Bidirektionales Path Tracing

Studienarbeit

vorgelegt von
Christian Weizel

Institut für Computervisualistik
Arbeitsgruppe Computergrafik

Betreuer: Markus Geimer, Dipl.-Inf.
Prüfer: Prof. Dr. Stefan Müller

Motivation

In der Computergraphik gibt es Beleuchtungsalgorithmen, die aus einer mathematischen Szenenbeschreibung eine möglichst realistische Darstellung der beschriebenen Szene berechnen. Diese graphische Darstellung wird anschließend in Form eines Bildes im Graphikformat oder direkt am Bildschirm ausgegeben. Es wurden zur Beleuchtungs-Berechnung einer Szene verschiedene globale Beleuchtungsmodelle eingeführt. Dabei wird die Szene unter Berücksichtigung jeder Beleuchtungsart, die direkt oder indirekt auf einen Punkt in der Szene einwirkt, gerendert.
In dieser Studienarbeit wird einer dieser Algorithmen zur globalen Beleuchtungssimulation näher untersucht: bidirektionales Path Tracing. Beim bidirektionalen Path Tracing werden von einem festgelegten Betrachterpunkt aus Strahlen ausgesendet, die den Blick eines Auges oder die Aufnahme einer Kamera simulieren. Diese Sehstrahlen treffen auf Objekte der Szene und werden je nach Oberflächenbeschaffenheit dieser Objekte unterschiedlich reflektiert oder gebrochen. Zusätzlich werden Lichtstrahlen von den Lichtquellen der Szene ausgesendet. Ähnlich wie bei den Sehstrahlen wird die Interaktion der Lichtstrahlen mit den Objekten der Szene berechnet. Anschließend werden Beleuchtungsbeiträge des Lichtstrahls und des Sehstrahls, die an den Schnittpunkten der Strahlen mit den Objekten aufgegriffen werden, verrechnet und in einen Farbwert konvertiert, der im Ergebnisbild gesetzt wird.

Im Wintersemester 2002/2003 wurde an der Universität Koblenz-Landau im Rahmen eines Projektpraktikums für den Studiengang Computervisualistik ein Ray Tracer in der Sprache C++ entwickelt. Nachdem er die gewünschten Eigenschaften aufwies, wurde er erweitert und konnte danach u.a. Szenen mit Hilfe des Path-Tracing-Algorithmus rendern. Meine Motivation ist es jetzt, diesen Ray Tracer um weitere Klassen und Algorithmen zu ergänzen, damit Szenen mit bidirektionalem Path Tracing gerendert werden können. Mit dem Path Tracer war es möglich, Color Bleeding in den Ergebnisbildern zu simulieren. Mit bidirektionalem Path Tracing soll es dann zusätzlich möglich sein, Kaustiken darzustellen.
Das ist das Ziel, das ich mir gesetzt habe: eine Kaustik rendern.

Ich habe dieses Thema gewählt, weil mich interessiert, wie sich mathematisch und mit Hilfe der Computergraphik Szenen beliebigen Aufbaus so rendern lassen, dass der Betrachter den Eindruck hat, es handele sich um eine photorealistische Szene. Inwieweit das Ergebnis der Realität nahe kommt, muss jeder Betrachter des Ergebnisbildes für sich selbst entscheiden.

Einleitung

Aufbau der Studienarbeit

Im ersten Kapitel wird erklärt, wie ein bidirektionaler Path Tracer im Vergleich zu Ray Tracing und Path Tracing funktioniert. Anschließend wird im ersten Teil des zweiten Kapitels die Mathematik, die die Grundlage für das bidirektionale Path Tracing ist, erklärt. Im zweiten Teil des zweiten Kapitels werden zunächst die Klassen kurz beschrieben, die der zu erweiternde Ray Tracer bereits enthält und die in Bezug auf eine Erweiterung für bidirektionales Path Tracing wichtig sind. Anschließend erläutere ich meine Ansätze zur Realisierung der Algorithmen wie zum Beispiel Lichtstrahlen-Generierung und Beleuchtungsberechnung, die wichtige Bausteine für das bidirektionale Path Tracing sind. Hier werde ich auch Auszüge aus dem C++ - Quellcode einfügen, die zeigen, wie meine Ansätze realisiert wurden.
Im dritten Kapitel werden die Ergebnisse gezeigt. Dazu zählen die Bilder, die mit dem bidirektionalen Path Tracer gerendert wurden. Im letzten Kapitel wird der Inhalt dieser Studienarbeit noch mal resümiert, ob das Ziel erreicht wurde und welche Erweiterungen dieser Arbeit noch möglich sind.

Kapitel 1
Bidirektionales Path Tracing

In diesem Kapitel werden die Grundzüge des bidirektionalen Path Tracings erklärt, wobei keine mathematischen Details erläutert werden. Ich möchte zunächst ausgehend vom konventionellen Ray Tracing zu Path Tracing übergehen und von dort zu bidirektionalem Path Tracing überleiten. Jeder dieser Algorithmen hat seine Vor- und Nachteile.

1.1 Ray Tracing

Ray Tracing ist ein Beleuchtungsalgorithmus, der sich auf die spekulare Interaktion von Licht beschränkt. Im Vordergrund steht hier nicht die möglichst einfache Realisierung der Beleuchtung wie bei lokalen Beleuchtungsmodellen, sondern die bestmögliche, realistische Wiedergabe einer Szene. Ray Tracing simuliert den Prozess der Lichtausbreitung und arbeitet dabei nach den Gesetzen für ideale Spiegelung und Brechung. Daher eignet sich Ray Tracing gut für Szenen mit hohem spiegelndem Anteil. Die Grundidee ist, Lichtstrahlen auf ihrem Weg von der Quelle bis zum Auge zu verfolgen. Beim konventionellen Ray Tracing werden nur ideal reflektierte und ideal gebