Raumwinkel allerdings über die Winkel zwischen den Vektoren vom Element dA_e zu den Eckpunkten einer Fläche A_s, errechnet. Diese Radianz-Winkel werden aufaddiert und dann ins Verhältnis zu dem gesamt zur Verfügung stehenden Winkel (2π) eines Kreises gesetzt.

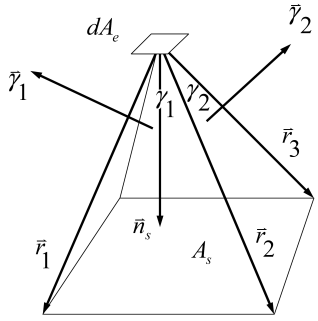


Abbildung 5: Das Bild zeigt die geometrische Berechnung der „Prisma"-Lösung.[4]

[4] Das Bild wurde aus Quelle [Müll05] kopiert.

Um mit Hilfe der „Prisma“-Lösung einen Formfaktor zu bestimmen, werden von dem Mittelpunkt der infinitesimal kleinen Fläche, hier der Empfänger, ein Vektor r_i zu jedem Eckpunkt der anderen Fläche aufgestellt. Diese Vektoren werden dann normiert damit bei dem gleich folgenden Skalarprodukt das Ergebnis nicht verfälscht wird.
Über das Kreuzprodukt zwischen zwei benachbarten, der eben aufgestellten, Vektoren wird dann eine Senkrechte $\gamma_{i/\,i+1}$ auf der von r_i und r_{i+1} aufgespannten Fläche erstellt, wobei auch diese normiert wird. Im Anschluss braucht nur noch der Winkel zwischen den beiden benachbarten Vektoren in Radianz gebildet und dieser mit der Senkrechten multipliziert werden. Jede errechnete Normale wird dann auf die Normale des Senders projiziert, aufsummiert und letztlich mit $-1/2\pi$ multipliziert.

$$F_{dA_e A_s} = -\frac{1}{2\pi}\sum_i \vec{\gamma}_{i/i+1} \circ \vec{n}_e$$

Die Lösung kann noch in der Hinsicht optimiert werden, dass die Bildung des letzten Skalarproduktes nicht für jede Normale durchgeführt werden muss. Es reicht aus, alle Normalen zu addieren und im Anschluss das Skalarprodukt zu bilden, da es mathematisch keinen Unterschied macht, ob erst für alle Vektoren die Länge einzeln bestimmt wird, oder für einen Vektor, der alle anderen vereint. Mit dieser Änderung können für ein Quad drei Skalarprodukte eingespart werden [Kres97].

Zur Berechnung des Formfaktors wurde in dieser Studienarbeit die „Prisma“-Lösung verwendet, da es ein sehr gut umzusetzendes Verfahren darstellt, bei dem keine versteckten Probleme auftreten können. Alternative Verfahren wie der Hemi-Cube würden sich nur lohnen, wenn dieser ein besseres Ergebnis liefern würde. Da man sich beim Hemi-Cube aber für ein gutes oder aber ein schnelles Resultat entscheiden muss, wäre auch hier ein Kompromiss nötig.

Codeauszug:

```
Vector normal   = receiver->getNormal();
Vector midpoint = receiver->getMidpoint();

short  numberOfVertices = emitter->getNumberOfVertices();

for (short i = 0; i < numberOfVertices; i++)
{
       //Two Vectors to the Vertices of one side
       a = emitter->getVertex(i) - midpoint;
       b = emitter->getVertex((i + 1) % numberOfVertices) - midpoint;

       //Normalize them for further calculations
       a.normalize();
       b.normalize();

       //Cross-prduct between a and b to get side-normal
       gamma = a.cross(b);

       //The length of the gamma vector
       length = acos(a.dot(b));

       //Normalize gamma to the length 1
       gamma.normalize();

       //Setting the angle between a and b as length for gamma
       gamma = gamma * length;

       //Create the sum of all gamma
       sum += gamma;
}
//Determine the formfactor and return to the caller
ff = -(1 / (2 * PI)) * sum.dot(normal);

return ff;
```

4. Visibilität

Ein wesentlich komplexerer Teil des Radiosity-Systems ist die Sichtbarkeitsprüfung. Wo bei Raytracing-Algorithmen schöne Schatten „geschenkt“ sind, können diese bei direkter Beleuchtung durch die Strahlverfolgung zur Lichtquelle und bei indirekter Beleuchtung durch das Verschießen von Photonen erhalten werden, ist das „Finden“ von Schatten für Radiosity-Systeme hingegen sehr aufwendig. So muss für jedes finite Element eine Reihe von Sichtbarkeitstests durchgeführt werden, um dann später durch das Orakel zu entscheiden, ob dieses Element voll, partiell oder gar nicht verschattet ist. Werden diese Verfahren jedoch nicht akkurat genug durchgeführt, werden manche Objekte gar nicht verschattet, wohingegen eine zu genaue Durchführung sehr viel Performanz kostet.

Für hierarchische Radiosity-Systeme exisitieren mehrere Stufen von Visibilitätsstests: Zuerst muss für ein hierarchisches Radiosity-System entschieden werden, ob zwei Flächen einander zu- oder abgewandt sind und somit ein Link zwischen diesen beiden erstellt wird oder nicht. Ist dies der Fall, muss für diesen Link gespeichert werden, wie gut oder schlecht sich beide Flächen „sehen“ können. Abhängig von der hier gewählten Methode und dem Grad der Genauigkeit hängt ab, ob neben großen Objekten später auch kleine Objekte verschattet werden. Im Abschnitt 4.3 werden hierfür zwei Verfahren erläutert: das „Magische Quadrat“, sowie ein vom Raytracing abgeleitetes Verfahren.
Letzlich muss dann für die Formfaktor-Bestimmung geprüft werden, ob zwei Flächen einander schneiden und so ein Support-Plane-Split durchgeführt werden muss.

4.1 Der Zugewandt/Abgewandt-Test

Der erste Test um zu entscheiden, ob ein Link zwischen zwei Flächen erstellt werden soll oder nicht, ist der Zugewandt/Abgewand-Test, der entscheidet, ob zwei Flächen einander zugewandt sind. Zuerst werden hierfür die Mittelpunkte des Senders und Empfängers verbunden.

$$\vec{d} = M_e - M_s$$

Nun wird das Skalarprodukt zwischen einer Flächennormalen und der eben erstellten Verbindung d berechnet. Ist das Ergebnis einmal > 0 und einmal < 0 kann eine Verbindung zwischen beiden Flächen etabliert werden. Der Grund warum ein Skalarprodukt als Kriterium verwendet wird ist, dass die Verbindung d ein Vektor des R^3 ist und dieser Vektor neben seiner Länge auch eine Richtung hat. Sind zwei Flächen nun zugewandt und parallel, zeigt die Normale einer Fläche genau in die entgegengesetzte Richtung der anderen Flächennormalen. Die Verbindung d zeigt für den Zugewandt-Fall nun in dieselbe Richtung der Normalen n_s und in die entgegengesetzte Richtung von n_e. Darum liefert das Skalarprodukt zwischen d und den Flächennormalen einmal ein negatives und einmal ein positives Ergebnis.
In allen anderen Fällen, also bei zwei negativen oder zwei positiven Ergebnissen, sind die Flächen einander abgewandt, und es wird keine Verbindung erstellt. Abbildung 6 veranschaulicht die Situation.

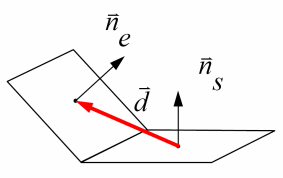


Abbildung 6: Sender und Empfänger sind einander zugewandt.[5]

[5] Das Bild wurde aus Quelle [Müll05] kopiert.

In den meisten Fällen wird der obige Test einwandfrei funktionieren. Es gibt aber auch Fälle in denen sich beide Flächen so überlappen, dass von der einen Fläche der Mittelpunkt hinter der Anderen liegt, sich diese aber trotzdem teilweise „sehen“.

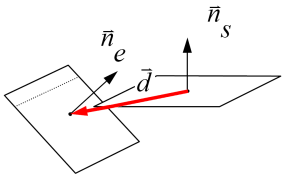


Abbildung 7: Sender und Empfänger sind teilweise einander zugewandt, ein Support-Plane-Split ist nötig.[6]

Damit auch dieser Fall korrekt erfasst wird ist ein Support-Plane-Split nötig. Mit diesem Split wird dafür gesorgt, dass die ursprüngliche Fläche in zwei Teile aufgeteilt wird: den Teil im vorderen Halbraum und den Teil im hinteren Halbraum des Divisors. Für den Zugewandt/Abgewandt-Test sind dann nur noch die Flächen des vorderen Halbraums relevant, da die im hinteren Halbraum dem Divisor nicht zugewandt sein können.

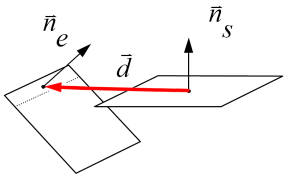


Abbildung 8: Nach dem Support-Plane-Split darf nur mit dem Teil im vorderen Halbraum gearbeitet werden.

Codeauszug:

```
bool canOneSeeAnother(Patch* receiver, Patch* emitter) const
{
	//Create the Vector which connects both Patches
	Vector connection = emitter->getMidpoint() - receiver->getMidpoint();
	Vector normalOfEmitter  = emitter->getNormal();
	Vector normalOfReceiver = receiver->getNormal();

	//Normal of receiver must point in same direction as connection, while
	//normal of emitter has to point in the opposite direction
	if (normalOfReceiver.dot(connection) > 0)
		if (normalOfEmitter.dot(connection) < 0)
			return true;

	return false;
}
```

4.2 Der Support-Plane-Split

Wofür der Support-Plane-Split (zu Deutsch: Hilfsaufteilung) benötigt wird, wurde am Ende von Kapitel 4.1 kurz angerissen. Der Support-Plane-Split wird jedoch nicht nur für den Zugewandt/-Abgewandt-Test benötigt, sondern auch für alle folgenden Visibilitätsberechnungen. Dies sind: die Berechnung der Visibilität, wie beschrieben in Kapitel 4.3, sowie die Berechnung des Formfaktors zwischen zwei Flächen.

[6] Das Bild wurde aus Quelle [Müll05] kopiert.

Würden zwei Flächen z.B. auf Zugewandt/Abgewandt getestet werden ohne den Support-Plane-Split anzuwenden, könnten diese als nicht zugewandt erkannt werden, obwohl sie es doch sind. Auch bei der Formfaktor-Bestimmung kann es ohne diesen Test zu falschen Werten kommen, die sogar den Wertebereich des Formfaktors überschreiten.

Es gibt zwei Verwendungsmöglichkeiten für den Support-Plane-Split. Im ersten Fall wird der Sender vom Empfänger geschnitten. Bei diesem sogenannten *source-split* [Kres97] ist es nicht erforderlich den Emitter zu unterteilen, da keine visuellen Artefakte auftreten können. Jedoch sollte der Formfaktor mit dem geclippten Empfänger-Polygon bestimmt werden, da der Formfaktor ansonsten trotzdem kleiner oder gar negativ werden kann.

Im zweiten Fall schneidet der Sender den Empfänger (*target-split* [Kres97]). Hier würde ohne eine Unterteilung der Empfänger-Fläche eine Unstetigkeit entstehen. Diese Unterteilung kann mit der Quadtree-Struktur erfolgen oder über einen asymmetrischen Schnitt vollzogen werden. Der Vorteil der asymmetrischen Unterteilung ist, dass für beide Polygone die Visibilität klar definiert ist: das Polygon im vorderen Halbraum des Senders ist komplett sichtbar und das im hinteren komplett verdeckt. Die Unterteilung mit Hilfe einer Quadtree-Struktur würde den Support-Plane-Split-Test mehrmals ausführen und so versuchen, die Fläche bis zum Erreichen des mininmal definierten Flächeninhalts zu unterteilen.

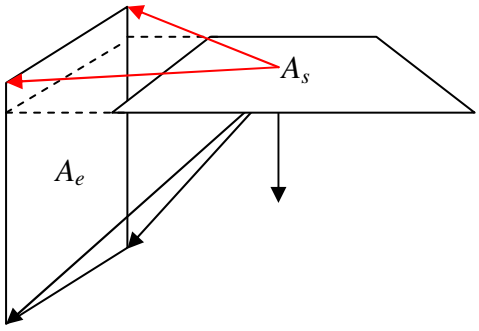


Abbildung 9: target-split: der Empfänger liegt im vorderen und hinteren Halbraum des Senders.

4.2.1 Durchführung des Support-Plane-Splits

Da der Support-Plane-Split für jede Visibilitätsbestimmung notwendig sein kann, muss vor den konkreten Berechnungen von Zugewandt/Abgewandt, Visibilität und Formfaktor geprüft werden, ob ein Support-Plane-Split nötig ist.
Die implementierte Überprüfung, ob eine Fläche einem Support-Plane-Split unterzogen werden muss, kann abhängig von der Reihenfolge der übergebenen Parameter (siehe Codeauszug) diese Frage für den Sender oder Empfänger beantworten. In der hier niedergeschriebenen Erläuterung wird davon ausgegangen, dass der Empfänger auf einen solchen Schnitt getestet wird.

Um also festzustellen, ob ein Support-Plane-Split nötig ist, muss überprüft werden ob alle Eckpunkte des Empfängers im vorderen bzw. hinteren Halbraum des Senders liegen oder auf beide Halbräume verteilt sind. Nur wenn die Eckpunkte auf beide Halbräume verteilt sind muss ein Support-Plane-Split durchgeführt werden.
Für den Test werden von dem Mittelpunkt des Senders Vektoren zu den Eckpunkten des Empfängers erstellt. Zwischen diesen und der Sendernormalen wird dann das Skalarprodukt gebildet. Befindet sich nun einer der Punkte im hinteren Halbraum des Senders, ist das Ergebnis des Skalarproduktes negativ, in allen anderen Fällen positiv. Es wird nun gezählt wie viele Punkte im hinteren Halbraum liegen und wie viele im vorderen. Liegen null oder vier Punkte dahinter, ist ein Split des Empfängers nicht nötig, in allen anderen Fällen schon.

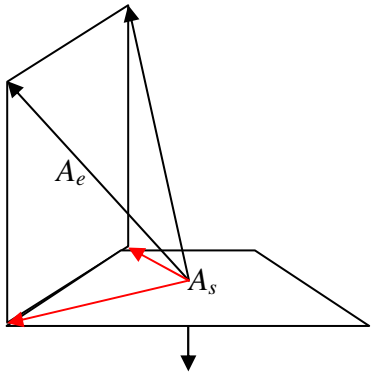


Abbildung 10: Beide Flächen haben vier Eckpunkte, wobei zwei jeweils aufeinander liegen. Test: < 0.

Dies ist die Theorie. In der Praxis müssen allerdings noch ein paar Sonderfälle beachtet werden: So ist z.B. nicht klar, was passiert, wenn zwei Flächen orthogonal zueinander stehen und an zwei Stellen identische Eckpunkte besitzen. Abhängig davon, ob man nun das Ergebnis des Skalarproduktes auf < 0 oder ≤ 0 testet, kommt es in beiden Fällen zu Problemen. Angenomnnen, ein Punkt würde im hinteren Halbraum liegen, wenn das Ergebnis des Skalarprodukts < 0 ist. Bei dem obigen Beispiel würden dann zwei Punkte im hinteren Halbraum und zwei Punkte im vorderen Halbraum erkannt werden. Es müsste also ein Support-Plane-Split durchgeführt werden. Der dann durchgeführte Schnitt wäre aber überflüssig.
Angenomnnen, ein Punkt würde im hinteren Halbraum angenommen, wenn das Skalarprodukt ≤ 0 ist. Nun liegt der zu testende Empfänger aber komplett im hinteren Halbraum des Senders. Auch hier würden wieder zwei Punkte im vorderen und zwei Punkte im hinteren Halbraum erkannt und somit ein Support-Plane-Split durchgeführt werden.
Um diese unnötigen Schnitte zu umgehen, muss also nicht nur geschaut werden welche Punkte im vorderen bzw. hinteren Halbraum liegen, sondern auch, welche Punkte variabel sind. Variable Punkte sind Punkte, an denen sich die Punkte von Sender und Empfänger berühren, bzw. allgemeiner gesagt, Punkte deren Skalarprodukt in einer gewissen Epsilonumgebung um null liegen.

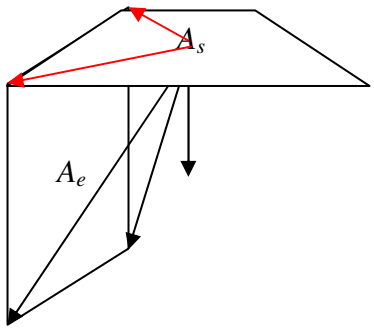


Abbildung 11: Beide Flächen haben vier Eckpunkte, wobei zwei jeweils aufeinander liegen. Test: ≤ 0.

Ist unter allen Eckpunkten also ein variabler Punkt dabei, wird kein Support-Plane-Split durchgeführt, da es in solchen Fällen keine neuen Schnittpunkte geben kann. Denn die Berührungspunkte sind bereits die Schnittpunkte.
Allerdings hat auch diese Lösung eine Schwäche. Angenommen, es sei nur ein Punkt variabel. In diesem Fall wäre nicht klar, ob sich die beiden Flächen schneiden und somit ein Support-Plane-Split nötig wäre. In diesem Fall sollte also trotzdem ein Schnitt-Test durchgeführt werden, da nicht sichergestellt werden kann, dass sich zwei Flächen nicht doch schneiden.

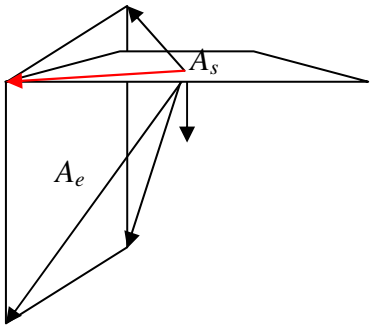


Abbildung 12: Beide Flächen haben nur einen gemeinsamen Punkt, trotzdem ist ein Support-Plane-Split nötig.

Codeauszug:

```
//Check if all Vertices lie in front of the Patch
for (short i = 0; i  < dividend->getNumberOfVertices(); i++)
{
	//Taking one point of the receiver and connect it with the
	//midpoint of the emitter
	A = dividend->getVertex(i);
	a = A - midpoint;

	//Is one Vertex out of sight?
	dot = normal.dot(a);

	//This is to avoid wrong splits
	if (dot < EPSILON && dot > -EPSILON)
	{
		variableOut++;
		continue;
	}

	//Is this Vertex behind the divisor?
	if (dot < 0)
		out++;
}
//Emitter and receiver that have same Vertices shall not be splitted
if (variableOut > 0)
	return false;

//Are all Vertices out, or none?
if (out < 4 && out > 0)
	return true;

//No Support-Plane-Split needed
return false;
```

4.2.2 Berechnung des Support-Plane-Splits

Muss ein Support-Plane-Split vorgenommen werden, geschieht dies über den Geraden-Ebenen-Schnitt der Mathematik. Hierzu wird für einen *target-split* der Sender als Ebene angesehen, die von den Geraden des Empfängers geschnitten wird, wobei die Verbindung zweier Eckpunkte jeweils eine Gerade bildet. Der durch den Schnitt bestimmte Parameter t kann dann in die Geradengleichung eingesetzt werden und liefert direkt den Schnittpunkt S. Dabei muss darauf geachtet werden, dass der Parameter t den Wertebereich $0 \leq t \leq 1$ nicht verlässt. Nur so ist gewährleistet, dass der Schnittpunkt innerhalb der Fläche des Empfängers liegt und die Senderebene nicht in der „Unendlichkeit“ schneidet.

Für den üblichen Weg würde man zuerst die Ebenengleichung in Parameterform aufstellen und diese anschließend in die Hesse-Form überführen. Da es aber nun aufwendig wäre, den Computer für jeden

Schnitt ein lineares Gleichungssystem lösen zu lassen, müsste man die zu verwendenden Variablen vorbestimmen.

$$g : \vec{x} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} + t \begin{pmatrix} r_1 \\ r_2 \\ r_3 \end{pmatrix} \Rightarrow \begin{array}{r} x_1 = p_1 + t \cdot r_1 \\ x_2 = p_2 + t \cdot r_2 \\ x_3 = p_3 + t \cdot r_3 \end{array}$$

Dies könnte man machen, indem man für die Überführung in die Hesse-Form die drei Gleichungen für die Normalenkomponenten x_1, x_2 und x_3 aufstellt. Allerdings müsste man sich zweimal entscheiden, welche Gleichung in welche der verbleibenden eingesetzt wird. Die dadurch erhaltenene Lösung zur Bestimmung der Komponenten der Normalen $\vec{n}$ würde nur in manchen Fällen funktionieren. Bei einer Berechnung kann es nämlich passieren, dass z.B. ein Richtungsvektor zwei Nullen enthält. Diese könnten dann dazu führen, dass die Normale $\vec{n}$, und somit auch der Parameter *t*, nicht bestimmt werden können.
Wenn man die Normale für die Ebene allerdings schon besitzt, z.B. über die Bildung des Kreuzproduktes zu zwei benachbarten Eckpunkten einer Fläche, kann man in die Hesse-Form der Ebene die Hesse-Form der Geraden einsetzen und somit den Parameter *t* der Geradengleichung bestimmen.

$$E : x_1 + x_2 + x_3 = d \text{ , mit } \begin{array}{r} x_1 = p_1 + t \cdot r_1 \\ x_2 = p_2 + t \cdot r_2 \\ x_3 = p_3 + t \cdot r_3 \end{array}$$

Ein wesentlich einfacherer und schnellerer Weg zur Lösung des Gerade-Ebene-Schnitt-Problems ist die allgemein verwendete Formel für die Bestimmung des Parameters *t*.

$$t = \frac{\vec{n} \circ P + d}{\vec{n} \circ (P - R)}$$

Diese leitet sich aus dem Sutherland-Hodgeman-Algorithmus zum Clippen eines beliebigen Polygonzugs ab. Denn auch hier ist über das Skalarprodukt zwischen der Normalen des Sendes und den Punkten des Empfängers eine Clipping-Ebene festgelegt. Ist so z.B. das Ergebnis des Skalarprodukts für einen Punkt einmal größer als null und für den Nachfolger kleiner null, muss für diese Punkte der Schnittpunkt bestimmt werden. Dies geschieht dann über die obige Formel.

4.3 Die Visibilität bestimmen

Visibilitätstests werden üblicherweise mit Strahlen durchgeführt, die das Volumen zwischen Sender und Empfänger „durchleuchten“. Hierfür wird die Anzahl der angekommenen Strahlen durch die Anzahl der losgeschickten Strahlen geteilt, und so erhält man für die Visibilität Werte zwischen null, keine Sichtbarkeit, und eins, volle Sichtbarkeit.
Das Ziel der Visibilitätsbestimmung ist dabei, einen möglichst genauen und robusten Wert in möglichst kurzer (Prozessor-) Zeit zu liefern. Robust bedeutet in diesem Zusammenhang, dass bei einer zufälligen Generierung von Punkten auf zwei Flächen, hier Sender und Empfänger, trotz der zufälligen Generierung auch nach mehrmaliger Durchführung Ergebnisse geliefert werden, die innerhalb eines bestimmten Fehlers liegen.
Ein robustes Verfahren muss die Punkte für den Visibilitätstest somit möglichst gleichverteilt auf beiden Flächen generieren und idealerweise auch das Volumen, dass sich zwischen Sender und Empfänger befindet, gleichverteilt abtasten. Ein nicht robustes Verfahren würde ansonsten für dieselbe Situa-

tion, z.B. einmal volle Sichtbarkeit und ein anderes Mal nur eine Teilsichtbarkeit liefern. Dies ist aber nicht erwünscht, da folglich auch sehr unterschiedliche Bilder entstehen würden. Also ist man bemüht, die Menge der möglichen Ergebnisse auf die Menge der gewünschten Ergebnisse zu reduzieren.
In den folgenden Abschnitten werden zwei Verfahren vorgestellt: die Visibilitätsbestimmung über Raytracing und das „Magische Quadrat".

4.3.1 Das „Magische Quadrat"

Für das „Magische Quadrat" [Müll05] werden sowohl der Sender als auch der Empfänger in ein regelmäßiges Gitter unterteilt. Für jedes Element der einen Fläche wird dann zufällig ein Partnerelement auf der anderen Fläche bestimmt. Dann wird von dem Mittelpunkt eines Elements ein Strahl zum Mittelpunkt eines Partnerelements geschickt. Damit keine absoluten Start- und Endpunkte für einen Strahl vorgegeben werden, werden die Start- bzw. Endpunkte gejittert, d.h. sie werden zufällig in einem bestimmten Radius um den eigentlichen Mittelpunkt erzeugt.
Dieses Vorgehen gewährleistet zwar eine Gleichverteilung der Punkte auf der Sender- bzw. Empfängerfläche, durch die zufällige Partnerwahl kann es aber auch zu Problemen kommen. Angenommen, es werden allen Senderelementen am Rand den Empfängerflächen des gegenüberliegenden Randes zugewiesen und Selbiges geschieht für die oberen und unteren Ränder. Werden nun die Strahlen verschossen, ist die Strahlendichte in der Mitte des Volumens sehr hoch, am Rand aber sehr niedrig. Somit ist es möglich, dass Okkluder, die sogar bis ein Viertel in das Volumen zwischen Sender und Empfänger hineinragen, nicht erkannt werden.

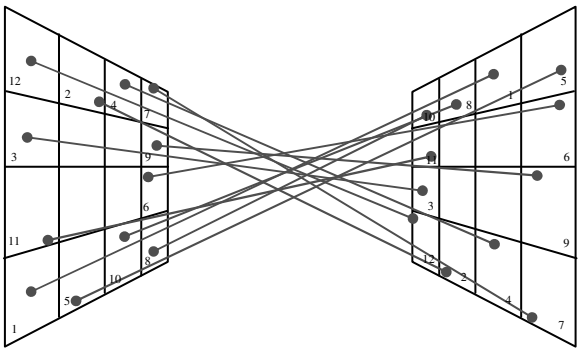


Abbildung 13: Bei dieser ungünstigen Konstellation wird nur die Mitte nach Objekten „durchleuchtet".[7]

Die Wahrscheinlichkeit für ein solches Vorkommen kann zwar reduziert werden indem mehr Strahlen verschossen werden, aber dies kostet dann natürlich auch mehr Zeit. Ausgeschlossen werden kann die Situation allerdings nicht.
Durch die zufällige Partnerwahl ist also die Wahrscheinlichkeit höher, dass Objekte im Inneren des Volumens entdeckt werden, die Wahrschenlichkeit, Objekte am Rand zu finden ist hingegen aber viel geringer.

4.3.2 Sichtbarkeit durch Raytracing

Eine Alternative zum „Magischen Quadtrat" ist die Sichtbarkeit durch Raytracing zu bestimmen. Bei diesem Verfahren geht man davon aus, dass eine Fläche, wie bei der Formfaktor-Bestimmung durch die erste Formfaktor-Vereinfachung, infinitesimal klein ist und die andere Fläche eine endliche Ausdehnung hat. Nun wird auch für dieses Verfahren wieder ein $n \times n$-Raster benötigt, welches aber diesmal nur über die endliche Fläche gelegt wird. Für jedes Element des Rasters wird nun wieder zufällig ein Punkt generiert, der dann als Endpunkt dient. Auch diese Punkte werden wie beim „Magischen Quadrat" gejittert, um unschöne Darstellungsfehler durch die Diskretisierung zu vermeiden.

[7] Zur besseren Übersicht wurden nur die äußeren Strahlen eingezeichnet.

Im nächsten Schritt wird dann vom Mittelpunkt der infinitesimal kleinen Fläche zu jedem Rasterelement der endlichen Fläche ein Strahl geschossen. Dieses Verfahren funktioniert gut, wenn man sich vor der Ausführung des Visibilitättests entscheidet, immer die kleinere Fläche als infinitesimal klein anzunehmen und von diesem Mittelpunkt aus die Strahlen zu verschießen. Sind die Flächen allerdings so groß, dass sich auch noch Objekte am Rand der infinitesimal kleinen Fläche aufhalten könnten, würden diese nie erkannt. Auch wenn das Verfahren in dem Sinne robust ist, dass es bei mehrfacher Ausführung gleichbleibende Ergebnisse liefert, reicht eine Gleichverteilung der Punkte auf der einen Seite bei großen Flächen meistens nicht aus.

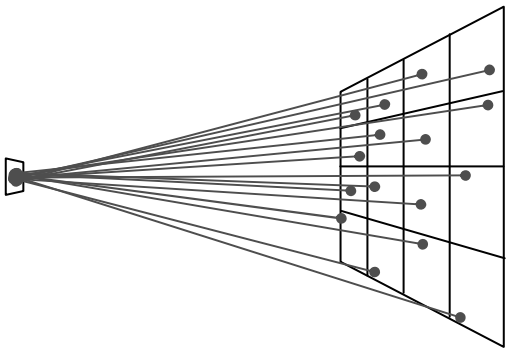


Abbildung 14: Bei diesem Verfahren werden Objekte, die in die linke Fläche reinragen, nicht erkannt.

4.3.3 Verwendetes Verfahren

Das in dieser Studienarbeit implementierte Verfahren ist ein nicht robustes Verfahren, welches sich eher an dem „Magischen Quadrat“ orientiert. Soll ein Visibilitätstest durchgeführt werden, wird zuerst festgelegt, wieviel Strahlen durch das Volumen zwischen Sender und Empfänger geschossen werden sollen. Dies kann entweder manuell oder aber durch einen speziellen Modus geschehen, den adaptiven Modus. Im adaptiven Modus werden zuerst beide Flächen verglichen und für die Berechnung der maximalen Anzahl an Strahlen die Grössere gewählt. Die Anzahl der Strahlen für den adaptiven Modus ergibt sich durch

$$\#Rays = \sqrt{A_{\max}} \cdot 4 + 1$$

wobei die „+1“ dafür sorgt, dass egal wie klein A_{max} ist, mindestens ein Strahl verschossen wird. Die obige Berechnung liefert aber, abhängig von der Skalierung der Szene, für ein und dieselbe Szene unterschiedliche Ergebnisse, da diese nur von der Fläche abhängt. Besser wäre es, die Beziehung Abstand-zu-Fläche in die Berechnung mit einzubeziehen, da nur so eine konstante Anzahl an Strahlen, auch für unterschiedlich große Flächen, gewährleistet wäre.
Im zweiten Schritt werden dann für jeden Strahl zwei Punkte generiert, einer auf der Empfänger- und einer auf der Senderseite, die dann als Start- und Endpunkt für einen Strahl gelten. Bei der Generierung wird zuerst aus allen Eckpunkten einer Fläche zufällig ein Eckpunkt ausgewählt. Mit diesem und dem nachfolgenden Eckpunkt kann ein Vektor erstellt werden, um dann eine Geraden-Gleichung aufzustellen. Dieser Geraden wird nun eine zufällige Strecke [0;1] gefolgt und ergibt dann den Punkt *S*. Von dem Punkt *S* aus wird nun ein neuer Vektor von *S* nach *M*, dem Mittelpunkt der Fläche, erstellt. Mit diesem Vektor kann nun erneut eine Geradengleichung aufgestellt werden. Der Geraden wird dann wieder eine zufällige Strecke [0;1] gefolgt.

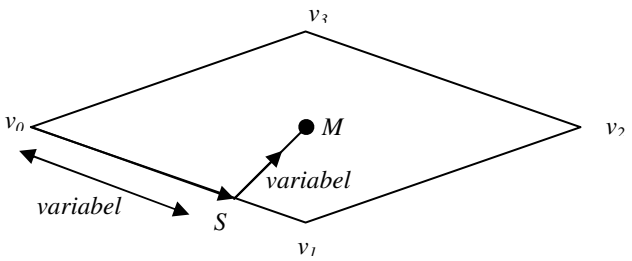


Abbildung 15: In diesem Beispiel wird am Punkt v_0 begonnen. Dieser wird aber zufällig ausgewählt.

Dieses Verfahren ist sehr stark vom Zufall gesteuert, so dass verschiedene Tests ergaben, dass selbst bei einer Strahlenanzahl von 1000 noch eine Abweichung von 14% von der minimalen (29%) und der maximalen Visibilität (43%) möglich ist. Das Problem ist, dass es durch die zufällige Generierung der Punkte auf beiden Flächen theoretisch passieren kann, dass alle Strahlen von demselben Punkt ausgehen und am selben Punkt enden. Dies würde dann so wirken, wie als würde nur ein Strahl verschickt werden. Eine Verbesserung könnte nur erreicht werden, indem wie beim „Magischen Quadrat“ ein Raster erzeugt wird und somit eine ungefähre Gleichverteilung der Punkte über die Fläche garantieren wäre.

Codeauszug:

```
Vertex generateVertex(Patch* patch)
{
	short size = patch->getNumberOfVertices();

	//Init all Vertices that are needed
	Vertex A, B, AB, S, M, SM;

	//Generate random values, up to number of Vertices, but not greater
	//than size-1
	short random = (float)(rand() % size);

	//Get two random Vertices from the Patch: from 0 to size-1
	A = patch->getVertex(random);
	B = patch->getVertex((random + 1) % size);

	//First Vector
	AB = B - A;

	//First random part of the random Vertex
	S = A + ((float)(rand() % 1000) / 1000) * AB;

	//Getting the midpoint of the Patch
	M = patch->getMidpoint();

	//Second part of the random Vertex
	SM = M - S;

	//Returning the random Vertex
	return S + ((float)(rand() % 1000) / 1000) * SM;
}
```

4.4 Beschleunigung von Visibilitätstests

Ein Thema das bis jetzt noch nicht erwähnt wurde, aber ebenso stark die Performanz beeinflussen kann wie das ausgewählte Verfahren, ist die Anzahl der getesteten Flächen. Abhängig von der Szenengröße müssen wenige oder sehr viele Flächen getestet werden, da letztlich jede Fläche in der Szene zwischen dem Sender und Empfänger liegen kann. Die Anzahl der zu testenden Flächen beeinflusst

die Performanz natürlich maßgeblich, da ein Visibilitätstest für jede Fläche erneut durchgeführt werden muss. Aber auch in diesem Bereich kann sehr viel optimiert werden.

4.4.1 Beschleunigung durch Szenengraphen

Die wohl besten Möglichkeiten der Optimierung bietet ein Szenengraph, da durch die Struktur des Szenengraphen für jedes Objekt klar ist, in welchem Bereich der Szene es sich befindet. Durch einen Szenengraphen, der z.B. durch einen Octree realisiert werden kann, brauchen nur die Flächen getestet werden, die sich in demselben Sektor befinden wie Sender und Empfänger. Natürlich kann es auch hier passieren, dass viele Flächen getestet werden die nicht betroffen sind, weil sie nicht zwischen Sender und Empfänger liegen, z.B. zwei sich gegenüberliegende Fliesen in einem Bad. Daher ist eine geschickte Aufteilung der Sektoren wichtig. Alternativ kann die Anzahl der zu testenden Flächen minimiert werden, wenn die Tiefe des Szenengraphen erhöht wird.

4.4.2 Beschleunigung durch Pre-Tests

Eine andere Möglichkeit zur Optimierung, ohne den Overhead des Szenengraphen, bietet ein eigens entwickeltes Verfahren der „Outcode-Test“, welches aber dem bereits 1991 von Haines und Wallace entwickelten Verfahren mit dem Namen „Shaft-Culling“ gleicht.
Der „Outcode-Test“ ist ein Pre-Test und dient dazu, vor dem Visibilitätstest mit möglichst wenig Aufwand möglichst viele Flächen vom eigentlichen Visibilitätstest auszuschließen. Dieser Test arbeitet dabei, ähnlich wie das Cohen-Sutherland Clipping-Verfahren, mit einem Outcode.
Mit dem hier vorgestellten Verfahren wird mit Hilfe eines Outcodes festgestellt, welche Flächen sich innerhalb bzw. außerhalb des Volumens zwischen Sender und Empfänger befinden. Der Raum um das Volumen, das von Sender und Empfänger aufgespannt wird, wird dabei in neun Sektoren unterteilt.

Damit die Sektoren aufgestellt werden können müssen allerdings zuerst die Minima und Maxima des Volumens in x-, y- und z-Richtung bestimmt werden. Diese dienen dann, unabhängig von der Lage des Volumens, als Grenzen zwischen den einzelnen Sektoren. Man kann sich dabei eine Grenze, z.B. das Minimum auf der x-Achse, als unendliche Ebene vorstellen, die man an das Minimum des Volumens heran schiebt, bis es auf der gleichen Höhe mit diesem liegt. Jeder dieser Sektoren hat dann seinen eigenen Outcode, wobei der Original Outcode von Cohen-Sutherland auf Grund der drei Dimensionen von 4 auf 6 Bit erweitert wurde.
Der Outcode selbst stellt ein Muster dar, anhand dessen die Position eines Eckpunktes im dreidimensionalen Raum abgelesen werden kann. Der Outcode für den Sektor in dem sich Sender und Empfänger befinden lautet 000000 und kann als Ursprung angesehen werden. Eine genaue Erläuterung wie der Outcode definiert ist wird in der Abbildung 16 erläutert.
Mit Hilfe des Outcodes kann also für jeden Eckpunkt einer Fläche festgestellt werden, ob er im Volumen zwischen Sender und Empfänger, oder außerhalb des Volumens liegt. So gibt es neben dem Fall, in denen Flächen nicht weitergetestet werden müssen, weil sie außerhalb des Volumens liegen, drei Fälle in denen eine Fläche an den Visibilitätstest weitergereicht wird:

1. ein Eckpunkt einer Fläche liegt innerhalb des Volumens
2. alle Eckpunkte liegen außerhalb und schneiden das Volumen nur
3. alle Eckpunkte liegen außerhalb und trennen den Sender vom Empfänger durch eine totale Verdeckung[8]

[8] Im letzten Fall kann der Visibilitätstest direkt abgebrochen werden, da sich Sender und Empfänger nicht sehen können.

Bei der Erstellung des Outcodes für jede Fläche sind grundsätzlich alle Eckpunkte beteiligt. Dabei werden zwei Eckpunkte immer als Linie betrachtet, indem der Outcode zweier Eckpunkte verundet wird. Diese können dann, wie in dem Clipping-Algorithmus von Cohen-Sutherland, behandelt werden: Liefert das Ergebnis einer Verundung von zwei Eckpunkten null, muss diese Fläche vom Visibilitätstest bearbeitet werden, da es möglich ist dass sie innerhalb des Volumens liegt.
Um den Test zu beschleunigen, werden für eine Fläche immer nur die Diagonalen einer Fläche getestet, da diese die Ausmaße einer Fläche ausreichend beschreiben und somit zwei Tests gespart werden können. Dies ist nötig, da das Verfahren nur so beliebige konvexe Quads unterstützen kann.

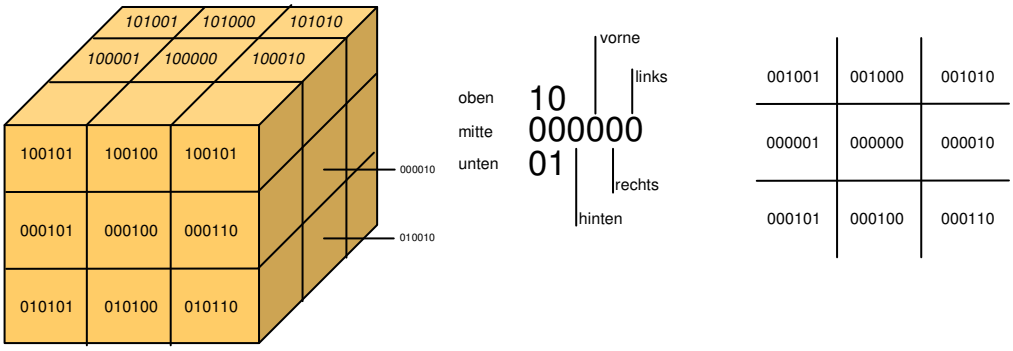


Abbildung 16: Die Mitte zeigt die Einteilung des Outcodes in die verschiedenen Ebenen. Rechts ist der original Cohen-Sutherland zu sehen, erweitert um die 3D-Bits. Links ist das 3D-Volumen mit einigen exemplarischen Outcodes zu sehen.

4.4.2.1 Bewertung

Im schlechtesten Fall ist der Pre-Test genauso wie der Szenengraph nutzlos, da alle Flächen zwischen Sender und Empfänger liegen. Somit kommt zu dem Aufwand des Visibilitätstests noch der des Pre-Tests dazu. Vergleiche zwischen dem Visibilitätstest mit und ohne Pre-Test haben ergeben, dass dieses Verfahren bei der Aussendung eines Strahls in der Demo-Szene bis zu 1.5mal langsamer ist. Sobald aber anstatt eines Strahls 11 Strahlen pro Test verschossen werden, ist dieses Verfahren um den Faktor 1.5 schneller als das Standardverfahren ohne Pre-Test. Das hängt damit zusammen, dass der Outcode immer für ein Volumen erstellt wird. Wird dieses Volumen nun mit nur einem Strahl durchschossen ist der Aufwand für die Erstellung der Liste mit den zu testenden Flächen größer als der Nutzen. Umgekehrt steigt dieser natürlich bei mehreren Strahlen. Wird dabei in den adaptiven Modus umgeschaltet, machen sich beide Erkenntnisse bemerkbar: Hierbei beschleunigt der Pre-Test solange den Visibilitätstest, bis auf Grund der Flächengröße nur noch einzelne Strahlen verschickt werden. Dies bestätigen auch die Ergebnisse der Tabelle 1.
Der maximale Gewinn kann erzielt werden, wenn alle Flächen außerhalb des Volumens liegen. Der Visibilitätstest würde hierbei komplett wegfallen, so dass nur noch der Aufwand des Pre-Tests zu Buche schlagen würde.

Strahlen	Modus	Zeit	Links
20	fast	141s	125k
20	normal	308s	125k
10	fast	85s	125k
10	normal	198s	124k
adaptiv	fast	149s	125k
adaptiv	normal	176s	125k

Tabelle 1: Gegenüberstellung einzelner Modi des Visibilitätstests: nomal = ohne Pre-Test, fast = mit Pre-Test.[9]

[9] Die hier aufgeführten Ergebnisse wurden in der Demo-Szene mit dem BFA-Orakel durchgeführt.

Hier ist eine Tabelle zu finden, in der verschiedene Performance-Messungen durchgeführt wurden. Diese enthält die Anzahl der Links, die Anzahl der Strahlen, die für den Visibilitätstest verwendet wurden, sowie die gemessene Zeit. Die Tests erfolgten im Release Modus, wobei die Applikation außerhalb der Entwicklungsumgebung gestartet wurde. Das Testsystem war ein P3 550MHz mit 256MB RAM. Es wurden nur Tests mit 10, 20 Strahlen und dem adaptiven Modus durchgeführt. Tests mit weniger als 10 Strahlen führten auf Grund des verwendeten Visibilitätstests zu keinen repräsentativen Ergebnissen, da es passieren kann dass Flächen auf hohen Leveln nicht verlinkt werden, da der Visibilitätstest negativ ausgefallen ist. Dies kann ganze Regionen aussparen und somit zu sehr schwankenden Ergebnissen führen.

Codeauszug:

```
//Get minima and maxima for the pre-test
std::vector<double> minimaAndMaxima = findMinimaAndMaxima(receiver, emitter);

//Iterate over all Surfaces of the Scene
for (unsigned int i = 0; i < mSurfaces->size(); i++)
{
	//Set pointer
	surface = (*mSurfaces)[i];

	//If the Surface is set to NULL, ignore it
	if (surface == NULL)
		continue;

	//If the current Surface has no Quadtree, it also has no Patch
	if (surface->hasQdtree() == false)
		continue;

	patch = surface->getRoot()->getPatch();

	//Check if this is receiver or emitter Patch don't test them
	if (patch->getNode()->getQdtree() == receiver->getNode()->getQdtree())
		continue;

	//...nor another Patch from the receiver/emitter Surfaces
	if (patch->getNode()->getQdtree() == emitter->getNode()->getQdtree())
		continue;

	//Get all needed parts
	v0 = &patch->getVertex(0);
	v2 = &patch->getVertex(2);
	v1 = &patch->getVertex(1);
	v3 = &patch->getVertex(3);

	//Get the outcode, so we know where the Vertices are
	outcode_v0v2 = outcode(v0, minimaAndMaxima)
		     & outcode(v2, minimaAndMaxima);

	outcode_v1v3 = outcode(v1, minimaAndMaxima)
		     & outcode(v3, minimaAndMaxima);

	//See Cohen-Sutherland: if zero, the Surface lies in the volume
	if (outcode_v0v2 == 0 || outcode_v1v3 == 0)
		suspiciousSurfaces.push_back(surface);
}
return suspiciousSurfaces;
```

5. Radiosity-Verfahren

In diesem Kapitel werden die klassischen Radiosity-Verfahren vorgestellt. Die ersten Verfahren mit denen ein Strahlungsaustausch über die Radiosity-Gleichung durchgeführt werden konnte, arbeiteten mit der Fullmatrix-Methode. Diese wurde dann 1988 von Cohen et al. durch das Progressive Refinement abgelöst. Letztlich wird in Abschnitt 5.3 noch auf den Vorgänger des hierarchischen Radiosity, das hierarchische Shooting, eingegangen.

5.1 Fullmatrix Methode

Die in Kapitel 1 vorgestellte Rendering Equation wurde als Ausgangsgleichung genutzt, um die Radiosity-Gleichung von ihr abzuleiten. Ist es in der Rendering Equation nötig für die Ermittlung der Leuchtdichte eines Flächenelements dA_e die einfallenden Leuchtdichten über alle möglichen Winkel zu betrachten, konnte dies durch die Radiosity-Gleichung stark vereinfacht werden. Denn da die Radiosity nur noch von Flächen ausgestrahlt werden kann die in einer Szene existieren, reicht es, diesen Anteil durch den Formfaktor zu bestimmen und mit der ausgestrahlten Radiosity der Senderfläche s zu multiplizieren. Summiert man alle einzelnen Radiosity auf, erhält man die Radiosity für die Empfängerfläche e.

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

Es muss also für eine Szene mit n Flächen für jede Fläche aus der Szene die obige Gleichung aufgestellt und gelöst werden. Anstatt nun alle diese Gleichungen untereinander zu schreiben, konnten diese durch eine Umformung, man löst jede Gleichung nach E auf und multipliziert ρ mit allen Elementen der Summe, zu einer Gleichung der Form

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

umgestellt werden. Der Vektor B enthält dann die gesuchte Radiosity für jede Fläche, wohingegen der Vektor E die Eigenemissionen für jede Fäche aufführt. Die Matrix M hat dann die Größe $n \times n$, wobei n die Anzahl der Flächen ist, die Radiosity aussenden können.

$B_1 = E_1 + \rho_1 (B_1 \cdot F_{11} + B1 \cdot F_{12} + ... + B_n \cdot F_{1n})$ für n Flächen ergibt

$$\begin{pmatrix} (1-\rho_1 F_{11}) & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & (1-\rho_2 F_{22}) & \cdots & -\rho_2 F_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \dots & (1-\rho_n F_{nn}) \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

Zur Lösung einer solchen Gleichung müssten also n^2 Formfaktoren berechnet werden, wobei das Gleichungssystem für jede Farbe, z.B. rgb-Farbraum, aufgestellt werden muss. Wenn allerdings die Eigenschaften des Formfaktors ausgenutzt werden, müssen nicht wirklich n^2 Formfaktoren berechnet werden. Durch die Reziprozitätsbedingung des Formfaktors reicht es somit aus, die Formfaktoren oberhalb der Diagonalen auszurechnen und den unteren Teil der Matrix über die Reziprozizätsbedingung zu bestimmen.

$$F_{es} \cdot A_e = F_{se} \cdot A_s$$

Ebenso ist es nicht nötig, die Formfaktoren für die Diagonale zu bestimmen, da der Formfaktor zu sich selbst stets als null definiert ist und die Diagonale somit nur Einsen enthalten kann. Zu guter Letzt kann der letzte Formfaktor einer Zeile erschlossen werden, da die Summe aller Formfaktoren in geschlossenen Umgebungen stets 1 ist. Der letzte Formfaktor ist daher die Summe aller anderen Formfaktoren einer Zeile subtrahiert von 1.
Es reicht im Idealfall also aus *1/2 (N-1)(N-2)* Formfaktoren zu berechnen, wobei dies in der Praxis meist mit der ersten Formfaktorvereinfachung geschieht.

$$\begin{pmatrix} 1 & x & x & x & * \\ & 1 & x & x & * \\ & & 1 & x & * \\ & & & \ddots & * \\ & & & & 1 \end{pmatrix}$$

Die Lösung dieses Gleichungssystems kann dann entweder über direkte Methoden, wie z.B. das Gauß-Eliminationsverfahren oder auch über die Singulärwert-Zerlegung (SVD) gelöst werden. Da diese Verfahren jedoch viel Rechenzeit benötigen, können alternativ die indirekten Methoden verwendet werden. Entsprechende Methoden sind die Gauß-Seidel-, oder aber die Jakobi-Iteration.
In diesen Verfahren wird die gesuchte Vektor *B* zu Beginn mit der Eigenemission *E* initalisiert. Anschließend werden dann der Vektoren und die Matrix auf eine Seite gebracht und ein Residuum-Vektor *R* bestimmt. Mit diesem Residuum-Vektor kann dann nach jeder Iteration die Abweichung von der wirklichen Lösung ermittelt werden. Die Verfahren konvergieren nach mehreren Iterationen, wobei als Abbruchkriterium entweder die Anzahl der Iterationen oder ein Epsilonwert für den Residuum-Vektor gewählt werden kann.

5.1.1 Bewertung

Das Verfahren ist durch die Anzahl der zu berechnenden Formfaktoren zwar realisierbar, aber dadurch, dass alle Formfaktoren im Speicher gehalten werden, ist das Fullmatrix-Verfahren nur für kleine Szenen möglich. Denn dadurch, dass für die Ausgangsflächen im schlechtesten Fall n^2 Formfaktoren gespeichert werden und diese auch noch unterteilt werden, was auf Grund der „Constant Radiosity Assumption“ nötig ist um ein Resultat ohne bzw. mit wenigen Unstetigkeiten in Form von harten Flächenkanten zu erhalten, werden für eine Unterteilung wieder m^2 Flächen erzeugt. Für jede Unterteilung müssen dann wieder n^2 Formfaktoren berechnet und gespeichert werden. Die folgende Formel liefert die Anzahl der zu berechnenden Formfaktoren, wobei im konkreten Fall noch die Speicherung im System beachtet werden muss.

$\#Formfaktoren = n^2 \cdot m^4$, wobei m die Anzahl der Unterteilungen pro Fläche ist

Da die Fullmatrix-Methode zu den Gathering-Verfahren zählt, wird analog zur Rendering-Equation, für eine Fläche alle Radiosity der Umgebung eingesammelt. Der Nachteil ist, dass auch die Radiosity von Flächen eingesammelt wird, die überhaupt keine oder nur sehr wenig Radiosity besitzen. Diese Probleme werden durch den nächsten Schritt in der Entwicklung, das Progressive Refinement, behoben.

5.2 Progressive Refinement

Das Progressive Refinement wurde bereits 1988 von Cohen et al. [Cohe88] vorgestellt und ist, anders als seine Vorgänger in denen die Radiosity „eingesammelt“ wurde, kein Gathering-Verfahren. Das

Progressive Refinement ist ein Shooting-Verfahren und unterscheidet sich in zwei Punkten grundlegend von den vorherigen Gathering-Verfahren. Der erste Unterschied ist, dass nicht mehr in Iterationen, sondern in Relaxationen vorgegangen wird. Wurde in einem Schritt der Fullmatrix-Methode für eine Fläche stets die gesamte Radiosity eingesammelt, also die Summe der Radiosity-Gleichung ausgewertet, wird in einem Schritt des Progressive Refinement für jede Fläche immer nur eine Komponente der Summe bestimmt. Somit benötigt das Progressive Refinement zwar mehr Relaxationen, nicht aber unbedingt mehr Zeit.
Der zweite Unterschied ist, dass wie zuvor, zwar alle Formfaktoren berechnet werden, eine Speicherung dieser hingegen ist nicht mehr nötig.

5.2.1 Das Verfahren

Um einen Strahlungsaustausch durchzuführen, besitzt eine Fläche im Progressive Refinement zwei Variablen in denen die Radiosity gespeichert wird. In B wird die absolute Radiosity der Fläche gespeichert, also der Gesamtanteil an Strahlung, der von dieser Fläche empfangen wurde. In ΔB wird hingegen nur der Anteil an Radiosity abgelegt, der von dieser Fläche auch weitergegeben werden darf. Der Ablauf des Progressive Refinement sieht wie folgt aus:

Zu Beginn werden alle Flächen mit ihrer Eigenemission initalisiert, d.h. B und ΔB wird der Wert der Eigenemission E zugewiesen. Im ersten Schritt wird dann die hellste Fläche s für eine Relaxation bestimmt. Die hellste Fläche ist dabei nicht die, deren absolute Radiosity am höchsten ist, sondern die Fläche, die noch am meisten Radiosity „verschießen" darf. Wurde diese Fläche s gefunden, wird ihre Radiosity an alle anderen Flächen verschossen und anschließend der Wert von ΔB auf null gesetzt. Nun kann wieder die hellste Fläche gesucht werden, die dann ebenfalls ihre Radiosity ΔB an alle anderen Flächen verschießt, wobei es vorkommen kann, dass eine Fläche mehrmals als Sender ausgewählt wird.
Diese Schritte werden nun so lange wiederholt bis entweder eine bestimmte Anzahl an Relaxationen erreicht wurde, oder aber die Radiosity ΔB des aktuellen Senders so gering ist, dass das Verfahren abgebrochen werden kann. Die Lösung ist dann konvergiert.

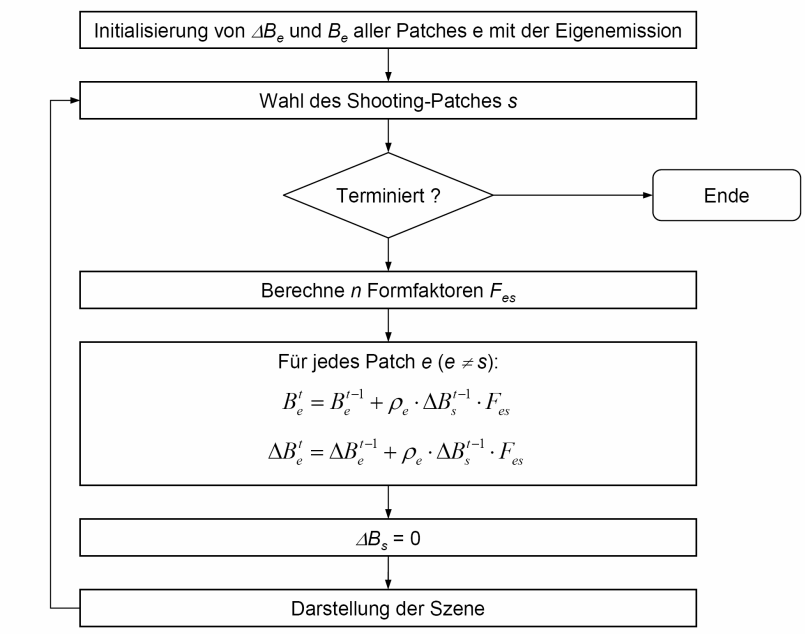


Abbildung 17: Der Ablauf des Progressive Refinements.[10]

[10] Das Bild wurde aus Quelle [Müll05] kopiert.

5.2.2 Bewertung

Dass jede Fläche mit berechnet werden muss, auch wenn sie keine oder nur wenig Radiosity aussendet, ist ein Nachteil der Fullmatrix-Methode. Das Progressive Refinement hingegen sucht für jede Relaxation den hellsten Sender, so kann garantiert werden, dass nur die Flächen ihre Strahlung „verschießen“ dürfen, die zu Beginn einer Relaxation die meiste noch unverschossene Radiosity besitzen. Flächen, deren Beitrag zur Gesamtlösung dabei so gering ist, dass sie das optische Ergebnis nur minimal beeinflussen, werden somit ignoriert.
Ein weiterer Vorteil des Progressive Refinement ist, dass dieses ohne die Speicherung der Formfaktoren auskommt und somit theoretisch auch für größere Szenen eingesetzt werden kann. Da der Strahlungsaustausch aber immer noch sehr zeitaufwendig ist, werden hierfür eher Verfahren wie das hierarchische Shooting oder das hierarchische Radiosity verwendet.
Anzumerken bleibt, dass im Gegensatz zur Fullmatrix-Methode für das Progressive Refinement ein Orakel verwendet werden kann, und somit eine adaptive Unterteilung ermöglich wird.

5.3 Das hierarchische Shooting

Grundsätzlich kann man bei Radiosity-Verfahren beobachten, dass die Berechnung des direkten Lichts einen Großteil der gesamt benötigten Zeit beansprucht, da dort auch am meisten Strahlung versendet wird. Aber warum dauert die Berechnung des indirekten Lichts genau solange, obwohl der Beitrag der versendet wird wesentlich geringer ist? Der Grund dafür ist, dass beim Verfahren, wie z.B. dem Progessive Refinement zwar nur die hellsten Flächen ihre Radiosity „verschießen“ dürfen, es aber abhängig von den Unterteilungen sehr viele sein können. Um hier die Effizenz zu verbessern, muss also entweder die Anzahl der Empfänger oder die Anzahl der Sender verringert werden.

5.3.1 Das Verfahren

Bei dem hierarchischen Shooting wird versucht, die Anzahl der Empfänger zu verringern. Dafür sind zwei Änderungen an dem Progressive-Refinement-Algorithmus nötig. Zuerst ist es nötig eine Baumstruktur einzuführen, wobei jede Fläche der Szene einen solchen Baum, z.B. einen Quadtree, erhält. Soll nun Radiosity empfangen werden, muss dann für den aktuellen Empfänger entschieden werden auf welchem Level des Baumes die Radiosity empfangen werden soll.

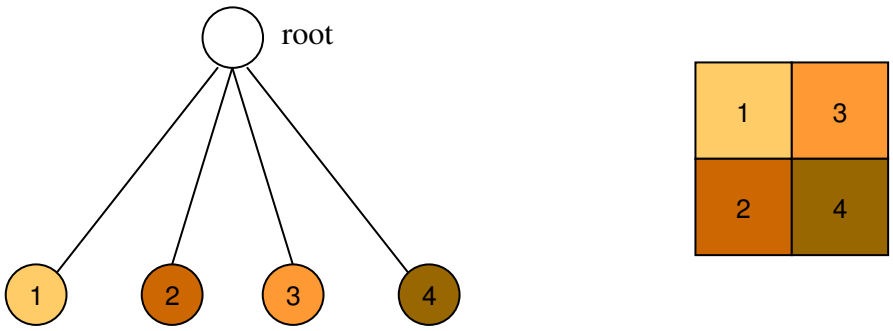


Abbildung 18: Dieses Bild zeigt die Verteilung der Elemente auf die Quadtree-Struktur.

Diese Entscheidung übernimmt die zweite Änderung, das Orakel. Soll Radiosity empfangen werden, wird immer zuerst versucht diese auf dem obersten Level in Empfang zu nehmen. Ist dies nicht möglich, weil der Radiosity-Betrag zu hoch ist, wird die Fläche unterteilt. Nun wird auf dem nächstniedrigeren Level versucht die Radiosity entgegen zu nehmen. Ist dies wieder nicht möglich, wird weiter unterteilt, usw. Hat das Orakel dann entschieden den Radiosity-Empfang auf einem Level zuzulassen, wird der Wert in dem Knoten des Baumes gespeichert. Es ist dabei durchaus möglich, dass der Radiosity-Betrag eines Senders so gering ist, dass das Orakel entscheidet, ihn auf oberster Ebene zuzulassen,

obwohl es bereits Kinder gibt. Damit die Radiosity, die auf höheren Leveln empfangen wurde, auch zu den Blättern gelangt, wird ein Push durchgeführt.

5.3.2 Der Push

Der Push sorgt dafür, dass die Radiosity innerhalb eines Baumes konsistent wird. Wird also Radiosity auf den oberen Ebenen empfangen, „drückt" dieser Vorgang die Radiosity zu den Blättern. Dies geschieht direkt nach dem Empfang der Radiosity, damit im nächsten Schritt, die Auswahl des neuen Senders, jedes Blatt die tatsächlich empfangene Radiosity besitzt.

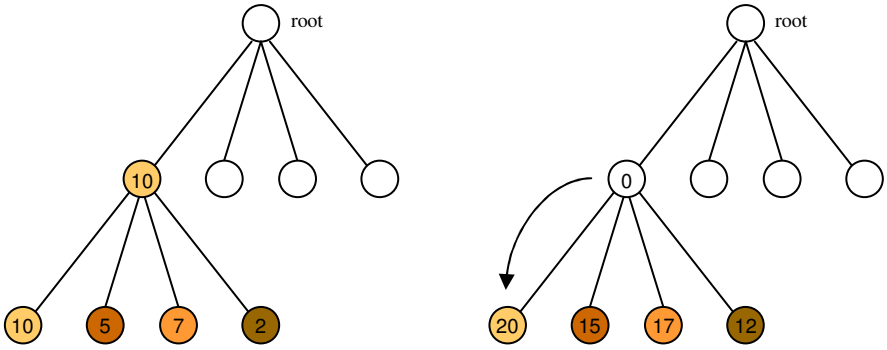


Abbildung 19: Im linken Bild besitzt ein innerer Knoten Radiosity, die er nach dem Push an seine Kinder weitergegeben hat.

5.3.3 Bewertung

Das hierarchische Shooting ist bereits wesentlich schneller als das Progressive Refinement: Wie bereits zu Beginn dieses Abschnitts festgestellt wurde, sind die Beiträge des indirekten Lichts gering im Vergleich zum denen des direkten Licht. Das initale Mesh einer Szene wird aber hauptsächlich von dem direkten Licht bestimmt, welches in der ersten Relaxation verschossen wird. Dieser Aufwand ist im Vergleich zum Progressive Refinement gleich geblieben.
Wird dann das indirekte Licht verschossen wird es dadurch, dass die Beiträge des indirekten Lichts nur gering sind, keine grossen Änderungen mehr am Mesh geben. Abhängig von den Radiosity-Beiträgen sind also höchstens genauso viele Berechnungen nötig wie bei dem Progressive Refinement. Im besten Fall sind hingegen aber nur soviel Berechnungen nötig, wie es Flächen gibt, nämlich genau dann, wenn die Radiosity immer auf dem obersten Level empfangen werden kann.
Ein Nachteil ist, dass auch bei diesem Verfahren die Formfaktoren für jede Berechnung neu bestimmt werden müssen, ausgenommen diese werden gecached.

Da dieses Verfahren nun die Anzahl der Empfänger reduziert hat, stellt sich die Frage warum nicht auch die Anzahl der Sender reduziert werden soll, denn bis jetzt versenden nur die Blätter der Bäume Radiosity, und dies können bei einer hohen Unterteilung des Meshs viele sein.

6. Hierarchisches Radiosity

Nachdem in dem Kapitel vorher bereits verschiedene Radiosity-Verfahren vorgestellt wurden, beschäftigt sich dieses Kapitel mit dem eigentlichen Thema dieser Studienarbeit, dem hierarchischen Radiosity.

6.1 Die Entwicklung des hierarchischen Radiosity

Bei den obigen Verfahren ist deutlich ein roter Faden zu erkennen, wenn man sich die Entwicklung dieser anschaut. Alle eben vorgestellten Verfahren verfolgen dasselbe Ziel: Die Strahlung einer Lichtquelle soll sich möglichst schnell so im Raum verteilen, dass in der nächsten Iteration/Relaxation keine Änderungen mehr auftreten, das System also konvergent ist. Das Erreichen der Konvergenz dauerte bei den ersten Verfahren, den Fullmatrix-Methoden, am längsten, da hier für eine Fläche die Radiosity aller anderen Flächen eingesammelt wurde. In diesen Verfahren wurden die Formfaktoren für jede Beziehung zwischen zwei Flächen in einer Matrix gespeichert. Da aber nicht genug Speicherplatz existierte um $n \cdot (n-1)$ Beziehungen für große Szenen zu speichern, konnten diese Verfahren nur für Szenen mit einer relativ geringen Anzahl von Flächen benutzt werden.
Um Radiosity-Verfahren auch auf größere Szenen anwenden zu können, wurden dann mit Progressive Refinement nur noch die Radiosity der „hellsten" Flächen an alle anderen verteilt, wobei hier keine Formfaktoren zwischengespeichert wurden, sondern für jede Relaxation die Formfaktoren und die Visibilität neu berechnet werden mussten.
Um für ein Shooting-Verfahren die benötigte Zeit bis zur Konvergenz zu reduzieren, wurde dann abhängig von dem Radiosity-Anteil entschieden, auf welchen Level Radiosity empfangen werden sollte. Wenn Radiosity dann auf einer der obersten Ebenen empfangen wurde, konnte so eine ganze Reihe von Formfaktorberechnungen ausgelassen werden. Dieses Verfahren heißt hierarchisches Shooting und ist die Vorstufe zum hierarchischen Radiosity.

Es gab also zuerst eine Entwicklung der Speicherung aller Beziehungen bzw. Formfaktoren, welche dann aber aufgrund des sehr hohen Speicherbedarfs verworfen wurde. Nachfolgende Verfahren versuchten dann so wenig Formfaktoren wie möglich zu berechnen, um den Aufwand zu minimieren. Von einer Speicherung der berechneten Formfaktoren wurde aber abgesehen.
Das hierarchische Radiosity geht hingegen zwei Schritte weiter: Zum Einen wird bei diesem Verfahren nicht nur entschieden, auf welchem Level die Radiosity empfangen wird, sondern auch, auf welchem Level diese versendet wird. Zum Anderen wird für jede Beziehung zwischen zwei Flächen der Formfaktor sowie die Visibilität berechnet, die dann in einem „Link" gespeichert werden. Wird zu einem späteren Zeitpunkt wieder Radiosity zwischen diesen beiden Flächen ausgetauscht, kann auf die bereits errechneten Formfaktoren zurückgegriffen werden.
Es wird in diesem Verfahren also versucht die vorherigen Verfahren zu vereinen, indem so wenig Formfaktoren wie möglich berechnet werden müssen, diese aber konsequent gespeichert werden um das Verfahren in möglichst kurzer Zeit zur Konvergenz zu bringen.

6.2 Das Verfahren

Das hierarchische Radiosity tauscht Radiosity auf mehreren Ebenen aus, wobei das Level für Sender und Empfänger unabhängig festgelegt werden kann. Auf welcher Ebene dies geschieht entscheidet das Orakel (Kapitel 6.6). Die Ebenen können z.B. durch einen Quadtree oder BSP-Tree (Kapitel 6.5) in einer Baumstruktur gehalten werden, wobei jeder Knoten ein Flächenelement der gesamten Fläche repräsentiert. Der Austausch erfolgt dabei über sogenannte Links, wobei ein Link einen Zeiger zum Sender und Empfänger, sowie den errechneten Formfaktor und Visibilitätsfaktor enthält.

```
class Link
{
        …
        Node*         mEmitter;          //Ein Zeiger auf den Sender
        Node*         mReceiver;         //Ein Zeiger auf den Empfänger
        double        mFormFactor;       //Formfaktor zwischen Sender u. Empfänger
        double        mVisible;          //Die Sichtbarkeit
}
```

Zu Beginn des Verfahrens werden durch das Initial Linking (Kapitel 6.3) zuerst nur die Root-Knoten der Flächen miteinander verbunden. Da es sich bei den sogenannten Links aber nur um Gather-Links handelt, d.h. die Radiosity kann über diese nur eingesammelt werden, müssen Links immer in zwei Richtungen erstellt werden.
Der nächste Schritt ist das Refinement (Kapitel 6.4), bei dem das Orakel für jeden Link entscheidet, ob die zu transportierende Radiosity über den aktuellen Link ausgetauscht wird, oder ob ein Link verfeinert (verfeinern, engl. to refine) wird und die Radiosity somit auf einem niedrigeren Level ausgetauscht wird.
Da für jeden Link entschieden wurde auf welcher Ebene Radiosity ausgetauscht werden kann, kommt es nun beim Gathering (Kapitel 6.7), also dem Einsammeln, zum eigentlichen Radiosity-Austausch zwischen den Sendern und Empfängern. Nachdem nun eine Fläche auf verschiedenen Leveln Radiosity eingesammelt haben kann, muss dafür gesorgt werden, dass diese nun konsistent an alle Kinder bzw. Väter weitergegeben werden, dies geschieht beim Push-Pull (Kapitel 6.8).
Alle Schritte des hierarchischen Radiosity, ausgenommen das Initial Linking, werden nun so oft durchgeführt bis das System konvergent ist, also z.B. keine neuen Links mehr erstellt.

6.3 Das Initial Linking

Beim Initial Linking wird eine Schleife über alle n Flächen der Szene angelegt, wobei jede Fläche einmal als Sender und $n - 1$ mal als Empfänger fungieren kann.[11] Der Aufwand für das Initial Linking liegt somit bei $O(n^2)$, wobei für jeden Link der Formfaktor sowie die Visibilität bestimmt werden muss.

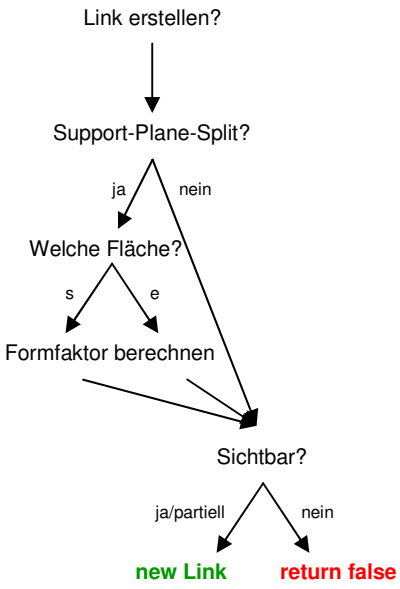


Abbildung 20: Dies ist der grobe Ablauf des Initial Linking.

[11] Eine planare Fläche kann per Definition des Formfaktors keine Strahlung mit sich selbst austauschen, somit braucht auch kein Link erstellt werden.

Der Vorteil des hierarchischen Radiosity im Vergleich zu den anderen Verfahren ist, dass bereits beim Initial Linking entschieden wird, ob eine Fläche an dem Radiosity-Austausch teilnimmt oder nicht. Dies geschieht, indem für jeden sogenannten Top-Level-Link, also die Links zwischen den Root-Knoten der einzelnen Flächen, bereits die Visibilität und die Formfaktoren berechnet werden. Wäre die Visibilität für einen Link null, d.h. Sender und Empfänger können einander nicht sehen, würde kein Link erstellt werden, da nie Radiosity zwischen diesen beiden Flächen ausgetauscht werden könnte. Eine weitere Möglichkeit, um die Anzahl der Links zu Beginn zu reduzieren, ist zu prüfen, ob das Produkt zwischen Formfaktor und der Visibilität so klein ist, dass selbst ein sehr großer Radiosity-Wert nur zu einem kleinen Transport über diesen Link führen würde. So könnte für das Produkt ein gewisser Schwellwert festgelegt werden der, wenn er unterschritten würde, die Erstellung eines neuen Links verhindert. In Verbindung dazu könnte man z.B. die maximale Eigenemission der Szene bestimmen und so sicherstellen, dass wirklich keine größere Radiosity über diesen Link transportiert werden würde.

6.3.1 Verwendetes Verfahren

Bei dem in dieser Studienarbeit implementierten Initial Linking ist es möglich zwischen zwei Modi zu wählen: Bei dem normalen Linking werden die Flächen wie gewohnt miteinander verbunden. Bei dem asymmetrischen Modus wird vor der Erstellung eines Links geprüft, ob ein Support-Plane-Split nötig ist oder nicht. Ist dies der Fall, wird die ursprüngliche Fläche in zwei bzw. drei neue Flächen aufgeteilt. Der Vorteil dieses Modus ist, dass der Empfänger nun in mehrere Empfänger aufgeteilt wird und sich so jeder Teil in einem klar definierten Bereich, im vorderen oder hinteren Halbraum des Senders, befindet. Ebenfalls kann so verhindert werden, dass der Support-Plane-Split mehrmals durchgeführt werden muss und somit das Verfeinern einer Fläche auf Grund eines Support-Plane-Splits nicht mehr nötig ist, was sich in der Gesamtperformanz des Systems niederschlägt.
Hinzu kommt, dass nur so klare Kanten an den Rändern von Objekten entstehen können, falls das System auf einem Quadtree basiert. Im Gegensatz zu einem BSP-Tree, der das beliebige Teilen von Flächen unterstützt, werden bei einer Quadtree-Struktur später Aliasingeffekte an den Stellen entstehen, wo der Sender den Empfänger schneidet. Denn hier wird versucht mit der Struktur des Quadtrees eine Linie anzunähern, was für gute Ergebnisse eine sehr feine Struktur erfordern würde und eine hohe Tiefe des Quadtrees zur Folge hätte.
Bei dem asymmetrischen Modus ist wichtig, dass nur die Flächen geteilt werden, die in unmittelbarem Kontakt zueinander stehen, da ansonsten auch Flächen in unendlicher Entfernung geteilt werden können. Dies wird in der vorliegenden Variante allerdings nicht beachtet.

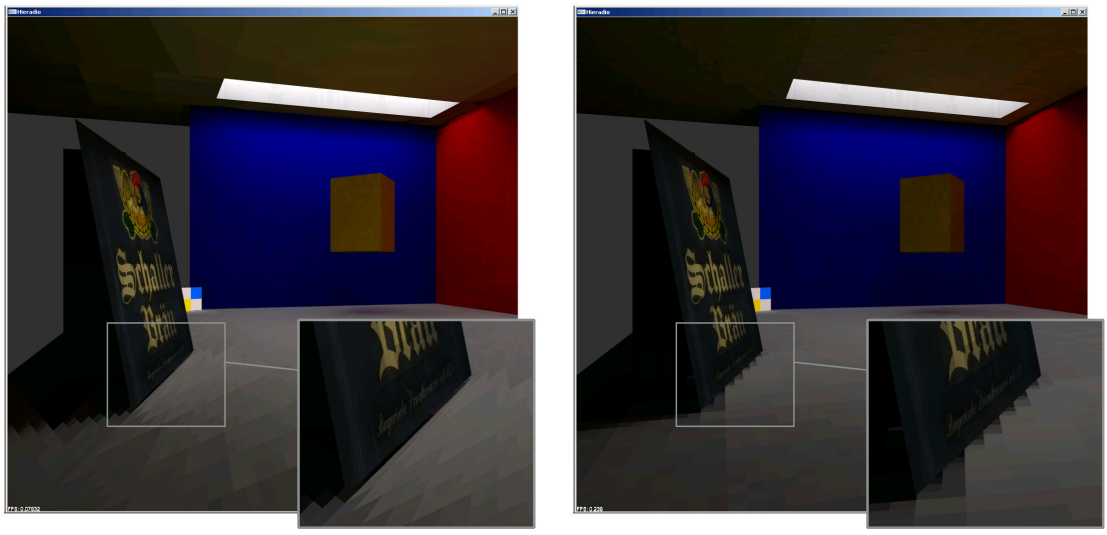


Abbildung 21: Links das Initial Linking mit asymmetrischen Schnitt und rechts die normale Variante.

6.4 Das Refinement

Der Verfeinerungsvorgang ist der zweite Schritt des hierarchischen Radiosity. In diesem werden alle beim Initial Linking erstellten Links vom Orakel überprüft und gegebenenfalls verfeinert. Eine Verfeinerung kann z.B. vorgenommen werden, wenn der Radiosity-Beitrag, der über diesen Link transportiert werden soll, zu hoch ist und somit Artefakte entstehen könnten.
Ist dies der Fall, wird der aktuelle Link gelöscht und eine der beiden Flächen unterteilt. Hierfür werden im Quadtree für die aktuelle Node vier Kinder erzeugt und vier neue Links zu den Kindern dieser Fläche erstellt. Unterteilt werden können sowohl der Sender als auch der Empfänger.

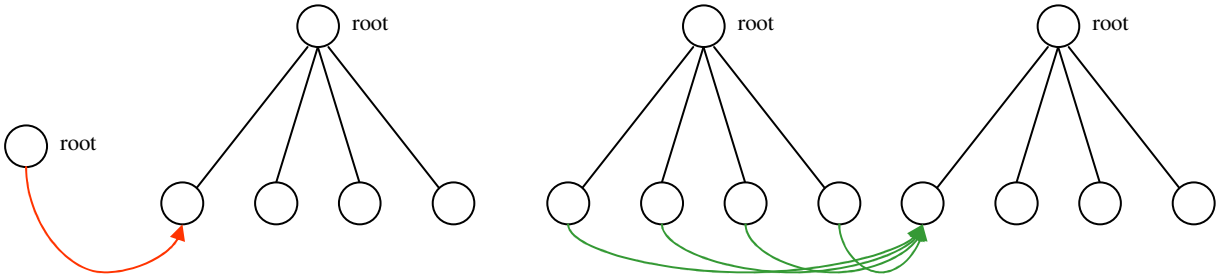


Abbildung 22: Der Link im linken Bild wurde verfeinert, es wurden vier neue Links von den Kindern erstellt.

Üblicherweise werden die Links eines hierarchischen Radiosity-Systems in einer Liste gehalten. Wird dann für eine Fläche entschieden, dass sie unterteilt werden muss, werden die neuen Links hinten an die Liste angehängt. Nun kann man sich entscheiden, ob dieser Schritt so lange ausgeführt wird, bis das Ende der Liste erreicht ist, oder ob immer nur die ursprüngliche Anzahl der Links behandelt wird. Wird solange verfeinert bis keine neuen Links mehr erstellt werden, sind nur wenige Iterationen nötig. Bei der zweiten Variante sind zwar mehrere Iterationen nötig, allerdings hat sich herausgestellt, dass die zweite Variante bei optisch gleichem Ergebnis schneller ist. Differenzen können auftreten, da in der zweiten Variante zwischen dem Verfeinern nochmal Radiosity eingesammelt wird, die dann wiederum die Entscheidung des Orakles beeinflusst.

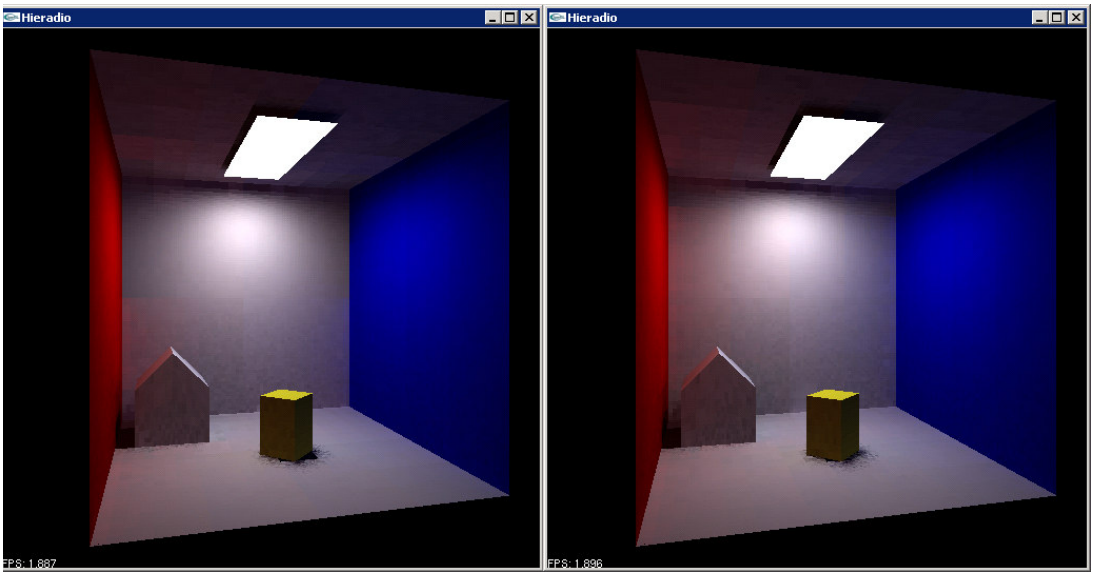


Abbildung 23: Im linken Bild wurde über alle Links iteriert, im rechten bei dem ursprünglich Letzten gestoppt.

Modus	Visibilität	Zeit	Links
Variante 1	fast	59s	72k
Variante 2	fast	49s	74k

Tabelle 2: Diese Tabelle enthält einen Vergleich beider oben vorgestellten Verfeinerungs-Modi.

6.5 Die Baumwahl

Der Baum ist ein elementarer Teil des hierarchischen Radiosity. Alle Unterteilungen, die durch das Orakel angeordnet werden, werden von ihm umgesetzt. Auch das Ergebnis der Simulation wird von ihm beeinflusst, da es seine Blätter sind die gezeichnet werden. Aus diesem Grund sollte auch die Wahl des Baumes wohl überlegt sein.
Zur Auswahl stehen zum Einen die Quadtrees, die hauptsächlich in traditionellen [Hanr91] Systemen eingesetzt wurden, sowie die BSP-Trees, die später in Verbindung mit Discontinuity Meshing [Lisc93] verwendet wurden. Bei der Wahl des Baumes wird somit nicht nur die Art der Datenhaltung, sondern zum Teil auch die Art des Radiosity-Systems festgelegt.

6.5.1 Der Quadtree

Abhängig von der Basis des Radiosity-Systems, Dreieck oder Quad, wird die Wahl entweder auf einen binären Baum oder einen Quadtree fallen.[12] Gerade bei der Verwendung von Quads bietet sich hier die Verwendung eines Quadtrees an, da sich Quads bei einer Unterteilung wieder in Quads zerlegen lassen, wobei jedes neue Quad ein Kind im Quadtree ist.
Allerdings hat der Quadtree Grenzen, die sich sehr schnell auch optisch zeigen. So kann das feste Raster des Quadtrees zu einem Nachteil werden, wenn z.B. eine schräg verlaufende Schattenkante dargestellt werden soll. Dies ist mit einem Quadtree nur durch eine hohe Tiefe und damit einhergehende Anzahl an Flächen möglich und trotzdem bleibt es bei einer Annäherung an die Schräge. Leider werden dabei auch viele unnötige Flächen erstellt, da ja zwangsweise immer vier Kinder erstellt werden müssen.

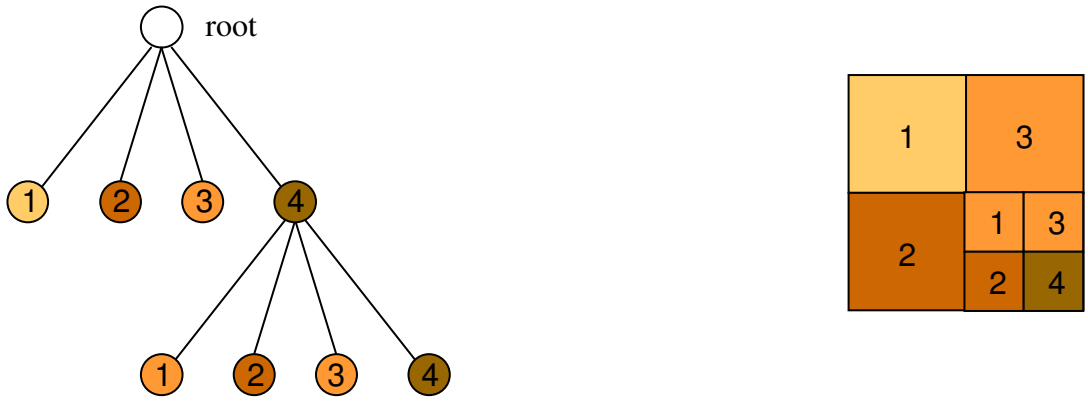


Abbildung 24: Beispiel für eine zweistufige Baumstruktur mit der dazugehörigen Elementstruktur.

[12] Abhängig von der Unterteilungsart eines Dreiecks kann auch für Dreiecke ein Quadtree verwendet werden.

6.5.2 Der BSP-Tree

In späteren Varianten des hierarchischen Radiositiy wurden BSP-Trees in Verbindung mit Discontinuity Mesh verwendet [Lisc93]. Bei Discontinuity Meshs werden die Unstetigkeiten der Radiosity-Funktionen aufgespürt, z.B. Schattenkanten, und explizit durch eine Unterteilung der Fläche an diesen Stellen umgesetzt. Der BSP-Tree (Binary-Space-Partition) ist dabei ein binärer Baum, der relativ zu diesen Grenzen speichert, ob sich Objekte vor oder hinter einer solchen Grenze befinden. Eine solche Grenze wird in der Regel als mathematische Schnittebene gespeichert. Bei mehreren Ebenen wird dies rekursiv fortgeführt, wobei sich die Position mit der Tiefe des Baumes konkretisiert. Die Objekte, die sich in einem Halbraum aufhalten, können dann an den Blättern des Baumes gefunden werden.
Der Nachteil ist, dass bei der Verwendung mehrerer Schnittebenen diese einander wiederum unterteilen und somit sehr schnell ein großer Baum entstehen kann. Auch wenn ein solches System komplexer zu implementieren und auf Grund der schnell anwachsenden Größe des Baumes langsamer ist, bietet es aber bei richtiger Umsetzung ein optisch schöneres und korrektes Resultat [Lisc93].

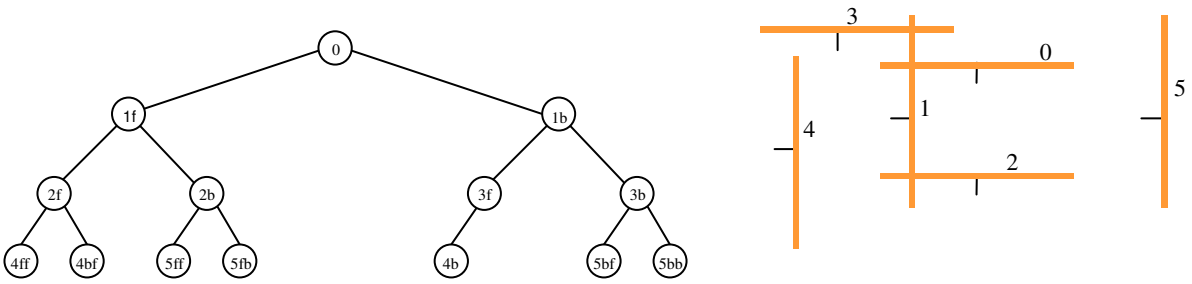


Abbildung 25: Die Aufteilung einer Szene durch Schnittebenen mit dem dazugehörigen BSP-Tree.

6.6 Das Orakel

Seit dem Radiosity-Verfahren existieren, die nicht mehr auf festen Meshstrukturen sondern auf adaptive Unterteilung setzen, existieren auch sogenannte Orakel. Orakel sind eine möglichst einfache aber abstrakte Beschreibung der Radiosity-Funktion, wobei sich folgende Frage stellt:

„Approximiert eine lineare Interpolation zwischen zwei betrachteten Stützstellen die tatsächliche Radiosity-Funktion mit einem tolerierbaren Fehler?“[Müll05]

Das Ziel des Orakles ist es also, anhand einiger lokaler Daten, nämlich die Informationen, die von Sender, Empfänger und dem Link zur Verfügung gestellt werden, zu entscheiden, ob eine weitere Unterteilung nötig ist oder ob die vorhandene Struktur ausreicht, um die Radiosity-Funktion zu approximieren. Mit dem Orakel soll so eine unötige Verfeinerung des Meshs an Stellen, an denen eine Unterteilung keine optische Aufwertung des Reslutats wäre, vermindert werden. Für ein hierarchisches Radiosity-System sollte somit das Wunsch-Orakel so wenig Links, und damit so wenig Unterteilungen, wie nötig erstellen und trotzdem ein Ergebnis liefern, in dem Unstetigkeiten auf Grund von falschen Unterteilungen nicht vorkommen.
In den folgenden Abbildungen sind einige Fehler zu erkennen, die durch falsche Orakelentscheidungen entstehen können.

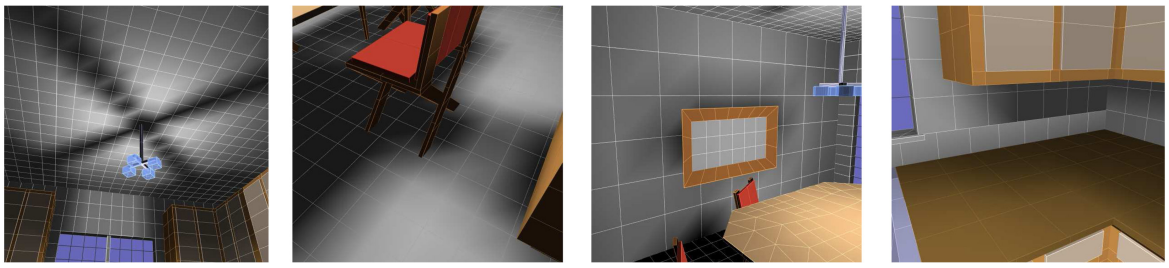


Abbildung 26: Gezackte Schatten, fehlende Schatten, Schattenlöcher und Nähte (v.l.n.r.).[13]

Da das Orakel aber auf lokale Daten beschränkt ist, ist es schwierig ein Orakel zu konzipieren das immer die richtige Entscheidung trifft. Trotzdem gibt es gute Ansätze. Vorgestellt werden hier sowohl das BFA-Orakel [Müll05] als auch das Lischinski-Orakel [Lisc94].

6.6.1 Das BFA-Orakel

Das BFA-Orakel entscheidet anhand von drei Faktoren, *B* für Radiosity, *F* für Formfaktor und *A* für den Flächeninhalt der zu prüfenden Fläche, ob eine Unterteilung vorgenommen werden muss oder nicht. Im Detail laufen die Tests in folgender Reihenfolge ab: Zuerst wird geprüft, ob es sich lohnt weitere Tests auszuführen, dies geschieht über einen Schwellwert, der für das Produkt aus $B \cdot F \cdot A$ festgelegt wird. Im zweiten Schritt kommt es zu den Prüfungen der Visibilität zwischen Sender und Empfänger, wobei es drei Zustände für den Visibilitätsfaktor gibt: totale Verdeckung, partielle Verdeckung und keine Verdeckung.

```
//Totale Verdeckung
if (visible == 0)
	return false;

//Partielle Sichtbarkeit
if (visible < volle Sichtbarkeit && visible > totale Verdeckung)
	return true;

//Enthält Entscheidung welche Fläche geteilt wird
if (visible == volle Sichtbarkeit)
{
	…
}
```

Die nachfolgenden Tests werden nur ausgeführt, wenn eine totale Sichtbarkeit attestiert wurde. In den beiden anderen Fällen wird bei totaler Verdeckung keine weitere Unterteilung angeordnet, wohingegen bei einer partiellen Sichtbarkeit immer unterteilt wird.
Im nächsten Schritt wird geprüft, ob für den Empfänger ein Support-Plane-Split durchgeführt werden muss. Dies ist nötig, um eine klare Trennung zwischen dem vorderen und dem hinteren Halbraum des Sender zu erreichen. Also zwischen dem Teil der Licht direkt empfangen soll und dem Teil der eventuell indirektes Licht empfängt.
Im letzten Teil wird dann der Formfaktor F_{es} mit dem Formfaktor F_{se} verglichen und dann die Fläche unterteilt, deren Formfaktor größer ist, da hier die Fläche am größten ist. In dem Vergleich geht zusätzlich zum Formfaktor noch die minimale Fläche ein, damit eine endlose Verfeinerung verhindert wird. Alternativ könnte hier eine maximale Rekursionstiefe für den Baum angegeben werden.

[13] Das Bild wurde aus Quelle [Bles06] kopiert.

Codeauszug:

```
criterion = Bemitter * ff * Areceiver;

//If this result is to small, the refinement would not be visible
if (criterion < BFA_VISIBLE)
      return 0;

//They can't see eachother
if (visible < EPSILON)
      return 0;

//They can see, but there is an occluder
if (visible < 0.9)
      return 'r';

//To avoid visual artifacts, we divide the receiver if needed
if (needsSupportPlaneSplit(receiver, emitter))
      return 'r';

//Can see, is there to less radiosity?
if (criterion < BFA_MAX)
      return 0;

//Shall the receiver be refined?
if ((Fer >= Fre) && (Areceiver > MinimalArea))
      return 'r';

//Shall the emitter be refined?
if ((Fer < Fre) && (Aemitter > mMinimalArea))
      return 'e';

return false;
```

6.6.1.1 Bewertung

Das BFA-Orakel ist einfach zu implementieren, leider aber sehr schwer zu kontrollieren. So stellt sich zum Beispiel die Frage, wo die Grenze zwischen voller Sichtbarkeit und partieller Sichtbarkeit ist. Eigentlich wäre die Grenze schon erreicht, wenn der Visibilitätsfaktor < 1 ist. Dies führt aber in der Regel dazu, dass es zu sehr vielen unnötigen Unterteilungen kommt. Ein Beispiel: Angenommen, es existieren drei Flächen, die alle von einer gemeinsamen Lichtquelle angestrahlt werden. Die linke und rechte Fläche seien mehrmals unterteilt, wohingegen die dritte Fläche hier nur als Okkluder dient. Es kann also sein, dass der untere Link, bei dem keine volle Sichtbarkeit existiert, unterteilt wird, obwohl die über ihn transportierte Radiosity nur einen geringen Anteil besitzt und so keine Veschattung erzeugen könnte.

Diesem Effekt wird versucht mit dem BFA-Kriterium entgegen zu wirken: Ein Prüfung am Anfang stellt fest, wie groß das Produkt aus $B \cdot F \cdot A$ ist und entscheidet, wieder anhand eines Schwellwertes, ob eine weitere Überprüfung lohnt oder nicht. Das eigentliche Problem entsteht dadurch dass, genau wie beim Visibilitätsfaktor, erst einmal eine Grenze festgelegt werden muss. Gerade dadurch, dass drei Faktoren das Ergebnis beeinflussen wird die Sache verkompliziert.

Angenommen, man entscheidet sich zuerst für einen minimalen Radiosity-Wert. Dann würde der minimale Formfaktor festgelegt, dieser verändert sich bei einer Verfeinerung des Empfängers nicht zwangsweise,[14] und kann somit als konstanter Wert angenähert werden. Der Flächinhalt A wäre hingegen der einzige Wert, der sich verkleinern würde wenn die Fläche unterteilt würde. Dies tut er aber nicht linear sondern quadratisch. Dazu kommt es bei der Festlegung zu folgendem Problem:

[14] Würde z.B. der Formfaktor F_{des} berechnet, wird die Empfänger Fläche als infinitesimal Flächenelement angenommen.

Wird z.B. die Strahlung der Lichtquelle kleiner, kann es sein dass es zu keiner Unterteilung kommt, obwohl bei einer helleren Lichtquelle dieselbe Fläche noch unterteilt wurde. Natürlich kann man sagen, dass eine dunklere Szene nicht so viele Nuancen aufweist wie eine helle Szene. Aber ist es nicht eher so, dass die Radiosity-Funktion identisch ist mit der einer hellen Szene, nur dass die der dunkleren Szene um einen bestimmten Faktor gedrückt ist?

Es wäre daher wünschenswert, dass sowohl die gedrückte Radiosity-Funktion der dunkleren Szene, als auch die Radiosity-Funktion der helleren, gleich behandelt werden und es somit in beiden Szenen zu identischen Unterteilungen kommt. Gerade durch den Einsatz von Tonemappern in diesem Gebiet, kann bei aktiviertem Tonemapper nur schwer auf die Ausgangsstrahlung der Lichtquelle geschlossen werden und so wäre ein ähnliches Ergebnis wünschenswert.

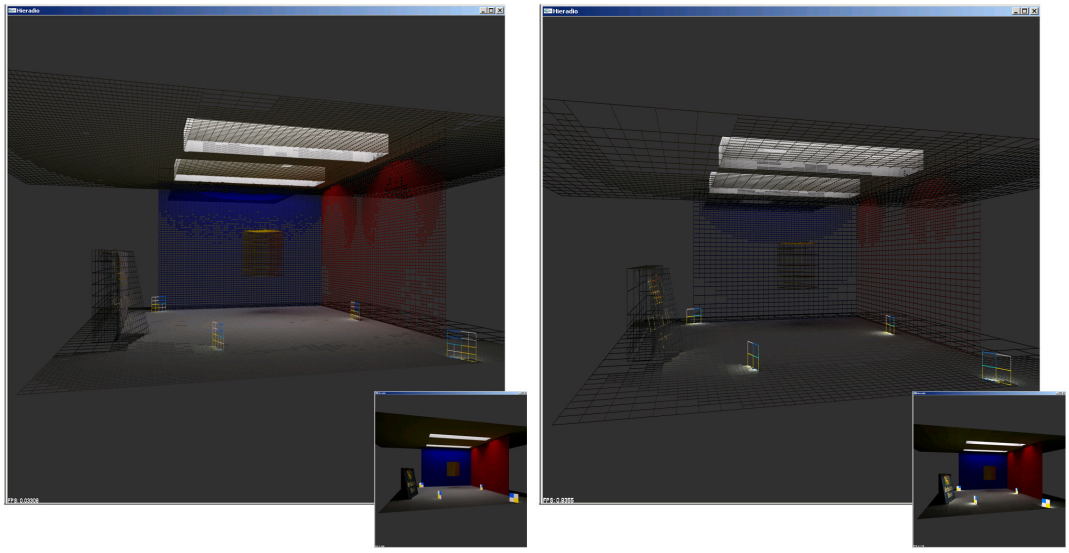


Abbildung 27: Das Mesh eimal mit heller (links) und dunkler (rechts) Lichtquelle, bei aktiviertem Tonemapper.

Dadurch, dass in das BFA-Orakel auch der Flächninhalt eingeht, existiert die obige Problematik nicht nur für eine hellere oder dunklere Szene. Sie existiert auch bei kleineren und größeren Szenen. Wird dieselbe Szene skaliert, bleiben der Formfaktor und die Radiosity konstant, jedoch geht die Fläche quadratisch in die Berechnung ein und sorgt so dafür, dass das kleinste Flächenelement der großen Szene genauso groß ist wie das kleinste Flächenelement der ursprünglichen Szene. Eine besser kontrollierbare Grenze könnte, wenn das Verhalten des BFA-Orakel so nicht erwünscht ist, erreicht werden, indem der Flächeninhalt ins Verhältnis zum Abstand gesetzt wird. Dann können, wie beim Formfaktor, größenunabhängige Werte erhalten werden.

6.6.2 Das Lischinski-Orakel

Anstelle des sehr starren BFA-Orakels, in dem nur anhand von Schwellwerten für die drei Variablen Radiosity, Formfaktor und Fläche entschieden wird, ob unterteilt wird oder nicht, kann beim Lischinski-Orakel ein maximaler Fehler garantiert werden. Die Anfangstests unterscheiden sich nicht von denen des BFA-Orakels. So wird zu erst getestet, ob genug Radiosity über den zu prüfenden Link geschickt wird, um dann im Anschluss, ebenfalls wie beim BFA-Orakel, zu prüfen wie es um die Sichtbarkeit steht, und ob ein Support-Plane-Split vorgenommen werden muss.
Im eigentlichen Kern des Orakels werden dann innerhalb einer Fläche die Minima und Maxima der Radiosity-Funktion abgeschätzt durch die Berechnung von mehreren, im konkreten Fall fünf [Lisc93], Formfaktoren. Diese fünf Formfaktoren werden zu den vier Eckpunkten und dem Mittelpunkt einer Fläche berechnet. Die Sichtbarkeit entscheidet dann darüber, ob nur das Minimum, das Maximum, beide Formfaktoren oder keiner von beiden bestimmt werden müssen.

```
//Es muss kein Formfaktor berechnet werden
if (totale Verdeckung)
{
	Fmin = 0;
	Fmax = 0;
}

//Es muss nur der maximale Formfactor berechnet werden
if (partielle Verdeckung)
{
	Fmin = 0;
	Fmax = findMaxFormFactor();
}

//Fmin liegt auf den Kanten der Fläche und
//Fmax liegt ebenfalls auf den Kanten oder innerhalb der Flächengrenze
if (keine Verdeckung)
{
	Fmin = findMinFormFactor();
	Fmax = findMaxFormFactor();
}
```

Zusätzlich zu minimalem und maximalem Formfaktor werden noch minimale und maximale Radiosity auf der Senderfläche bestimmt. Der Grund für diese zwei Minima und Maxima ist, dass Lischinski et al. festgestellt haben, dass drei Fehlerquellen existieren, die zu einer Abweichung von der eigentlichen Radiosity-Funktion führen: die Radiosity, der Formfaktor und der Visibilitätsfaktor [Lisc94]. Weiter wird gesagt, dass der Fehler der durch den Formfaktor verursacht wird, verkleinert werden kann indem der Empfänger unterteilt wird. Der Fehler, verursacht durch die Radiosity, kann dagegen reduziert werden indem der Sender unterteilt wird. Nachdem festgestellt wurde, dass die vorhandene Abweichung von der Radiosity größer ist als gewünscht, ergibt sich dann folgende Bedingung für die Unterteilung von Sender und Empfänger:

Codeauszug:

```
//Ist der Fehler größer als der gewollte -> Unterteilen
if (Bmax * (ffdelta) > MaximalError)
{
	//Lischinski Kriterium
	if ((ffmax - ffmin) * Bmax >= ffmax * (Bmax - Bmin))
		return 'r';
	else
		return 'e';
}
```

Die Genauigkeit dieses Orakels kann über einen einzigen Faktor gesteuert werden: die maximale Abweichung von der Radiosity-Funktion.

6.6.2.1 Bewertung

Ein grundsätzliches Problem bei den hier vorgestellten Orakeln ist, dass die Visibilitätskomponente zu viel Eigendynamik besitzt. In den beiden Bildern, hier erstellt mit einem Lischinski-Orakle, liegt der maximal vorgegebene Fehler bei 0.15. Bei diesem vorgegebenen maximalen Fehler hätte das linke Bild durch das Orakel nicht so fein unterteilt werden dürfen. Denn trotz der feinen Unterteilung des Meshs treten keine visuellen Veränderungen im rechten Bild auf. Die vorgelagerten Visibilitätsterme sind für diese überflüssigen Unterteilungen verantwortlich.

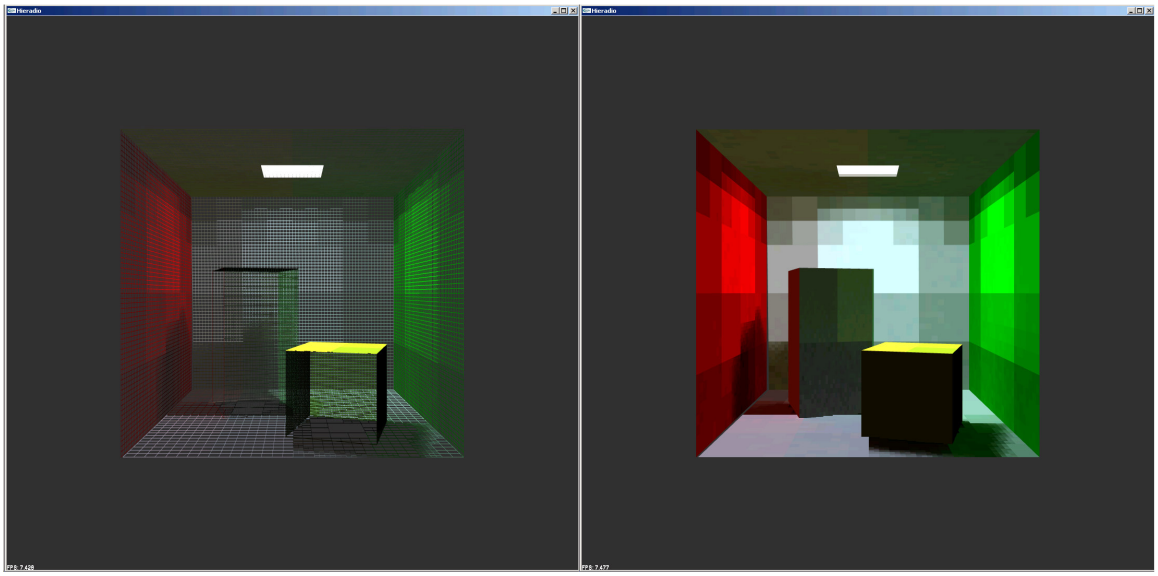


Abbildung 28: Links ist das Mesh des rechten Bildes zu sehen. Die feine Unterteilung hat keine Auswirkungen.

6.7 Das Gathering

Eine Fläche in der hier vorgestellten Studienarbeit, in der es möglich ist Radiosity mit anderen Flächen auszutauschen, besitzt drei Radiosity-Werte. Der erste Wert $B_{selfillumination}$ ist der Initialwert, den eine „Lichtquelle“ beim Starten der Simultaion erhält. Dieser wird oft auch als $B_{emission}$ referenziert. Der zweite Wert B_{unshot} enthält die Radiosity, die von anderen Flächen eingesammelt werden kann, wobei die von allen Flächen eingesammelte Radiosity in der Variablen B_{gather} gespeichert wird.
Das Gathering übernimmt hierbei die Aufgabe in einer einfachen Schleife über alle Links einer Fläche zu laufen, die Radiosity entlang dieser einzusammeln, und in B_{gather} zuspeichern. B_{gather} errechnet sich dabei wie folgt:

$$B_{gather} += \rho_e \cdot B_s \cdot F_{es} \cdot visble \text{ , mit } B_s = B_{unshot}$$

Das Gathering ist auch bei einer hohen Anzahl an Links so performant, dass Ansätze für Optimierungen in diesem Bereich überflüssig sind. So liegt die benötigte Zeit für das Gathering selbst bei 900k Links weit unter einer Sekunde. Erst wenn die Anzahl der Links so groß ist, dass diese nicht mehr in den Hauptspeicher passen und von der Festplatte nachgeladen werden müssen, steigt die benötigte Zeit durch das Nachladen deutlich an.

6.8 Das Push und Pull

Das Push-Pull ist der letzte Schritt in der Hauptschleife des hierarchischen Radiosity. Da es in dem Hierachischen Radiosity möglich ist, dass ein Radiosityaustausch auf verschiedenen Ebenen des Baumes geschieht, kann es sein, dass z.B. nur die kleinsten Elemente einer Fläche, also die Blätter des Baumes, Radiosity eingesammelt haben, die Ebenen darüber jedoch nicht. Sollte nun in einer späteren Iteration Radiosity auf einem der höheren Level ausgetauscht werden, würde der „Sammler“ keine Radiosity vorfinden, obwohl die Fläche welche besitzt, nämlich in den Blättern.
Damit dies nicht passiert, muss es zu einem Ausgleich der Radiosity innerhalb der einzelnen Ebenen des Baumes kommen. Diesen Ausgleich übernimmt dieser letzte Schritt, wobei er eigentlich aus zwei Schritten besteht: dem Push und dem Pull. Zuerst wird die Radiosity B_{gather}, die in den höheren Leveln

eingesammelt wurde, bis zu den Blättern gedrückt (Push), wobei dies für jedes Level oberhalb der Blätter wiederholt wird.

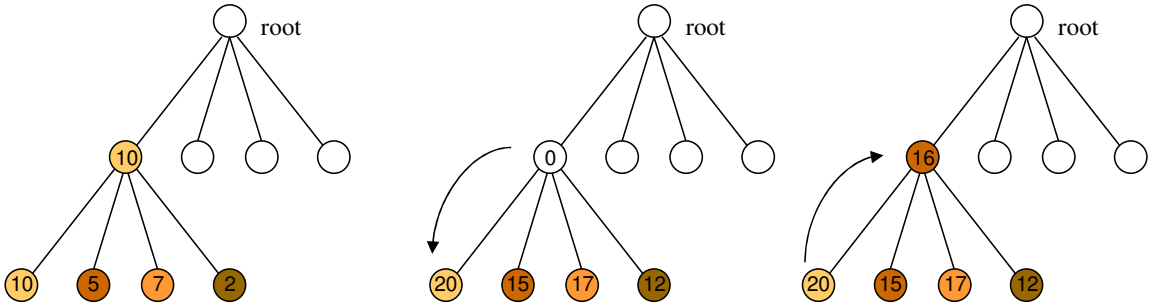


Abbildung 29: Der Ausgangszustand, der Push zu den Kindern und der Pull zu dem gemeinsamen Vater (v.l.n.r).

Die Blätter des Baumes enthalten dann die gesamte Radiosity der Fläche. Im zweiten Schritt wird die Radiosity B_{gather} der Blätter wieder nach oben zurückgeschoben (Pull), bis der Root-Knoten des Baumes erreicht wurde. Dies geschieht dabei flächengewichtet, da es ja durchaus vorkommen kann, dass die Kinder eines Quadtree-Knotens unterschiedlich groß sind und somit die Radiosity einer kleineren Fläche weniger „wert“ ist als die einer großen Fläche. Bei jedem Pull überschreibt dabei die Summe der Kinder-Radiosity die Radiosity der Väter.

$$\text{Push: } B_{child} += B_{father}$$

$$\text{Pull: } B_{father} = \frac{1}{A}\sum B_{child} \cdot A_{child}$$

Auch dieser Schritt benötigt, wie das Gathering, nur wenig Zeit und ist alleine von der Tiefe des Baumes und der Anzahl der Flächen abhängig. Der Aufwand für das Durchlaufen eines Baumes liegt für einen Quadtree bei $O(n^4)$, wobei n die Tiefe des Baumes ist. Eine Reduktion von diesem Aufwand kann jedoch nur durch die Wahl des Baumes geändert werden. Bei mehreren Flächen steigt der Aufwand hingegen nur linear an und ist auch unabhängig von der Baumwahl.

Codeauszug:

```
//Check if this Node has children
if (node->hasChildren() == true)
{
	Bpush = node->getPatch()->getGatheredRadiosity();

	for (short i = 0; i < 4; i++)
	{
		child = node->getChild(i);

		//Add own radiosity to the childs'
		child->getPatch()->gatherRadiosity(&Bpush);
		Bpull += pushPull(child);
	}
	//Set pulled Radiosity from the children to the current Node
	node->getPatch()->setGatheredRadiosity(&Bpull);
	node->getPatch()->transformGatheredRadiosity();
}
```

```
else
{
	//Set Radiosity to pull up to the father
	Bpull = node->getPatch()->getGatheredRadiosity();

	//Transform the gathered Radiosity to unshot and absolut
	node->getPatch()->transformGatheredRadiosity();
}

//The root-Node has no father so he return his Radiosity only
if (node->isRoot() == false)
{
	//Get area of father and child
	Achild  = node->getPatch()->getArea();
	Afather = node->getFather()->getPatch()->getArea();

	return Bpull * Achild / Afather;
}
return Bpull;
```

6.8.1 Bewertung

Auch das Push-Pull bietet noch Möglichkeiten für Optimierungen. Angenommen, eine Fläche würde einmal unterteilt werden und bestehe daher aus vier Teilen. Das Element der unteren linken Ecke hätte nun sehr viel Radiosity eingesammelt, wohingegen die anderen Elemente keine Radiosity eingesammelt hätten. Würde nun Push-Pull ausgeführt würde der Vater zwar Radiosity erhalten, die benachbarten Flächen allerdings nicht. Es entsteht eine Unstetigkeit.
Da es sein könnte, dass das Element der unteren linken Ecke mit vielen anderen Flächen verlinkt ist, wäre es möglich, dass ein Link der viel Radiosity transportiert, dafür sorgen könnte, dass das untere linke Element unterteilt würde. Die verbleibenden Links zu diesem Knoten würden einzeln nur wenig Radiosity transportieren, aber in der Summe wäre es trotzdem genug um sichtbar zu sein.
Durch den dann ausgeführten Push-Pull würde zwar die Radiosity auch an die benachbarten Knoten gegeben, aber dadurch, dass sehr viel Radiosity auf einem Zwischenlevel eingesammelt würde, würden die Kinder dieses Levels wesentlich mehr Radiosity erhalten als angrenzende Elemente, und so könnte es zu deutlichen Unstetigkeiten im Resultat kommen.[15]

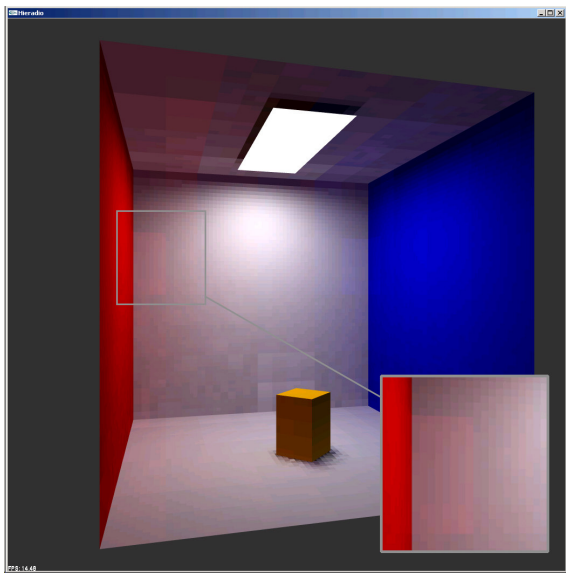


Abbildung 30: Dieser Bereich hat auf einem höheren Level Radiosity empfangen, die bis zu den Kinder weitegegeben wurde.

[15] Es werden immer nur die Blätter gezeichnet, so können sich auch Unstetigkeiten in höheren Leveln negativ auf das Ergebnis auswirken.

Um solche Unstetigkeiten zu verhindern, könnte die Radiosity, die an die Kinder gegeben wird, zum Teil auch an benachbarte Knoten übertragen werden, so dass auch die Kinder eines benachbarten Knoten etwas Radiosity empfangen und somit die Unstetigkeit abgeschwächt wird. Dies würde abhängig von der Tiefe des Baumes allerdings zu einer starken Verlangsamung des ursprünglichen Algorithmus führen, da die direkten Nachbarn erst einmal bestimmt werden müssten.

7. Hierarchisches Radiosity mit Texturen

Die Szenen, die durch ein klassisches hierarchisches Radiosity-System dargestellt werden können, können durchaus komplex sein und erscheinen durch die natürliche Beleuchtung durchaus realistisch. Aber leider sind die Möglichkeiten, realistische Szenen nur mit Flächen einheitlicher Farbe zu modellieren, doch sehr beschränkt. So wäre eine Fläche, auf der ein einfacher Verlauf stattfinden würde in einem klassischen Radiosity-System schwer möglich, abgesehen von Bildern die z.B. Wände schmücken. Natürlich könnten diese einfach nachträglich auf die Flächen „aufgetragen" werden, so dass sie in dem Resultat einer Radiosity-Simulation zu sehen wären. Ein wirklicher Strahlungsaustausch mit diesen Bildern wäre dadurch aber trotzdem nicht geschehen.

Die Erweiterung des hierarchischen Radiosity um Texturen macht es jedoch möglich, dass ein Strahlungsaustausch nicht nur zwischen Flächen, sondern auch zwischen Texturen, sowie Texturen und Flächen möglich ist. Somit kann z.B. eine dunkle Textur, die einen hellen farbigen Schriftzug enthält, als Strahlungsquelle, z.B. für Leuchtreklame, genutzt werden. Auch wäre es möglich Texturen mit transparentem Kanal, oder aber transparenter Farbe zu verwenden, die dann für Flächen die in dem transparenten Bereich einer Textur liegen, dafür sorgen, dass diese nicht gezeichnet werden. So könnten auch mit nur einfachen Primitiven, wie einem Dreieck oder einem Quad, runde Flächen erreicht werden.[16]

7.1 Bestimmung der Texturkoordinaten

Damit Texturen für den Strahlungsaustausch verwendet werden können, ist es nötig eine Reihe von Randbedingungen zu bedenken. Zuerst muss es möglich sein Texturen zu laden, wobei sich hier die Frage stellt, welche Formate denn mit dem System verwendet werden sollen. Mit dem vorliegenden System können Formate wie .pgm und .ppm verwendet werden, da sowohl die Header- als auch die Bildinformationen unverschlüsselt vorliegen und Texturen somit leicht eingelesen werden können. Der Markt an Konvertern bietet außerdem die Möglichkeit, die Konvertierung eines beliebigen Bildformats in ein .ppm- bzw. .pgm-Bild vorzunehmen, so dass durch die Wahl dieses Formats keine reelle Einschränkung existiert.
Im nächsten Schritt muss es möglich sein Texturen auf Oberflächen zu legen. Dies geschieht durch das Hinzufügen von n Texturkoordinaten, wobei n die Anzahl der Eckpunkte einer beliebigen Fläche ist. Dahingehend soll eine einfache, oder am besten automatische, Bestimmung für Texturkoordinaten möglich sein. Eine Funktion, die dies für eine beliebige Fläche in ihrem initalen Zustand tut, also ohne dass die Fläche unterteilt wurde, kann in der Online-Dokumentation [Wiki] gefunden werden.
Es reicht dabei nicht, nur die Texturkoordinaten für das initale Mesh zu besitzen, da ja immer nur die Blätter eines Baumes gezeichnet werden. Dadurch, dass die Fläche unterteilt werden kann, müssen auch für die Kinder des Root-Knotens entsprechende Texturkoordinaten existieren, so dass die Textur nach einer beliebigen Anzahl an Unterteilungen nicht verzerrt dargestellt wird. Da für die Unterteilung einer Fläche aber jeweils zwei Eckpunkte gemittelt werden, erhält man somit vier neue Punkte, mit denen die vier neuen Flächen erstellt werden können. Selbiges geschieht auch mit den Texturkoordinaten, so dass für die neuen Flächen ebenfalls korrekte Texturkoordinaten errechnet werden, wenn die ursprünglichen korrekt waren.

[16] Für wirkliche Rundungen müssten allerdings sehr hochauflösende Texturen verwendet werden.

7.2 Beschränkung auf quadratische Texturen

Grundsätzlich können Texturen in OpenGL nur verwendet werden, wenn deren Größe 2^n beträgt. Auch das System dieser Studienarbeit leidet noch unter dieser Beschränkung. Eine solche Beschränkung kann aber umgangen werden, indem das Array, in dem die Textur abgelegt wird, immer nach oben hin aufgerundet wird, z.B. 400x300 Textur wird in ein 512x512-Array gespeichert. Die nicht benötigten Pixel, unterhalb und rechts des Bildes, können dann mit nullen aufgefüllt werden.
Damit trotzdem eine korrekte Darstellung ohne schwarze Pixel erfolgt, reicht es, die Texturkoordinaten nur für den gewünschten Bereich zu berechnen. Dies reicht aus, da die vorliegenden UV-Koordinaten immer gemittelt werden, und so die „Hülle“ der ursprünglichen Koordinaten nicht verlassen wird.
Es ist hierfür sinnvoll, dass einmal die Größe der Textur und einmal die Größe des Arrays gespeichert wird.

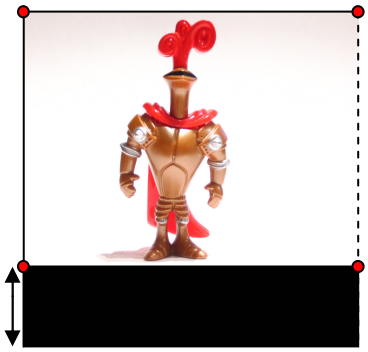


Abbildung 31: Die Texturgröße wird auf die nächsthöhere 2^n-Zahl aufgerundet und der leere Bereich mit nullen gefüllt. Die Texturkoordinaten, hier rote Punkte, mappen nur den eigentlichen Bildbereich.

7.3 Bestimmung der Radiosity

Nachdem nun die Texturkoordinaten bestimmt wurden muss noch die Farbe für eine Fläche ermittelt werden, die ausgestrahlt wird, falls eine texturierte Fläche z.B. eine Eigenemission besitzt. Hierfür müssen im Prinzip alle Pixel der Textur, die sich in der Fläche befinden die von den Texturkoordinaten aufgespannt wird, gemittelt werden. Aber auch hier steckt der Teufel im Detail. Kann für rechteckige Flächen einfach die Kantenlänge des Rechtecks und ein Bezugspunkt bestimmt werden, um alle betroffenen Pixel zu finden, reicht dies für beliebige Flächen nicht aus. Da das hier vorgestellte System es zulässt beliebige konvexe Quads zu erstellen, müssen auch für die daraus resultierenden Texturkoordinaten die betroffenen Pixel gefunden werden. Dies geschieht im konkreten Fall über das „Inverse-Scanline-Verfahren“[Wiki]. Müssen bei Scanline-Verfahren aus Objektkoordinaten die Pixel im Bildschirm gefunden werden, die gefüllt werden sollen, arbeitet das Inverse-Scanline-Verfahren genau umgekehrt. Hierbei müssen für die vorliegenden Texturkoordinaten zwar ebenfalls die betroffenen Pixel gefunden werden, allerdings werden diese nicht gefüllt, sondern gelesen.
Durch dieses Verfahren kann für jede konvexe Fläche eine Farbe bestimmt werden, die anschließend ausgesendet werden kann.

Die Farbe einer Fläche gilt jedoch immer nur für einen Knoten im Baum. Muss eine Fläche unterteilt werden, muss für jedes Element eine neue Farbe bestimmt werden. Dies ist notwendig, da im Prinzip vom Allgemeinen, dem Mittelwert eines gesamten Bildes, zum Speziellen, dem Mittelwert eines Ausschnittes des Bildes, verfahren wird. Angenommen, der Root-Knoten wird unterteilt nachdem er bereits Radiosity eingesammelt hat. Die Frage die sich bei der Unterteilung stellt ist, wieviel Radiosity bekommen denn die Kinder? Die Kinder bekommen, wie bei dem Push-Pull die gesamte Radiosity, aber eigentlich nur die Luminanz dieser. Da in dem vorliegenden System aber mit dem rgb-

Farbsystem gearbeitet wurde, musste jedes Mal der Mittelwert gebildet werden um einen ungefähren Eindruck der Helligkeit zu erfahren. Besser wäre es gewesen, intern mit dem xyz-Farbsystem zu arbeiten, da sich in diesem Farbsystem die Luminanz immer in der y-Komponente befindet.

Codeauszug:

```
Vertex dimension = Vertex(mWidth, mHeight, 0);

//Copy vector, there shall be no changes
std::vector<Vertex> coordinates = *textureCoordinates;

//Change coords. from [0,1] to [mWidth,mHeight]
for (int i = 0; i < size; i++)
	coordinates[i] = coordinates[i] * dimension;

//Determine min(y) and max(y) and init vectors
std::vector<int> height = calcVerticalSpan(coordinates);
std::vector<int> width;

//Read all pixel lying within the Texture-Coordinates: min = 0, max = 1
for (int row = height[min]; row < height[max]; row++)
{
	//Determine min(x) and max(x) of one row
	width = calcHorizontalSpan(coordinates, row);

	//Check if intersection was sucessfull
	if (width.size() != 2)
		continue;

	//Switch to datafile coord-system: lower left to upper left corner
	y = (mHeight - 1) - row;

	//Read pixel and add to mean: min = 0, max = 1
	for (int x = width[min]; x < width[max]; x++)
		mean += mTextureData[mWidth * y + x];

	//Remeber how many pixels were parsed
	numberOfPixels += width[max] - width[min];
}
mean = mean / numberOfPixels;
return mean;
```

7.4 Die Darstellung einer texturierten Fläche

Wurden die Texturkoordinaten und Farbe für eine Fläche bestimmt, kann die Strahlung wie bei normalen Flächen mit den in Kapitel 6 vorgestellten Schritten ausgetauscht werden. Beim Zeichnen muss dann nur noch darauf geachtet werden, dass die eingesammelte Radiosity als Farbe verwendet wird. Es gibt dabei zwei Möglichkeiten, wie die texturierten Flächen gezeichnet werden können.
Die erste Möglichkeit ist, dass wie bei einer normalen Fläche die Blätter des Baumes ermittelt werden, und dann jedes Blatt mit seinen eigenen Texturkoordinaten und seiner eigenen Farbe gezeichnet wird. Dies ist eine relativ aufwendige Methode, da für jedes Blatt die *draw()*-Methode zum Zeichnen aufgerufen werden muss. Ebenso kann es wie bei dem Zeichnen von normalen Flächen dazu kommen, dass auf Grund von Rechenungenauigkeiten Lücken zwischen zwei Elementen entstehen.
Mit der zweiten Methode können diese Probleme ganz vermieden werden, wobei diese Methode erst nach dem vollständigen Strahlungsaustausch eingesetzt werden kann. Hierbei wird die Farbe eines jeden Blattes in den Bereich der Texturkoordinaten geschrieben. Damit die Originaltextur nicht überschrieben wird, wird eine zweite sogenannte Maskentextur zum Speichern dieser Werte verwendet. Sind alle Werte in dieser Maske gespeichert, reicht es aus, diese Maske mit der Originaltextur zu kombinieren und dann den Root-Knoten des Baumes mit der Kombination (Blend) aus Masken- und Originaltextur zu zeichnen.

Durch das zweite Verfahren kann eine wesentlich höhere Zeichengeschwindigkeit erreicht werden, da immer noch der Root-Knoten anstelle aller Blätter gezeichnet werden muss. Der zweite Vorteil ist, dass durch das Zeichnen von nur noch einer Fläche keine Löcher mehr im Mesh entstehen können.

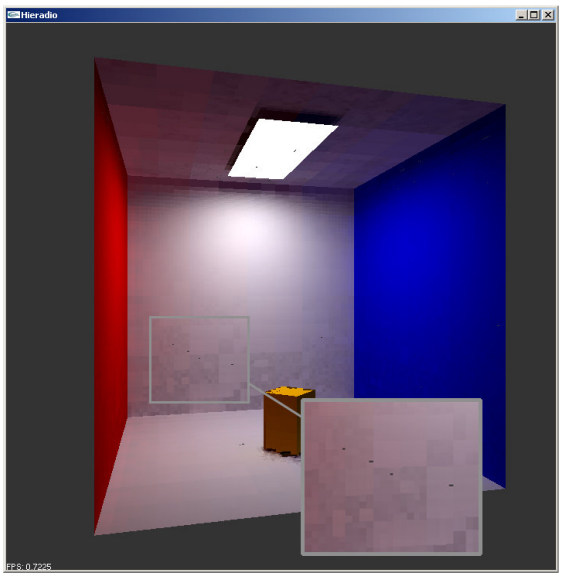


Abbildung 32: Die Lücken enstehen zwischen zwei aneinanderliegenden Flächen.[17]

7.5 Probleme

Das einzige Unschöne an dieser Erweiterung ist, dass die texturierten Flächen sehr stark unterteilt werden müssen, da ansonsten eine „zweite Textur“, die wesentlich niedriger aufgelöst ist, über der eigentlichen Textur liegt. Dies ließe sich jedoch nur verhindern, indem anstatt der eigenen Farbe ausschließlich die Luminanz gezeichnet wird. Allerdings würde dadurch auch andere eingesammelte Radiosity keine farbliche Auswirkung auf die Textur zeigen, was auch unerwünscht wäre. Eine Lösung des Problems könnte wohl nur die strikte Trennung der eigenen Farbe von der eingesammelten Farbe bedeuten. Aber selbst dabei würde sich dann die Frage stellen, wie eine Textur mit Eigenemission dann dargestellt werden soll.

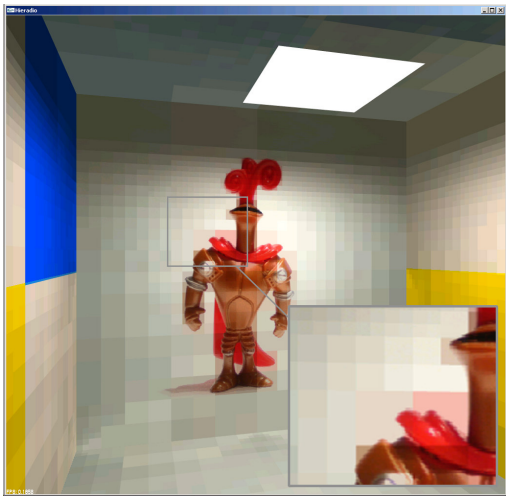


Abbildung 33: Diese Blöcke können bei zu geringer Meshauflösung auftreten.

[17] Dieses Bild konnte nur von einem P3 550 mit einer ATI Mobility Grafikkarte 8MB erzeugt werden. Auf anderen Systemen sind diese Probleme nicht aufgetreten.

8. Vorgehensweise

Dieses Kapitel enthält einen Abriss über die Vorgehensweise zur Erstellung dieser Studienarbeit. Insbesondere wird Wert darauf gelegt, ob der aufgestellte Zeitplan eingehalten wurde und in welche Abschnitte sich dieser letztlich gliedert.

8.1 Die Planung

Dieser Zeitplan zeigt eine grobe Übersicht über den Ablauf der einzelnen Phasen im Rahmen der vorliegenden Studienarbeit. Für die beiden ersten Phasen „Wissensaufbau“ und „Softwaretechnischer Entwurf“ wurden jeweils zwei Wochen angesetzt. Die dritte Phase, die für die Implementation des Grundsystems vorgesehen war, umfasste sechs Wochen. Des Weiteren wurden für die Phasen „Implementierung II“, „Implementierung III“ und die „Ausarbeitung“ jeweils 4 Wochen eingeplant.
Die Ausarbeitungsphase war die letzte, und ist mit einem zeitlichen Ansatz von vier Wochen großzügig gewählt, so dass sie als zeitlicher Puffer dienen konnte.
Allerdings war bereits nach dem Wissensaufbau klar, dass eine Phase alleine für den Schattenwurf überflüssig war. Der Schattenwurf wurde bei der Implementation des Grundsystems, welches auch die Umsetzung von Visibilitätstests beinhaltete, die letztlich für die Erzeugung von Schatten zuständig sind, mit umgesetzt. Aus diesem Grund wurde die Zeit der zweiten Implementierungs-Phase dazu genutzt sich intensiver mit der Beschleunigung der Visibilitätstests zu befassen (Siehe Kapitel 4).

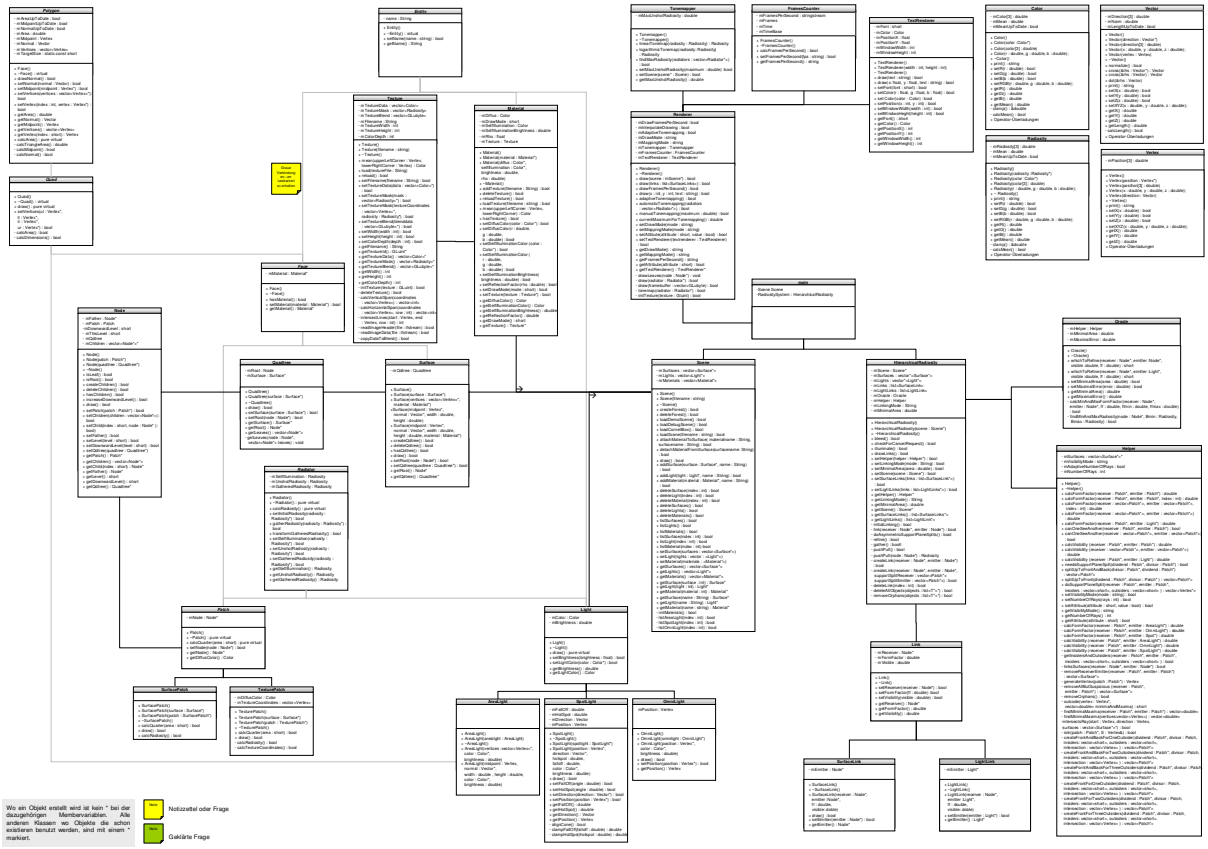
Themen	Juni	Juli	August	Sept.	Okt.	Nov.
Wissensaufbau						
Softwaretechnischer Entwurf						
Implementierung I - Grundsystem						
Implementierung II - Schattenwurf						
Implementierung III - Texturen						
Ausarbeitung						

Tabelle 3: Diese Tabelle zeigt die für diese Studienarbeit geplanten Phasen.

8.2 Die Umsetzung

Die Planung der Studienarbeit wurde wie im Zeitplan vorgesehen eingehalten. Die Phasen für den Wissensaufbau und den softwaretechnischen Entwurf des Systems konnten am 10. Juli 2006 abgeschlossen werden. In dieser Zeit wurde für ein besseres Verständnis zuerst ein grobes Klassendiagramm des hierarchischen Radiosity erstellt. Dieses Klassendiagramm enthielt in der ersten Version nur die wichtigsten Klassen und Funktionen des Systems und bestand bereits aus 19 Klassen. Um das Verständnis für das System zu schärfen und die Beziehungen bzw. Interaktionen zwischen den einzelnen Klassen zu verdeutlichen, wurde im nächsten Schritt ein natürlichsprachlicher Ablaufplan für das hierarchischen Radiosity erstellt. Dieser Ablaufplan umfasste sowohl die einzelnen Schritte zur Initialisierung des Systems, als auch die für das hierarchische Radiosity typischen Iterationsschritte: die Verfeinerung, das Einsammeln der Radiosity und der Radiosity-Ausgleich innerhalb einer Fläche.
Mit Hilfe dieser Ablaufbeschreibung war es möglich, die Zusammenhänge zwischen den einzelnen Klassen so sehr zu vedeutlichen, dass Fehler in der ursprünglichen Version des Klassendiagramms aufgedeckt und behoben werden konnten.
Nach mehreren Änderungen war das Klassendiagramm[18] soweit ausgereift, dass zum 10. Juli 2006 mit der Implementation des Systems begonnen werden konnte.

[18] Eine größere Version kann als .pdf-Datei auf der beiliegenden CD oder im [Wiki] gefunden werden.



8.2.1 Die Vorbereitung

Die Implementation des Systems wurde ebenfalls in mehrere Phasen unterteilt, so dass stets eine Aussage zum aktuellen Status getroffen werden konnte. Diese Phasen sind die Implementation der Hilfsklassen, der Verwaltung, des hierarchischen Radiosity und des texturbasierten Radiosity.
So wurden zum Beginn der Implementationsphase die Programmrahmen erstellt, die dem Klassendiagramm entnommen werden konnten. Dies umfasste das Anlegen der Dateien in der Entwicklungsumgebung, sowie die Implementation der Header-Dateien mit allgemeinen Kommentaren.
Im zweiten Schritt wurden dann die Kommentare um doxygen-Kommentare erweitert, wobei die erste und zweite Phase am 19. Juli 2006 beendet werden konnten.
Es folgte dann die Implementierung der Hilfsklassen, also den Klassen die von sehr vielen anderen Klassen benutzt werden und die Basis des Systems darstellen, wie z.B. Vertex oder Color. Diese wurden gleichzeitig mit den Dummys erstellt. Bei der Erstellung der sogenannten Dummys wurden die Klassendateien, die zur Verwaltung gehören, mit den Funktionsrümpfen und Rückgabetypen gefüllt. Diese beiden Phasen wurden parallel bearbeitet und konnten am 7. August 2006 abgeschlossen werden. Im Anschluss folgte dann die Implementation der Verwaltung, welche die Szenendaten und Materialien bzw. Texturen enthält. Nach einem Monat war auch diese Phase am 9. September 2006 beendet.

8.2.2 Die Implementation

Nachdem die Implementation der Verwaltung abgeschlossen war, konnten Szenendaten erstellt werden, die die Entwicklung des hierarchischen Radiosity durchgängig unterstützten. Denn nur durch diese Vorgehensweise konnten die einzelnen Funktionen des Radiosity-Systems getestet und verifiziert werden.

So konnten am 9. September alle notwendigen Dummys für das hierarchische Radiosity erstellt, und folglich am 11. September 2006 mit der Erstellung des Radiosity-Systems begonnen werden. Nach eineinhalb Monaten war auch dieses am 22.Oktober 2006 fertiggestellt. Am Ende dieser Phase wurde das System noch auf Fehler untersucht und beschleunigt.
Zwei Wochen später konnte die vorletzte Phase und ein wichtiger Teil des Systems fertiggestellt werden: das texturbasierte Radiosity. In der Zeit vom 3. bis zum 21. November wurden dann noch verschiedene kleinere Fehler behoben und die Dokumentation vervollständigt.

Diese Aufteilung in die vier allgemeinen Phasen kann auch dem endgültigen Klassendiagramm entnommen werden, wobei die linke Hälfte des Systems die gesamte Verwaltung umfasst die das Radiosity-System mit Daten versorgt. Der rechte Teil enthält alle Klassen die zum hierarchischen Radiosity gehören und für die Funktion notwendig sind.

Themen	Juli	August	Sept.	Okt.	Nov.
Erstellung der Programmrahmen					
Dokumentation der Programmrahmen					
Hilfsklassen implementieren					
Erstellung der Dummys für die Verwaltung					
Implementation der Verwaltung					
Erstellung der Dummys für das hierarchische Radiosity					
Implementation des hierarchischen Radiosity					
Testen des Radiosity Systems					
Beschleunigen und Debugging					
Implementation des texturbasierten Radiosity					
Debugging					

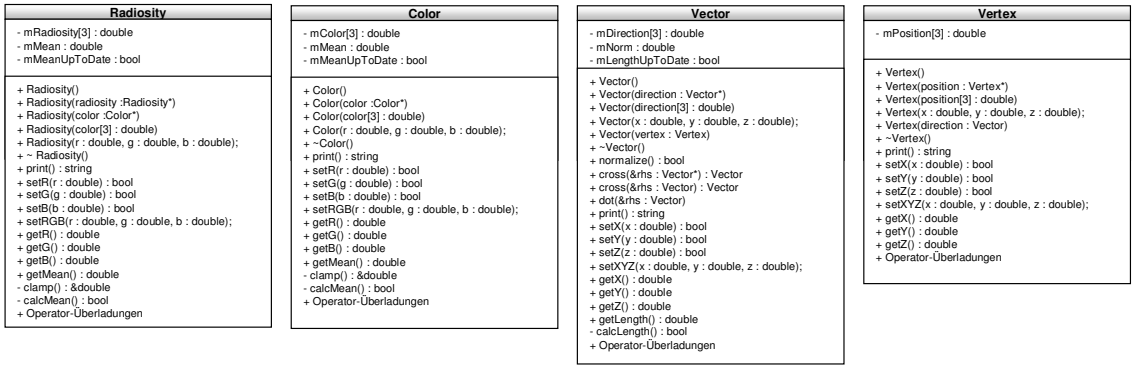
Tabelle 4: Diese Tabelle zeigt den Verlauf der Studienarbeit in detaillierten Phasen.

9. Die Bestandteile des Systems

Das gesamte System besteht aus insgesamt vier Teilen: den Hilfsklassen, den Zusatzklassen wie z.B. einem Tonemapper, der Verwaltung und dem hierarchischen Radiosity. Hierbei bilden die Hilfsklassen die Basis des Systems, so dass ohne diese Hilfsklassen weder die Verwaltung noch das Radiosity-System betrieben werden können. Die Verwaltung und das Radiosity-System sind hingegen modular aufgebaut. Weitere Zusatzklassen sind ebenfalls unabhängig von allen anderen, wobei diese Erweiterungen des vorhandenen Systems darstellen, welches auch ohne jene voll funktionstüchtig ist.

9.1 Die Basisklassen

Die Basisklassen, oder auch Hilfsklassen, bilden die Grundlage des Systems. Zu diesen gehören die Klassen Color, Vertex, Vector, sowie die Klasse Radiosity. Die ersten drei Klassen waren von Beginn an vorhanden, die Klasse Radiosity hingegen wurde erst im Laufe des Entwicklungsprozesses hinzugefügt. Die Klasse Color wird nach außen hin genutzt, wo eine Farbe definiert werden muss, z.B. bei Materialien, und so lässt sie lediglich Werte zwischen null und eins zu.
Die Klasse Radiosity hingegen ist ein internes Datenformat. So wird die angegebene Materialfarbe in Radiosity konvertiert, da nur diese für den Strahlungsaustausch benutzt werden kann. So lässt diese auch Werte von null bis „unendlich“ zu.



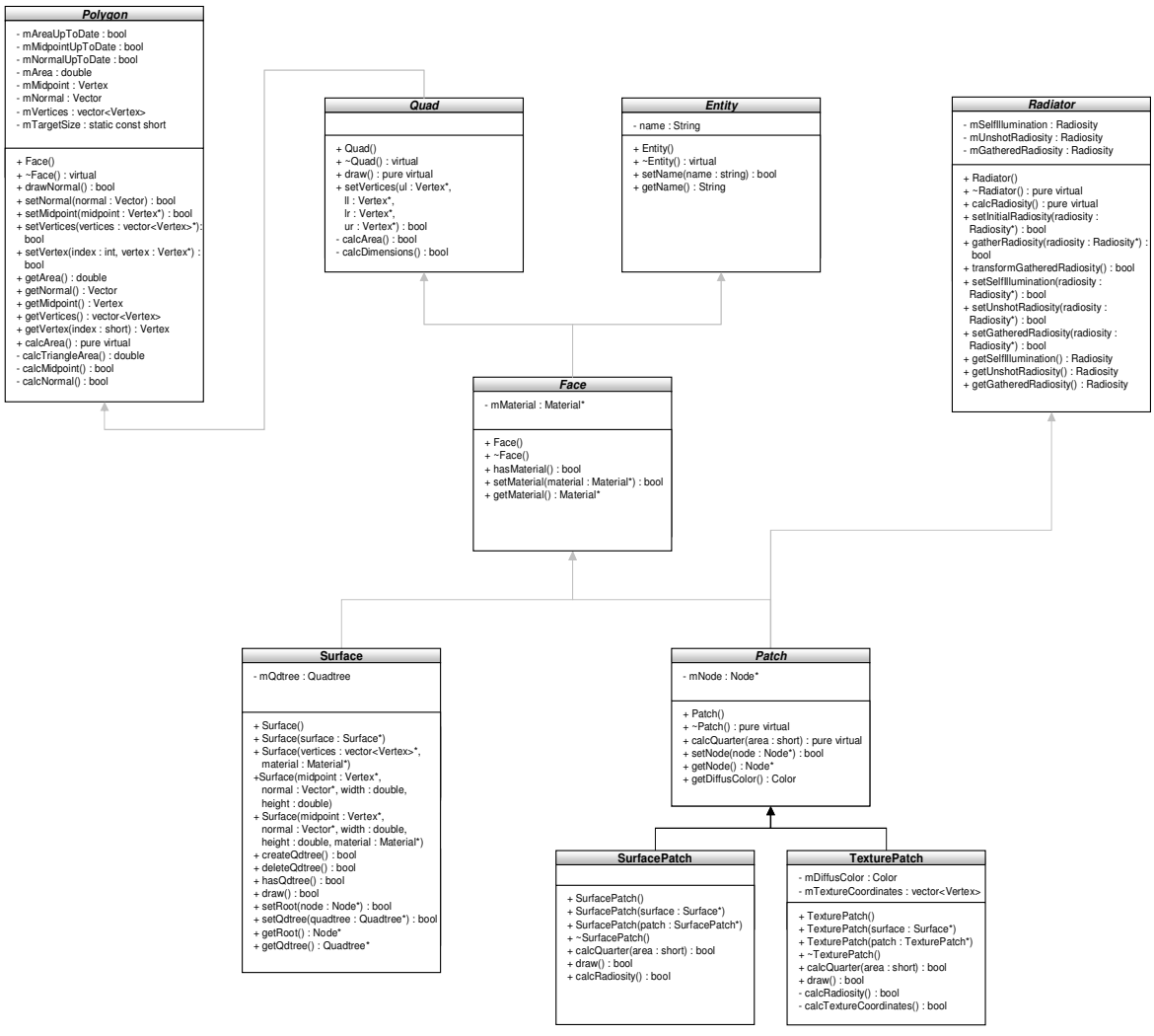
9.2 Die Verwaltung im Detail

Die Verwaltung existiert unabhängig von dem hierarchischen Radiosity-System und kann somit auch für andere Beleuchtungssysteme als Basis dienen. Die Verwaltung hat während der Implementierungsphase ca. ein Viertel der Zeit beansprucht die nötig war um das gesamte System zu implementieren und wird daher ebenfalls näher beschrieben.
Die Verwaltung wurde so konzipiert, dass es später möglich sein könnte einen einfachen Szenen-Editor als Frontend für die Verwaltung zu programmieren. Sie bietet die Möglichkeit, Materialien, Lichtquellen und Flächen zu verwalten. Zu einer Szene können dabei beliebig viele Flächen, Lichtquellen und Materialen hinzugefügt werden. Leider ist die Erstellung einer Szene bis jetzt nur über die Programmierschnittstelle möglich, wenngleich auch eine Funktion existiert, über die eine Szene direkt von einer Datei geladen werden kann. Die Umsetzung dieser Funktion war aus Zeitgründen leider nicht mehr möglich.
Alternativ ist es immer möglich, der Verwaltung bereits fertige Objekte durch ein beliebiges System zu überreichen, welche dann als Kopie in der Verwaltung abgelegt werden.

9.2.1 Die Flächen

Die Flächen die in der Verwaltung genutzt werden heißen Surface und bilden die Verwaltung für eine Fläche in der Szene, wobei eine Fläche eine beliebige Anzahl von Eckpunkten besitzen kann. Die maximale Anzahl von Eckpunkten für eine Fläche wird in der Klasse Polygon festgelegt. Von einem Polygon wurde für dieses System die Klasse Quad abgeleitet, welche spezielle Funktionen für Quads bietet, wie z.B. eine Funktion zur Berechnung des Flächeninhalts. Von einem Quad werden dann alle verwendeten Flächen abgeleitet, wie die Klasse Surface und die Klasse Patch.

Ein Surface enthält einen Zeiger auf ein ihm zugewiesenes Material, sowie einen Quadtree, der für die spätere Unterteilung einer Fläche durch das hierarchische Radiosity benötigt wird. Ein Quadtree besteht selbst aus mehreren Nodes, welche jeweils die innneren Knoten und Blätter des Baumes bilden. Jede Node enthält Zeiger auf vier weitere Nodes und einen Patch, wobei anhand des Materials entschieden wird, ob es sich um einen TexturePatch oder einen SurfacePatch handelt: Besitzt das Material, das dem Surface und somit auch den Patches im Baum zugewiesen ist, eine Textur, wird ein TexturePatch erzeugt, der später für das texturbasierte Radiosity genutzt werden kann.
Am Anfang existiert für ein Surface nur eine Root-Node im Quadtree, wobei alle Informationen für die Erstellung dieser aus dem Surface genommen werden. Für alle weiteren Kinder der Root-Node werden die Informationen jeweils von der Vater-Node bezogen.

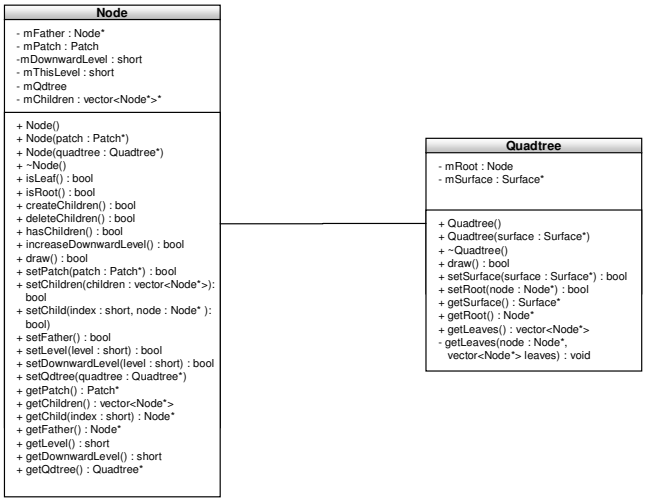


Soll ein Surface gezeichnet werden, wird ein Zeichnenaufruf an den Quadtree delegiert. Dieser teilt es seiner Root-Node mit, welche dann überprüft ob sie Kinder hat und, falls dies der Fall ist, es an diese

weiterleitet. Durch diesen rekursiven Aufruf wird garantiert, dass nur die Blätter des Baumes gezeichnet werden. Wurde ein Blatt erreicht, delegiert dieses den Zeichnenaufruf wiederum an seinen Patch. Im Normalfall zeichnet der Patch sich selbst mit seinen Eckpunkten und der empfangenen Radiosity. Das Zeichnen eines TexturePatches funktioniert etwas anders, wobei dies in Abschnitt 7.4 des texturbasierten Radiosity nachgelesen werden kann.

9.2.2 Der Baum

Die Klasse Quadtree ist die Kapselung des gleichnamigen Baumes. Sie enthält nur eine handvoll Funktionen, sowie einen Zeiger auf das Surface zu dem sie gehört und einen Zeiger auf ihre Root-Node. Eine Node bildet dann die Klasse aus der ein Quadtree besteht. Die Nodes können dabei sowohl innere Knoten, als auch Blätter des Baumes sein. Eine Aufteilung von Node in zwei Klassen, eine für innere Knoten und eine für die Blätter, wurde in Betracht gezogen, aber wieder verworfen, da bei einer solchen Struktur von vornherein klar sein muss, welche Nodes Blätter sind und welche nicht. Da dies aber bei einem adaptiven Mesh nicht möglich ist, existiert lediglich eine Klasse für alle Nodes.



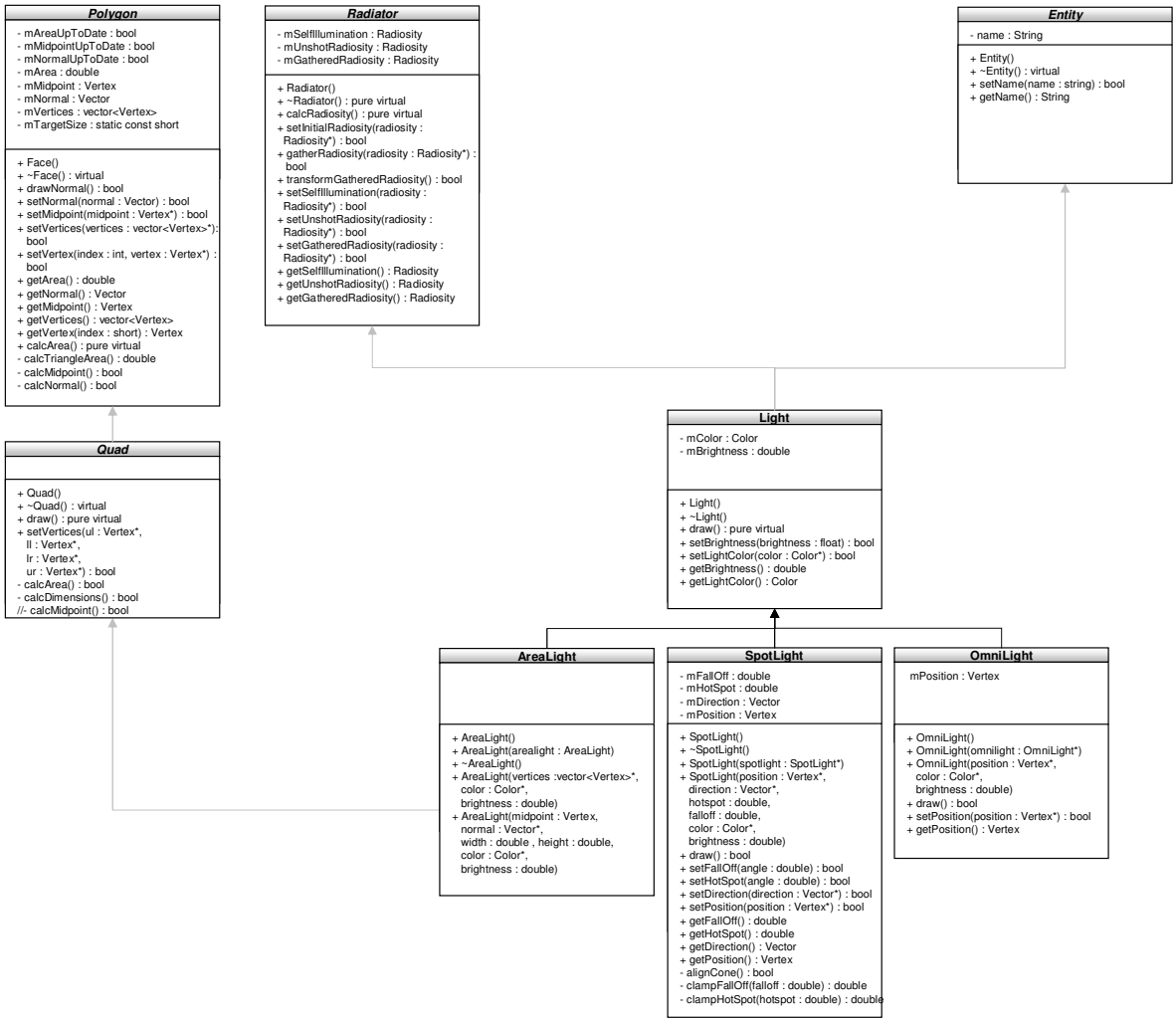
9.2.3 Dreiecke oder Vierecke

Auf Grund dieser Struktur ist es nicht möglich, innerhalb des Systems zwischen verschiedenartigen Flächen zu unterscheiden. So kann es passieren, wenn die Klasse Quad von Polygon abgeleitet ist, dass die Funktionen der Klasse Quad für vier Punkte funktionieren, jedoch nicht für drei. Alternativ könnten dann vier Eckpunkte für ein Dreieck definiert werden, wobei sich dann einer der Eckpunkte auf der Verbindungslinie zwischen zwei anderen befindet. So kann zwar ein Dreieck erstellt werden, dieses wäre allerdings intern nichts anderes als ein Quad.

Um dieses Problem zu umgehen hätte das System nicht auf Quads, sondern auf Dreiecken aufgestellt werden dürfen. Die Wahl von Dreiecken wurde auch in Betracht gezogen, jedoch aus einem entscheidenden Grund verworfen.
Für Dreiecke spricht, dass in der Computergraphik jedes Objekt, ausgenommen Splines, aus Dreiecken besteht. So wäre es möglich, eine Liste von Punkten für jedes Objekt der Szene einzulesen und über die Delauney-Triangulierung die benötigten Dreiecke zu erzeugen. Der Nachteil von Dreiecken in Bezug auf das texturbasierte Radiosity ist, dass es bei dem Mapping von Texturen auf ein solches auf Grund der verschiedenen Formen zwangsläufig zu Problemen kommt. Da die Texturen meistens vier Eckpunkte besitzen, müssten Texturen für ein Radiosity-System das auf Dreiecken basiert, auf mehrere Dreiecke verteilt werden. Da dies zu automatisieren sehr aufwendig wäre, ist die Wahl auf Quads gefallen. Bei einem rechteckigen Quad werden die Texturen immer korrekt dargestellt.

9.2.4 Die Lichtquellen

Die Verwaltung und das hierarchische Radiosity-System unterstützen neben Flächen auch Lichtquellen. Auch mit dem Wissen, dass jede Fläche in einem Radiosity-System eine Lichtquelle darstellt, wurden zusätzlich Lichtquellen aufgenommen. So wird zwischen einem AreaLight, einem SpotLight und einem OmniLight unterschieden. Zum Einen wurden die Lichtquellen aufgenommen, da es somit möglich ist auch noch andere Beleuchtungen zu simulieren, außer die von Flächenlichtquellen. Zum Anderen besteht so später die Möglichkeit, das System um ein Final-Gathering-System zu erweitern. Final-Gathering-Systeme sind eine Kombination aus einem Raytracing- und einem Radiosity-System, wobei das Radiosity-System auch ohne Photon-Mapping eine indirekte Beleuchtung ermöglicht. Die Szene wird dabei zu erst mit einem Radiosity-System beleuchtet. Im Anschluss wird das direkte Licht des Radiosity-Systems entfernt und dann die direkte Beleuchtung mit Schatten über den Raytracer abgewickelt.
Mit der Klasse Light ist es somit möglich, direkt die Lichtquellen auszumachen, da diese in unabhängigen Listen von den Surfaces „gelagert" werden, und diese direkt in dem Raytracer zu nutzen. Da ein Raytracer ebenfalls die Möglichkeiten bietet SpotLights und OmniLights zu simulieren, macht es Sinn, diese auch in einem Radiosity-System zu implementieren.

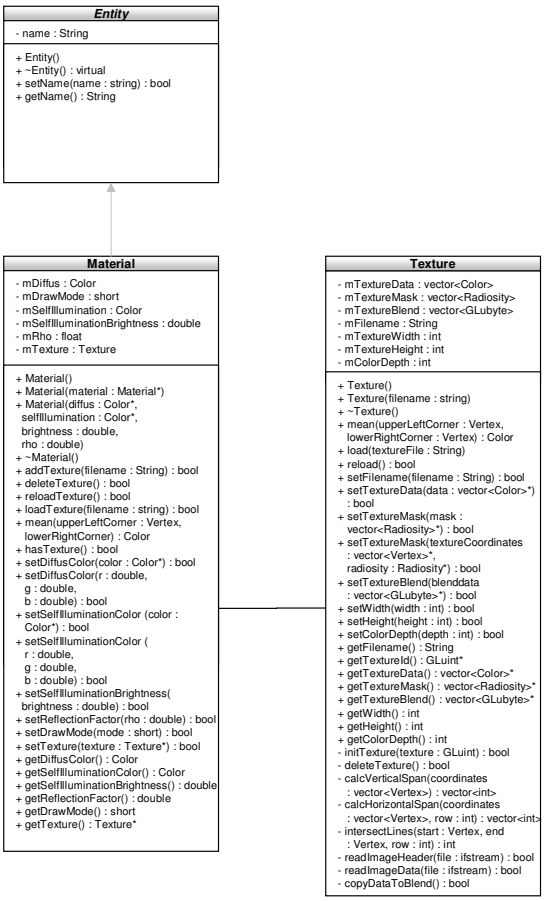


Diese Wahl stellte sich später als problematisch heraus. Denn dadurch, dass jede Fläche in einem Radiosity-System Radiosity versenden kann, stellt sich die Frage zu welcher Superklasse gehört ein AreaLight: Zu der Klasse Polygon, von der die Flächen abgeleitet sind, oder zu der Klasse Light?
Eine eindeutige Antwort auf diese Frage gibt es nicht, sondern nur verschiedene Sichten. Würde man die AreaLight als Fläche betrachten, wäre dies korrekt aber es wäre nicht mehr möglich eine konkrete

Flächenlichtquelle der Klasse Light zu erzeugen. Wäre AreaLight eine Lichtquelle, müsste diese Fläche eine gesonderte Behandlung erfahren, z.B. bei der Formfaktorberechnung, obwohl sie dieselben Eigenschaften wie eine Fläche hat. Ebenfalls hätte eine unschöne Code-Duplizierung erfolgen müssen, da die Attribute und Funktionen nicht mehr von Polygon hätten abgeleitet werden können.
In der vorliegenden Lösung existiert die Klasse AreaLight und wurde sowohl von Polygon als auch von Light abgeleitet. Um dem Problem der Diamond-Vererbung aus dem Weg zu gehen wurden verschiedene Maßnahmen getroffen, auf die hier nicht näher eingegangen wird. Details können im projekteigenen Wiki nachgelesen werden [Wiki].
Obwohl eine Implementation der Lichtquellen Klassen vorgesehen wurde kam es aus Zeitgründen nicht zu einer Umsetzung dieser Planung. Die Klassen existieren in der Verwaltung und sind auch vollständig implementiert, jedoch sind die Funktionen des Radiosity-Systems leer.

9.2.5 Materialien und Texturen

Die Eigenschaften einer Fläche und das damit verbundene Erscheinungsbild werden in dem vorliegenden System über Materialien festgelegt. Ein Material kann dabei wie in einem Modellierungs-Editor mehreren Flächen zugewiesen werden. Das Material legt die diffuse Farbe einer Fläche fest. Weiterhin wird die Eigenemission als Farbe festgelegt und kann dann über einen Faktor aktiviert werden. Da das System mit dem rgb-Farbraum arbeitet, werden alle Parameter als rgb-Wert angegeben. Letzlich werden noch der Reflektionsfaktor ρ und der aktuelle Zeichenmodus über das Material definiert. Bei den Zeichenmodi kann zwischen einem wireframe-und eine solid-Modus gewählt werden.

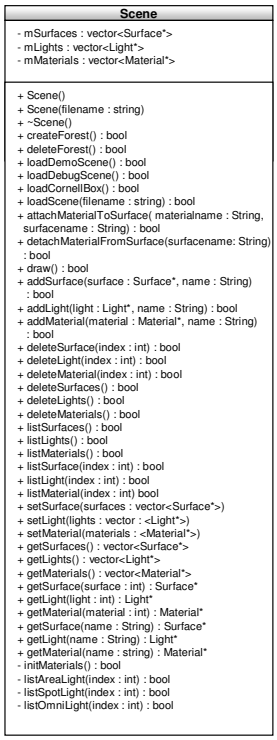


Für das normale hierarchische Radiosity reicht das Material aus, dieses kann aber für das texturbasierte Radiosity auch noch um eine Textur erweitert werden. Eine Textur kann dabei von einem beliebigen Ort geladen werden, wobei bis jetzt nur .pgm und .ppm Dateien geladen werden können.

Das Laden von Texturen wird hier noch über eine Funktion in der Klasse Texture realisiert sollte aber nach Möglichkeit in einen Loader ausgelagert werden, wobei es dann für jeden Datentyp einen eigenen Loader geben würde. Abhängig von der angegebenen Datei sollte dann der richtige Loader ausgewählt werden.

9.2.6 Die Klasse Scene

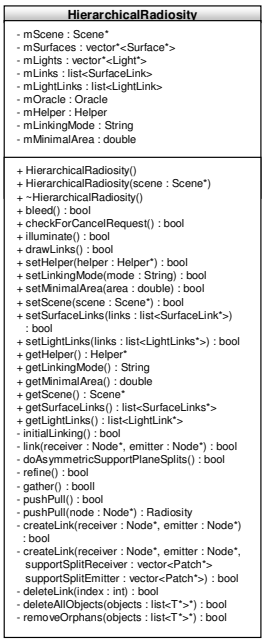
Die Klasse Scene stellt die eigentliche Verwaltung dar. Sie bietet mehrere Funktionen an, um Flächen, Lichtquellen und Materialien zu verwalten. Dazu zählt das Löschen sowie das Hinzufügen, Auflisten und Zugreifen auf alle Objekte die verwaltet werden.
Die Klasse Scene hat neben den Objekten der Szene auch eine Standard-Bibliothek, die verschiedene Materialien enthält. Da auch diese mit Namen verwaltet werden, ist eine einfache Zuweisung von Materialien auf Flächen möglich. Sollen neue Materialien hinzugefügt werden, und sind diese identisch zu den bereits vorhandenen, wird ein Material nicht in die Bibliothek aufgenommen, sondern ein Zeiger zu dem bereits vorhandenen Material gesetzt.
Damit Objekte identifiziert werden können, existiert die abstrakte Klasse Entity. Sie bietet die Möglichkeit, allen von ihr abgeleiteten Klassen einen Namen zuzuweisen, bzw. abzufragen. Ähnliches hätte auch über eine Map realisiert werden können, jedoch wäre der Name dann nur innerhalb der Verwaltung verfügbar gewesen. Der Vorteil an der Klasse Entity ist, dass auch innerhalb des Radiosity-Systems auf den Namen des Objektes zugegriffen werden kann. Diese Funktion war auch beim Debugging sehr hilfreich.



Momentan enthält die Klasse Scene noch mehrere Methoden, die jeweils eine unterschiedliche Szene laden. Besser wäre es, einen Loader zu schreiben, der die Szenendaten aus einer Datei lädt und dann ein entsprechendes Scene-Objekt erstellt. Dann könnten auch mehrere Szenen gleichzeitig existieren, die über den Loader verwaltet werden können, z.B. Übergabe der darzustellenden Szene an das Radiosity-System.

9.3 Das Radiosity-System im Detail

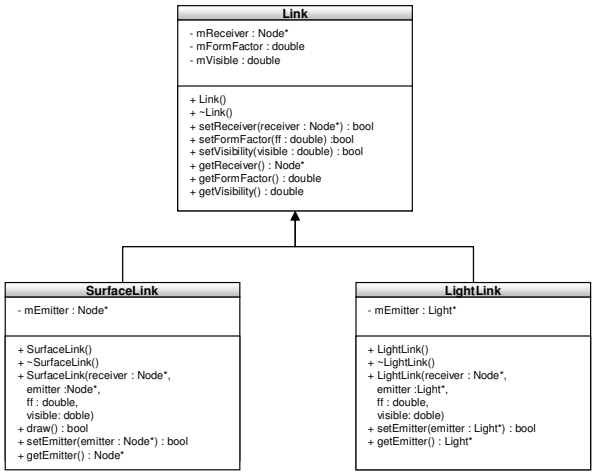
Das Radiosity-System besitzt, genau wie die Verwaltung nur eine Schnittstelle, über die Daten an das System übergeben werden können. Woher die Daten kommen spielt für das Radiosity-System somit keine Rolle. Die Klasse HierarchicalRadiosity realisiert alle Funktionen, die für ein hierarchisches Radiosity-System nötig sind: das Initial Linking, das Refining, das Gathering und das Push-Pull. Für Initial Linking stehen dabei zwei Modi zur Auswahl. Zum Einen der normale Modus, der auf Sichtbarkeit prüft und enstprechend zwei Flächen miteinander verlinkt oder nicht.
Der zweite Modus bedient sich des asymmetrischen Support-Plane-Splits. Bevor zwei Flächen miteinander verlinkt werden, wird geprüft ob eine Support-Plane-Split nötig ist. Ist das der Fall, wird dieser durchgeführt und eine Fläche real geteilt. Dadurch entstehen dann zwei oder drei neue Flächen, abhängig von der Teilung, wobei die ursprüngliche Fläche gelöscht wird. Damit dies nicht unkontrolliert geschieht, kann als Abbruchkriterium eine minimal zu teilende Flächengröße angegeben werden. Ist der Flächeninhalt einer Fläche kleiner als dieser Wert, findet keine Unterteilung statt. Dieser Modus war ursprünglich dazu gedacht, Aliasing-Effekte an Kanten zu verhindern. Durch die asymmetrischen Unterteilungen sieht das Gesamtergebnis allerdings schlechter aus als mit dem normalen Initial Linking, da u.a. im aktuellen Zustand alle Flächen unabhängig von der Entfernung geschnitten werden. Sinnvoll wäre es hier, nur die Flächen zu teilen, die auch wirklich aneinander grenzen.
Neben dem Modus für das Initial Linking und den Listen für die Links, hält das Radiosity-System auch noch ein Orakel- und ein Helper-Objekt.



9.3.1 Die Links

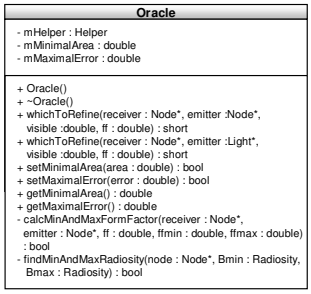
Für den Strahlungsaustausch zwischen Flächen arbeitet das hierarchische Radiosity-System mit Links. Es werden in der Klasse HierarchicalRadiosity zwei Listen für Links gehalten. Es wird unterschieden zwischen SurfaceLinks und LightLinks, wobei SurfaceLinks für den Strahlungsaustausch zwischen zwei Flächen, und LightLinks für den Austausch von Strahlung zwischen Lichtquelle und Fläche dienen. Jeder Link kann als gerichtete Kante betrachtet werden, denn er enthält zwei Pointer, einen zum Sender und einen zum Empfänger der Radiosity. Des Weiteren enthält ein Link den Formfaktor, d.h. eine Information darüber wieviel Strahlung beim Empfänger ankommt, und den Visibilitätsfaktor. Beide Informationen werden bei der Erstellung des Links einmal errechnet und gelten solange bis der Link zerstört wird.

Nach einiger Überlegung wurde den LightLinks eine Bedingung hinzugefügt: Bei einem Austausch zwischen einer Fläche und einer Lichtquelle kann die Lichtquelle nur als Sender dienen, es existiert also keine Möglichkeit, für eine Lichtquelle andere Strahlung zu empfangen.
Der Grund für diese Entscheidung ist der Realität abgeschaut: Betrachtet man alleine das Leuchtmittel einer Lichtquelle, ist der eigentliche Strahlungserzeuger ein Draht durch den Strom fließt oder ein elektrisch aufgeladenes Gas. Am einfachsten wird es jedoch an dem Beispiel einer Kerze klar. Eine Kerze strahlt Licht ab, diese kann aber, auch wenn sich eine noch so helle Lichtquelle neben ihr befindet, keine andere Farbe annehmen. Das Einzige was sich verändert, wenn sich neben der Kerze eine hellere Lichtquelle in demselben Raum befindet, ist, dass der Einfluss des von der Kerze ausgesandten Lichts schwindet.



9.3.2 Das Orakel

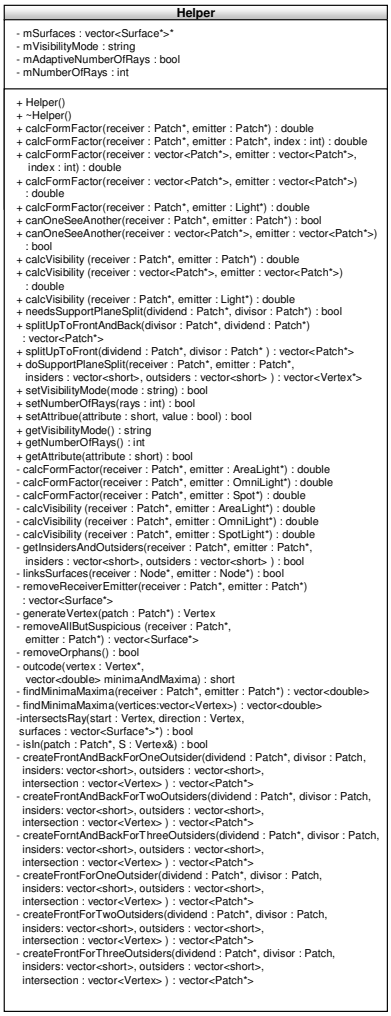
Das Orakel enthält in diesem System nur zwei Funktionen und eine Variable, die auf den Gehilfen des Radiosity-Systems zeigt. Sollte dem Radiosity-System einmal ein neuer Gehilfe zugewiesen werden, dann trägt dieses auch Sorge dafür, dass das Orakel denselben nutzt. Dies ist nötig um sicherzustellen, dass etwaige Einstellungen des Gehilfen auch von dem Orakel verwendet werden, z.B. Anzahl der Strahlen für den Visibilitätstest.
Die vorhandenen Funktionen des Orakels dienen dazu, für einen LightLink bzw. einen SurfaceLink zu entscheiden, ob er verfeinert werden soll oder nicht. Getestet wird dies mit Hilfe des Lischinski-Orakels.



9.3.3 Der Gehilfe

Der Gehilfe ist die Klasse Helper und trägt sehr elementare Funktionen für die Durchführung des Strahlungsaustauschs. Dies sind alle jene Funktionen, die nicht in die Klasse HierarchicalRadiosity

gehören und sowohl von diesem, als auch von dem Orakel genutzt werden. Dazu gehören u.a. der Visibilitätstest, der beschleunigte Visibilitätstest (Outcode), die Formfaktor-Berechnung, sowie die Berechnung des Support-Plane-Splits, der einen Großteil der Funktionen ausmacht. Da die Klasse sehr viele verschiedene Themen behandelt und mit Kommentaren über 2000 Zeilen Code fasst wäre es sinnvoller gewesen, diesen einen Gehilfen in mehrere aufzuteilen, die sich dann jeweils auf ein Thema beschränken. Dies wäre durchaus nützlich, wenn z.B. nur eine andere Berechnung für Formfaktoren eingefügt werden soll, da dann eine Klasse ausgetauscht werden könnte, ohne dass die anderen Funktionen verloren gehen würden.
Der Gehilfe enthält selbst keine anderen Objekte, sondern nur eine Reihe von Parametern, mit denen verschiedene Eigenschaften des Visibilitätstests festgelegt werden können. Es kann somit zwischen dem normalen und dem beschleunigten Test gewählt werden, oder aber auch die Anzahl der Strahlen, die für den Test verwendet werden sollen, festgelegt werden.



9.4 Weitere Bestandteile

Die weiteren Bestandteile sind Module, die das Gesamtsystem aufwerten, jedoch für den Betrieb nicht unbedingt nötig sind. Zu diesen Klassen zählen der Tonemapper und der Renderer.

9.4.1 Der Renderer

Zu Beginn der Planung war das System so konzipiert, dass der Tonemapper auch ohne einen Renderer benutzt werden konnte. Zu diesem Zeitpunkt wurde der *draw()*-Befehl an die Klasse Scene gegeben, welche dann in der OpenGL-Mainloop dafür sorgte, dass die Szene in regelmäßigen Zeitintervallen gezeichnet wurde.
Aber durch zusätzliche Funktionen wurde eine Möglichkeit verlangt, diese einfach und vernünftig hinzuzufügen ohne die ursprüngliche Funktionalität einzuschränken. Dies ließ sich nur über eine zusätzliche Klasse, den Renderer, realisieren, was sich später auch als klug herausstellte (Kapitel 10: Multi-Threading).
Der Renderer kann im Prinzip als Verwaltungsinstanz für alle Rendering-Aufgaben gesehen werden. Dieser bietet die Möglichkeit, Flächen, Lichtquellen und Links darzustellen. Über den Renderer kann so z.B. angegeben werden in welchem Modus die Szene gezeichnet werden soll, wobei zwischen wired, solid und materialabhängig unterschieden werden kann. Es ist auch schon die Möglichkeit eingeplant, dass interpoliert gezeichnet werden kann, so existieren bereits Methoden und eine Variable, über die dieser Modus aktiviert werden kann. Eine Umsetzung ist jedoch bis jetzt nicht erfolgt.
Eine weitere Funktion des Renderers ist, dass diese kontinuierlich die aktuelle Framerate berechnet und auf dem OpenGL Fenster ausgibt. Die konkrete Berechnung wird dabei von der Klasse FramesCounter übernommen. Die Darstellung der Frames per Second wird dann von der Klasse TextRenderer übernommen, die verschiedene Strings innerhalb eines OpenGL-Fensters über OpenGL Funktionen zeichnet.

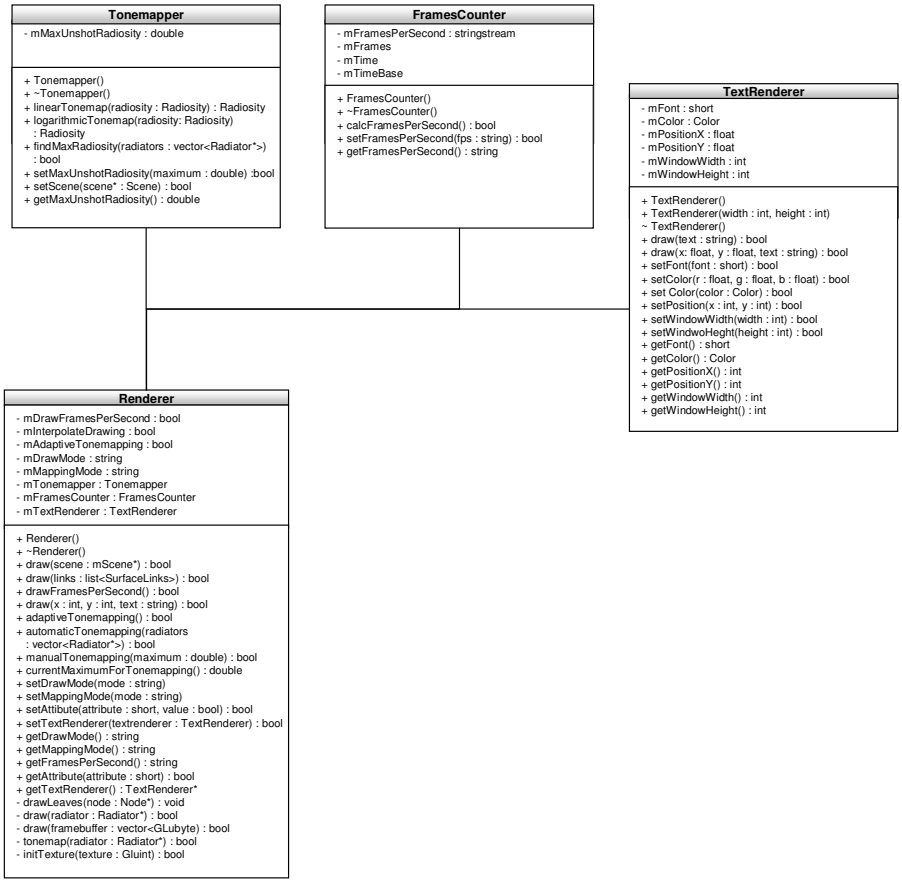
Neben der normalen Zeichenmethode, wie sie bereits in Kapitel 7.4 erläutert wurde, kann das Zeichnen weiter beschleunigt werden. Es ist möglich, sich von jeder zu zeichnenden Fläche eine Drawlist geben zu lassen. Diese Drawlist ist eine Liste, die alle darzustellenden Blätter eines Baumes enthält. Diese Liste kann dann in die Liste des Renderers eingefügt werden. Tut man dieses für alle Flächen, kann jeweils die Blättersuche in einem Baum gespart werden. Damit allerdings jeweils korrekt gezeichnet wird und die Drawlist nur einmal erstellt werden muss, empfiehlt es sich, diese Funktion erst nach dem Ende des Radiosity-Austauschs zu benutzen.

9.4.2 Das Tonemapping

Die letzte Aufgabe des Renderers ist das Tonemapping. Die Umsetzung des Tonemappings erfolgt dabei in der Klasse Tonemapper. Hier wird abhängig von dem aktuellen Modus das lineare oder das logarithmische Tonemapping ausgeführt. Da die Originalwerte der Objekte jedoch nicht verändert werden dürfen, da ansonsten die Simulation verfälscht wird, geschieht dies für jeden Zeichenschritt. Um den Tonemapper zu steuern kann über den Renderer automatisch der maximale Radiosity-Wert aller der Objekte ermittelt werden, die keine Eigenemission besitzen.[19] Dieser wird dann vom Tonemapper für die Berechnung verwendet. Für individuelle Änderungen kann dieser Wert dann manuell angepasst werden.
Zusätzlich zu dem normalen Tonemapper existiert noch ein adaptiver Tonemapper. Dieser rendert das aktuelle Bild in eine Textur und ermittelt dort das Maximum. In der aktuellen Version tut dieser dort nicht mehr, als den maximalen Wert zu ermitteln und alle anderen Werte durch diesen zu teilen. Leider reicht es aus, wenn nur ein Pixel den Wert „weiss“ hat, dass keine Änderungen an der Anzeige vorgenommen werden.
Im Idealfall würde es zusätzlich eine Gewichtungsfunktion geben, die die Pixel in der Mitte des Bildschirms höher gewichtet und so für eine Aufhellung sorgt, wenn dunkle Bereiche in die Mitte des Bildschirms kommen. Ein zusätzliches Problem des adaptiven Tonemappers ist, dass es bei einer Division durch einen kleineren Wert als „weiss“, also einer Aufhellung des Bildes, auf Grund des rgb-Farbsystems zu Farbverfälschungen kommt. Hier wäre ebenfalls eine Umstellung auf das xyz-Farbsystem sinnvoll.

[19] Würden die Objekte mit Eigenemission einbezogen, z.B. Lichtquellen, wäre die Szene permanent zu dunkel.

Ein weiteres Manko ist die Geschwindigkeit des adaptiven Tonemappers. Da die Farbwerte der Textur pixelweise abgefragt werden, ist der Vorgang sehr zeitintensiv, und somit sinkt die Framerate abhängig vom System stark ab. Eine Beschleunigung kann erzielt werden, wenn diese Berechnungen mittels Shader-Programmierung auf die Grafikkarte ausgelagert werden. Dadurch sollte auch der adaptive Tonemapper echtzeitfähig werden.



10 Multi-Threading

Die Berechnungen von Radiosity-Systemen können, abhängig von der Szene, sehr zeitintensiv sein. Damit der Benutzer nicht im Ungewissen über den Fortschritt der Simulation bleibt und die Veränderungen beobachten und weiterhin kontrollieren kann, wurde das System in mehrere Threads aufgeteilt. Die Implementierung erfolgte mit der Bibliothek von POSIX, mit der das Erstellen und Kontrollieren von Threads gut umgesetzt werden kann.
Das System besteht im aktuellen Entwicklungsstatus aus insgesamt drei Threads. Der erste Thread enthält die OpenGL-Mainloop. Somit ist es möglich, zu jedem Zeitpunkt „Interrupts“, die über die Tastatur eingehen, zu verarbeiten. Des Weiteren kann so garantiert werden, dass der Inhalt des OpenGL Fensters kontinuierlich aktualisiert wird.
Der zweite Thread wird gestartet, wenn das Radiosity-System aufgerufen wird um eine Szene zu beleuchten. Diese Funktion wurde integriert, um die teilweise sehr feine Unterteilung von Szenen durch ein falsch konfiguriertes Orakel zu unterbinden. Durch einen Tastendruck kann somit der Strahlungsaustausch gestoppt, und der aktuelle Status gesichert werden.
Ein grundsätzlicher Vorteil dieser Aufgabenverteilung ist, dass während des Strahlungsaustauschs eine kontinuierlicher Fortschritt beobachtet werden kann und nicht erst gewartet werden muss bis das System konvergent ist. Damit der Mainloop-Thread auf einem Single-Prozessor-System nicht zu viel Rechenleistung verbraucht, gibt es die Möglichkeit, das stete Aktualisieren des OpenGL-Fensters zu deaktivieren. Der Inhalt wird dann nur auf Tastendruck neugezeichnet.
Der dritte Thread ist das Main-Programm, welches aber nichts weiter tut als auf das Ende des Programms zu warten.

Da das Multi-Threading erst am Ende den Weg in das System gefunden hat, gibt es auch hier noch Verbesserungsmöglichkeiten. So ist der Observation-Mode, also das permanente Beobachten des Fortschritts, nicht möglich wenn das asynchrone Initial Linking durchgeführt wird. Hier kann es passieren, dass neue Surfaces erstellt werden und dann ein Lesezugriff auf noch nicht komplett initalisierte Teile des Objekts ausgeführt wird. Damit das System threadsicher wird, sollte die Klasse Surface eine Kontrollstruktur bekommen, die darüber informiert, ob sich in dem „kritischen Bereich“ aktuell ein schreibender Thread aufhält. Da dies aber sicherlich keine leicht Aufgabe darstellt musste von der Implementation zum aktuellen Zeitpunkt abgesehen werden.

11. Die Verbesserungen

Es gibt eine Reihe von Verbesserungsmöglichkeiten. Dazu gehören unter anderem die Beschleunigung des Systems, z.B. durch die Verwendung des zweiten Formfaktors, als auch die Implementation von weiteren Funktionen. Konkrete Optimierungsmöglichkeiten können an dem Ende der Teilabschnitte in der Systembeschreibung nachgelesen werden. Alternativ existiert eine Aufgabenliste zu jeder Klasse im projekteigenen Wiki [Wiki].

12. Dokumentation

Ein wichtiger Punkt, der neben der Verbesserung von Funktionalitäten erwähnt werden sollte, ist die Dokumentation. Ein Ziel der Studienarbeit war es, den gesamten Entwicklungsprozess von der Planung bis zur Ausarbeitung zu dokumentieren. Die Aufmerksamkeit galt hierbei nicht nur den doxygen-Kommentaren und der ausführlichen Dokumentation des Codes, sondern insbesondere den Me-

tainformationen. Diese wurden während der Entwicklung des Systems in einem Wiki-System fortlaufend gepflegt, damit zu einem späteren Zeitpunkt, nach der Implementierung, z.B. verschiedene Entscheidungen nachvollzogen werden können. Das Wiki enthält somit Themen wie die Aufgabenstellung, das aktuelle Klassendiagramm, ein Logbuch das alle Änderungen enthält, das Pflichtenheft, Optimierungsvorschläge und viele weitere Themen. Der Aufwand für die gesamte Dokumentation, d.h. Wiki plus Kommentare, beläuft sich auf ca. 50% der gesamten Zeit.
Da leider am Ende auch hier die Zeit fehlte, müsste das Wiki ebenfalls auf den neuesten Stand gebracht werden. Dies bezieht sich hauptsächlich auf die Dokumentation der Metainformationen. Die aktuelle Dokumentation kann unter https://www.sowenig.de/wiki gefunden werden. Die doxygen-Online-Dokumentation kann ebenfalls dort gefunden werden.

13. Das Programm

Da die Ergebnisse in den Kapiteln vorher bereits ausführlich behandelt wurden, werden in diesem Kapitel nur noch die Bedienungsanleitung, die Features und aktuelle Bilder des Systems präsentiert.

13.1 Bedienungsanleitung

Das Programm besitzt keine GUI und lässt sich somit auschließlich mit der Tastatur bedienen. Es ist zwar möglich sich in der Szene mit der Kamera zu bewegen, das Drehen der Kamera über die Pfeiltasten ist jedoch nicht implementiert. Zusätzlich kann die Kamera mit den `Bildab` und `Bildauf` Tasten hoch und runter bewegt werden. Der Grund dafür ist, dass die bis jetzt benutzte Kamera von OpenGL in eine eigene Klasse gekapselt werden sollte, aber auch dies konnte aus Zeitgründen nicht mehr umgesetzt werden.
Wird das Programm gestartet, öffnen sich zwei Fenster: die Konsole, sowie das OpenGL-Fenster. Das OpenGL-Fenster stellt das eigentliche Fenster in die virtuelle Welt dar, wobei hier am Anfang nur eine nicht beleuchtete Szene zu sehen ist. Die Konsole enthält hingegen größtenteils nur Logging-Informationen. Bevor die Simulation gestartet werden kann, können noch einige Einstellungen vorgenommen werden.

Zuerst sollte der Modus für den Visibilitätstest festgelegt werden. Es kann zwischen „normal“ und „fast“ umgeschaltet werden, wobei „fast“ die Variante mit Pre-Test ist. Dies ist die Standard-Einstellung. Mit den Tasten + und − kann somit die Anzahl der Strahlen festgelegt werden, mit denen der Sichtbarkeitstest vollführt werden soll. Hier bei gilt das Motto: Je mehr Strahlen, desto besser das Endergebnis, aber umso länger dauert die Simulation. Es empfiehlt sich dabei den Wert abhängig von der Komplexität der Szene zu wählen.
Können sehr viele Schatten auftreten, oder sind die Lichtquellen gut versteckt, dann empfiehlt es sich, mehrere Strahlen zu verwenden. Ein gutes Maß sind Werte zwischen 20 und 100 Strahlen. Falls man sich keine Gedanken darüber machen möchte, kann mit der Taste `N` die Wahl der Strahlenmenge dem System überlassen werden. Wie angemerkt in Kapitel 4.3.3, kann diese noch nicht ausgereifte Funktion den Strahlungsaustausch erheblich verlangsamen.

Ebenso muss vor dem Starten der Simulation festgelegt werden, in welchem Modus das Initial Linking vollführt werden soll. Es sind auch hier zwei Modi verfügbar: das normale Linking und das asymmetrische Linking. Der Modus kann mit der Taste `L` festgelegt werden. Sollte die Wahl auf das asymmetrische Linking fallen, kann über die Tasten `Shift+A` und `A` die Grenze verändert werden, wann eine Fläche durch den Support-Plane-Split unterteilt wird. Mit `Shift+A` wird die Grenze nach oben, und mit `A` nach unten gesetzt. Die Grenze ist dabei der minimale Flächeninhalt, der für eine Unterteilung benötigt wird. Die Standard-Einstellung ist das normale Linking.[20]
Eine weitere Einstellungsmöglichkeit vor dem Starten ist die Genauigkeit des Orakels. Da in diesem Programm das Lischinski-Orakel benutzt wird, kann eine maximale Abweichung von der Radiosity-Funktion festgelegt werden. Dies ist mit den Tasten `Shift+E` und `E` möglich. Die Tastenkombination `Shift+E` sorgt dabei für ein ungenaueres Ergebnis, wohingegen `E` für eine genauere Unterteilung sorgt. Die Standard-Einstellung braucht nicht verändert werden, da sie sich als gutes Maß herausgestellt hat.
Es gibt noch eine Möglichkeit die Ausführung der Simulation zu beschleunigen. Startet das Programm, befindet es sich im „Observer“-Modus. Dies bedeutet, dass das OpenGL-Fenster stetig aktualisiert wird. Dies ist auch gut, wenn man beständig über den Zustand der Simulation informiert sein möchte, allerdings kostet diese Funktion auf Single-Prozessor-Systemen ca. 50% der CPU-Leistung. Es

[20] Sollte das asymmetrische Linking ausgewählt worden sein, empfiehlt es sich, den „Observer“-Modus abzuschalten, da es ansonsten zu einem Absturz kommen kann, weil diese Funktion nicht threadsicher ist.

empfiehlt sich daher, diesen Modus während der Simulation mit der Taste `O` abzuschalten. Eine Aktualisierung ist dann immer noch auf Befehl möglich, wenn z.B. die Pfeiltasten gedrückt werden.

Sind die Vorbereitungen getroffen worden, kann mit der Taste `B` für „bleed“, abgeleitet von Color-Bleeding, die Simulation gestartet werden. In der Konsole werden währenddessen Informationen zu dem aktuellen Schritt des hierarchischen Radiosity-Systems ausgegeben. Interessant sind in diesem Zusammenhang die Ausgaben über die Anzahl der Links. Die Funktion für die Erstellung der Links verhält sich zu Beginn wachsend, bis zu einem Maximum nach dem die Anzahl der neu erstellten Links fällt. Hat diese null erreicht, ist die Simulation beendet. Ein Abbruch der aktuellen Simulation kann zu jedem Zeitpunkt über die Taste `B` erfolgen.

Ist die Simulation beendet, kann der Tonemapper eingeschaltet werden.[21] Dies geschieht mit dem Tastendruck auf die Taste `T`. Erstmaliges Drücken aktiviert den linearen Tonemapper, der zweite Tastendruck den logarithmischen und ein weiterer Tastendruck deaktiviert den Tonemapper wieder. Damit das Bild mit dem Tonemapper möglichst korrekt dargestellt wird, können die Tasten `Shift+T` gedrückt werden. Mit dieser Kombination wird automatisch das Radiosity-Maximum aller der Objekte gesucht, die keine Eigenemission besitzen. Dies ist die optimale Darstellung der Szene, da nur so gewährleistet ist, dass die Werte des Monitors nicht überschritten werden. Für manuelle Änderungen kann mit `Shift+R` bzw. `R` eine Korrektur an dem Tonemapper vorgenommen werden. Das Bild wird dadurch heller oder dunkler. Letztlich gibt es dann die Möglichkeit den adaptiven Tonemapper über die Taste `Z` einzuschalten. Da dieser aber noch nicht ausgereift ist, kann es zu Fehldarstellungen kommen. Dieser lohnt sich natürlich nur, wenn der normale Tonemapper abgeschaltet ist.

Zu guter Letzt kann noch der Zeichenmodus festgelegt werden. Hier gibt es die Möglichkeit die Szene im wired- oder solid-Modus darzustellen. Der wired-Modus kann mit der Taste `W`, der solid-Modus mit der Taste `S`, eingeschaltet werden. Die default-Einstellung ist der solid-Modus.
Ist die Simulation abgeschlossen können mit `Shift+L` noch die Links eingeblendet werden, die für den Strahlungsaustausch verwendet wurden. Da es aber noch nicht möglich ist diese levelweise anzuzeigen, können keine Details erkannt werden.

Das Programm kann mit der Taste `Esc` zu jedem Zeitpunkt beendet werden.

13.2 Features

Die Features können der Anforderungsliste im übernächsten Abschnitt entnommen werden. Diese enthält alle Features des Systems, ausgenommen das Multi-Threading, welches aber bereits näher in Kapitel 10 beschrieben wurde.

13.3 Bilder

In diesem Abschnitt können eine Reihe von Bildern betrachtet werden, die mit dem in dieser Studienarbeit entwickelten Programm entstanden sind. Alle Bilder wurden mit der Version erstellt, die der Studienarbeit beiliegt.

[21] Der Tonemapper kann schon während der Simulation aktiviert werden.

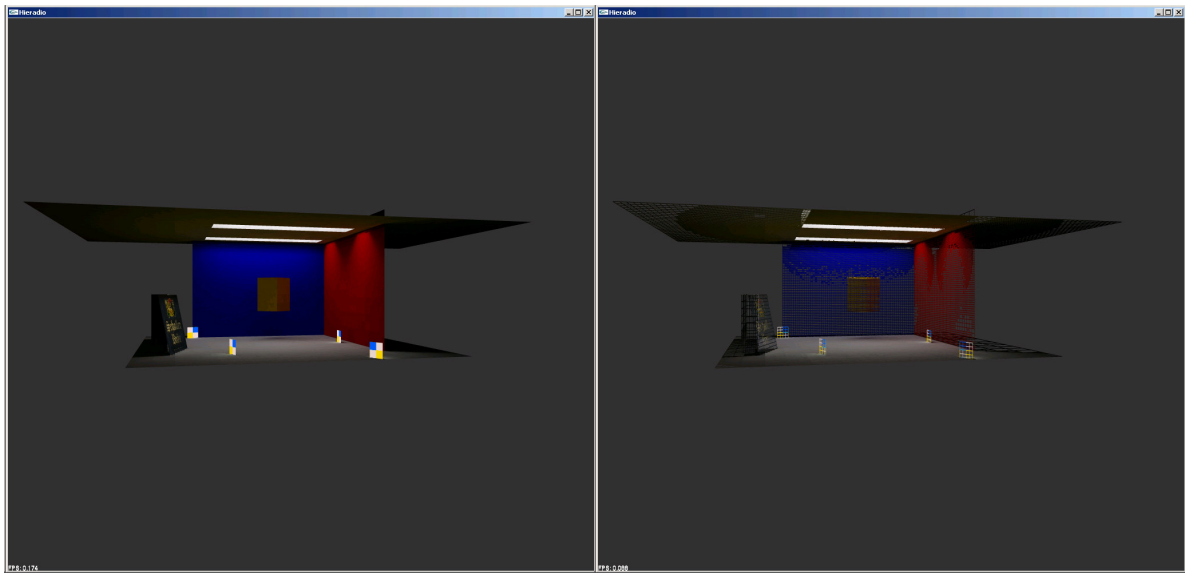


Abbildung 34: In dieser Szene kann der Strahlungsaustausch zwischen Texturen und Flächen beobachtet werden. Die kleinen verteilten Quadrate sind Texturen, die ihre Strahlung in die Welt senden.

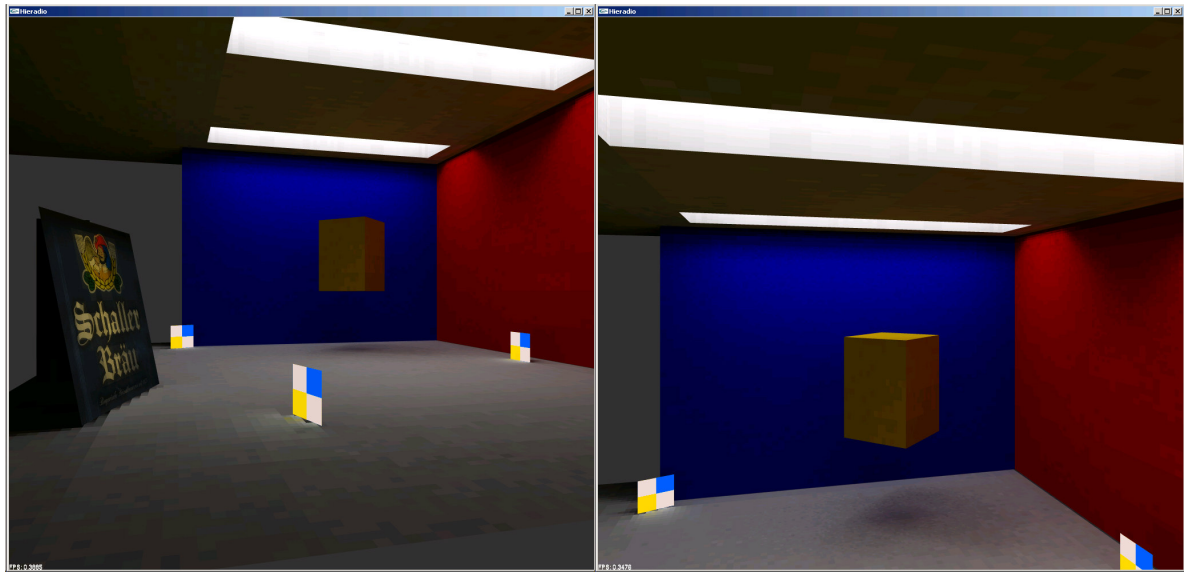


Abbildung 35: Eine Nahaufnahme der obigen Szene. Hier können der Schattenwurf im linken Bild sowie die Beleuchtung der Szene durch die kleinen Quadrate betrachtet werden.

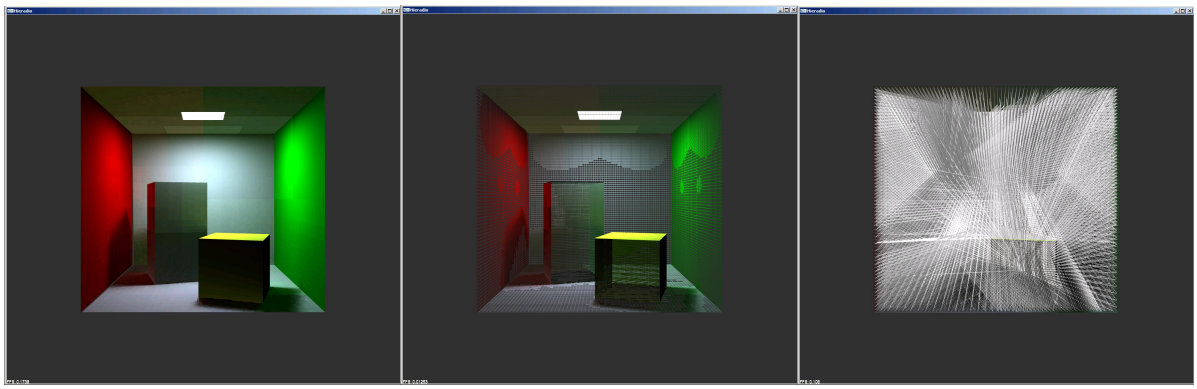


Abbildung 36: Diese Szene basiert auf den Originaldaten der Cornell Box [Corn98], wobei links die Szene im solid-Modus und in der Mitte im wired-Modus gezeichnet wurde. Das rechte Bild zeigt alle für den Strahlungsaustausch verwendeten Links.

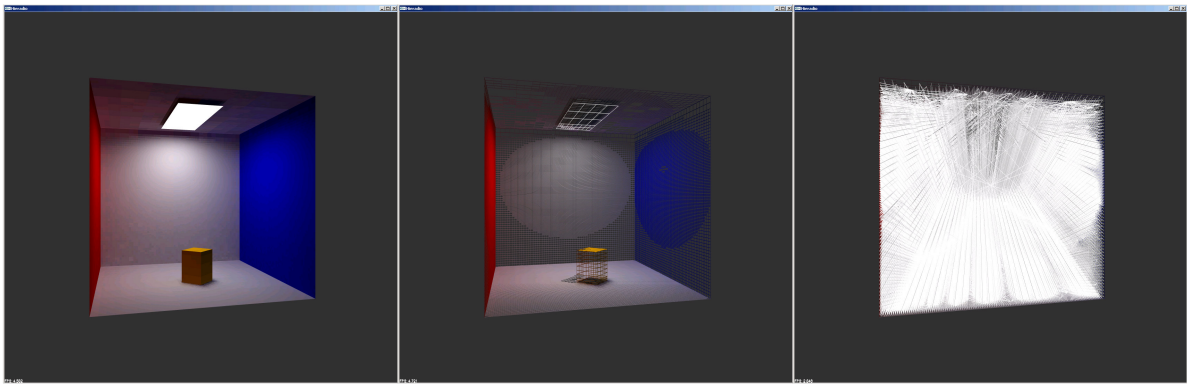


Abbildung 37: Eine eigens für dieses Radiosity-System erstellte Cornell Box. Auch hier sind beide Zeichen-Modi, sowie die für den Strahlungsaustausch verwendeten Links zu sehen.

13.4 Anforderungsliste

Diese Liste enthält alle offiziellen Anforderungen die an das System gestellt wurden. Nicht alle Anforderungen konnten umgesetzt werden, was aber auch nicht beabsichtigt war, da diese Liste auch als Ideensammlung für künftige Erweiterungen dienen soll.

Lfd.Nr.	Priorität	Status	Anforderung
Ladevorgang			
1.1	1	OK	Es muss mind. eine Demo-Szene in das Programm integriert sein, die geladen werden kann.
1.2	1	OK	Es muss eine Schnittstelle existieren, um eine Szene an die Objekt-Verwaltung zu übergeben.
1.3	2	OK	Eine Szene kann extern geladen und an das Programm übergeben werden.
1.4	3	-	Mehrere Polygone die eine Fläche bilden, sollen beim Ladevorgang zu einem Surface-Objekt zusammengefasst werden.
1.5	3	-	Das Laden einer Szene von einer Datei darf möglich sein.
1.6	3	-	Ein Surface-Object soll beim Ladevorgang für jedes Polygon erstellt werden.
1.7	3	-	Ein Licht-Object soll beim Ladevorgang für jedes Polygon erstellt werden.
1.8	3	-	Ein Material-Object soll beim Ladevorgang für jedes Polygon erstellt werden.
1.9	3	-	Das Laden kann durch eine zusätzliche Oberklasse Reader getätigt werden.
Materialien			
2.0	1	OK	Es müssen Materialien erstellt werden können.
2.1	1	OK	Ein Material muss zwei Farbkanäle und einen Texturslot besitzen.
2.2	1	OK	Es muss möglich sein Texturen zu laden. [.pgm]
2.3	3	-	Es können weitere Texturen geladen werden. [.gif,jpg,...]
2.4	1	OK	Texturen müssen einem Material zugewiesen werden können.
Editor			
3.0	3	-	Über einen grafischen Editor können Flächen erstellt werden.
3.1	3	-	In dem Editor können Flächen erstellt werden.
3.2	3	-	In dem Editor können Flächen verschoben werden.
3.3	3	-	Mit dem Editor können Lichtquellen erstellt werden.
3.4	3	-	Mit dem Editor können Lichtquellen verschoben werden.
3.5	3	-	In dem Editor können Materialien erstellt werden.
3.6	3	-	In dem Editor können Materialien einzelnen Flächen zugewiesen werden.
3.7	3	-	Über den Editor können Texturen geladen werden.
3.8	3	-	Über den Editor können Texturen mit Materialien verbunden werden.

Radiosity			
4.0	1	OK	Es muss ein Strahlungsaustausch zwischen Flächen durchgeführt werden können.
4.1	1	OK	Es muss ein Strahlungsaustausch zwischen Flächen und Texturen durchführbar sein.
4.2	1	OK	Es muss ein Strahlungsaustausch zwischen Flächen und Texturen durchführbar sein.
4.3	1	OK	Es muss verschiedene Lichtquellen für den Strahlungsaustausch geben.
4.4	1	o	Der Strahlungsaustausch zwischen Flächen und Lichtquellen muss funktionieren.
4.5	1	OK	Es muss eine Beschleunigungsstruktur für den Strahlungsaustausch bestehen. [Quadtree]
4.6	1	OK	Die Flächen müssen durch die Beschleunigungsstruktur adaptiv unterteilt werden.
4.7	3	-	Objekte die nahe beieinander liegen, sollen durch einen Szenengraph zusammengefasst werden.
4.8	3	OK	Der Support-Plane-Split soll Flächen asymmetrisch teilen.

Final Gathering			
5.0	3	-	Das System kann um Final Gathering erweitert werden, um ein verbessertes Ergebnis zu erzielen.
5.1	2	-	Es soll möglich sein negative Strahlung zu verschießen.
5.2	3	-	Es soll möglich sein Objekte interaktiv zu bewegen, so dass eine Änderung der direkten Beleuchtung stattfindet.
5.3	3	OK	Nach der Erweiterung soll das System interaktiv sein. [10fps]

Technisches			
6.0	1	OK	Der Formfaktor muss mit der Prisma-Methode bestimmt werden.
6.1	2	-	Die Formfaktorberechnung kann durch ein beliebiges Verfahren erweitert werden.
6.2	1	OK	Ein Tonemapper soll die Radiositywerte auf darstellbare Monitorwerte umrechnen.
6.3	2	OK	Es soll verschiedene Tonemapper geben.
6.4	1	-	Zwischen mehreren Patches muss eine Interpolation der Farbwerte stattfinden.
6.5	2	-	Die Patches unter dem Surface sollen für eine permanente Aktualisierung mit Pointern arbeiten.
6.6	3	-	Die draw()-Methode sollte abhängig von der Normalen einer Fläche entscheiden wie gezeichnet wird.
6.7	3	OK	Es soll ein Performance-Messung durchführbar sein.
6.8	2	OK	Es soll eine Schnittstelle existieren, um das Radiosity-System leicht austauschen zu können .
6.9	1	OK	Es muss ein einfacher Visibilitätstest für den Strahlungsaustausch verwendet werden. [Schattenwurf]
6.10	3	OK	Es darf ein Visibilitätstest mit hierarchischer Beschleunigung benutzt werden. [Octree, Szenengraph,...]
6.11	1	OK	Die Entwicklung muss unter Windows mit Visual Studio geschehen.
6.12	1	OK	Das System muss unter Windows lauffähig sein.
6.13	1	OK	Es muss ein Versionskontrollsystem für die Entwicklung genutzt werden. [SVN]
6.14	1	OK	Es soll möglich sein symbolische Lichtquellen einzufügen.
6.15	2	-	~~Das SpotLight soll zum Zeichnen über Axis & Angle realisiert werden.~~
6.16	2	OK	Der Visibilitätstest muss durch das Magische Quadrat erfolgen.
6.17	2	-	Das Magische Quadrat sollte durch ein Rastersystem ersetzt werden.
6.18	2	OK	Es soll einen Tonemapper geben, der mit einem Parameter verschiedene Belichtungsbereiche anzeigt.
6.19	3	-	Es soll das Prinzip der Photometrischen Konsistenz gewahrt werden.
6.20	3	-	Idealerweise soll für ein TexturePatch nur der Root-Knoten gezeichnet werden, so dass nicht alle Kinder des Baumes gezeichnet werden müssen.

Dokumentation			
7.0	1	o	Der Programmcode muss leicht wartbar sein.[22]
7.0	1	o	Der Programmcode muss leicht verständlich sein. [22]
7.1	1	OK	Es müssen doxygen-Kommentare benutzt werden.
7.2	1	OK	Einmal täglich muss ein doxygen-Dokumentation erstellt werden.
7.3	1	OK	Die Arbeitsgänge und das Vorgehen sollen dokumentiert werden.
7.4	1	OK	Es muss ein Wiki-System zur Dokumentation benutzt werden.

[22] Die Bearbeitung dieser Anforderungen ist erst abgeschlossen, wenn das Projekt abgeschlossen ist.

Verwaltung			
8.0	2	OK	Die Objekt-Verwaltung soll beliebige konvexe/planare Flächen verwalten können.
8.1	1	OK	Die Objekt-Verwaltung soll beliebige gleich-eckige Flächen verwalten.
8.1	2	-	Die Objekt-Verwaltung soll beliebige n-eckige Flächen gleichzeitig verwalten.
8.1	1	OK	Es muss eine automatische Flächenberechnung existieren.
8.2	1	OK	Es muss eine automatische Normalenbestimmung existieren.
8.3	1	OK	Es muss eine automatische Mittelpunktbestimmung existieren.
8.4	1	OK	Die Flächenberechnung muss für Dreiecke funktionieren.
8.5	1	OK	Die Normalenberechnug muss für Dreiecke funktionieren.
8.6	1	OK	Die Mittelpunktberechnung muss für beliebige konvexe und planare Flächen funktionieren.
8.7	2	-	Es soll nur ein Texturausschnitt auf eine Fläche gelegt werden können.
Legende			[nicht bearbeitet: -] [in Bearbeitung: o] [endlich fertig: OK]
			[unbedingt muss: 1] [kann/soll/darf: 2] [nice-to-have: 3]

14. Referenzen

[Bles06] Gabi Bleser: „Visualisierung und Virtuelle Realität“. Vorlesung. Universität Darmstadt, 2006

[Cohe85] Michael F. Cohen and Donald P. Greenberg: The hemi-cube: a radiosity solution for complex environments. In: Proceedings of the 12th annual conference on Computer graphics and interactive techniques 1985, ACM Press, S. 31-40

[Cohe88] Michael F. Cohen and Shenchang Eric Chen and John R. Wallace and Donald P. Greenberg: A progressive refinement approach to fast radiosity image generation. In: Proceedings of the 15th annual conference on Computer graphics and interactive techniques 1988, ACM Press, S. 75-84

[Cook81] Robert L. Cook and Kenneth E. Torrance: A reflectance model for computer graphics. In: SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques 1981, ACM Press, S. 307-316

[Corn98] Cornell University Program of Computer Graphics: „The Cornell Box“. URL: http://www.graphics.cornell.edu/online/box/ [Stand: Dezember 2006]

[Hanr91] Pat Hanrahan and David Salzman and Larry Aupperle: A rapid hierarchical radiosity algorithm. In: Proceedings of the 18th annual conference on Computer graphics and interactive techniques 1991, ACM Press, S. 197-206

[Kaji86] James T. Kajiya: The rendering equation. In: Proceedings of the 13th annual conference on Computer graphics and interactive techniques 1986, ACM Press, S. 143-150

[Kres97] Wolfram Kresse: "Effizientes und anwendbares Hierarchisches Radiosity unter besonderer Berücksichtigung der Visibilitätsbetrachtung". Diplomarbeit. Universität Darmstadt, 1997

[Lisc93] Dani Lischinski and Filippo Tampieri and Donald P. Greenberg: Combining hierarchical radiosity and discontinuity meshing. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques 1993, ACM Press, S. 199-208

[Lisc94] Dani Lischinski and Brian Smits and Donald P. Greenberg: Bounds and error estimates for radiosity. In: Proceedings of the 21st annual conference on Computer graphics and interactive techniques 1994, ACM Press, S. 64-74

[Müll05] S. Müller: „Photorealistische Computergrafik“. Vorlesung. Universität Koblenz, 2005

[Nuss28] W.Nusselt: Graphische Bestimmung des Winkelverhaltnisses bei der Warmestrahlung. VDIZ, 72, 673, 1928

[Phon73] Bui Tuong Phong: “Illumination of Computer-Generated Images”. Department of Computer Science. University of Utah, UTEC-CSs-73-129, July 1973.

[Schr93] P. Schröder and P. Hanrahan: On the Form Factor Between Two Polygons. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques 1993, ACM Press, S. 163-164

[Wiki] Sören Kewenig: „Hierarchisches Radiosity unter Berücksichtigung von Texturen“. URL: https://www.sowenig.de/wiki [Stand: Dezember 2006]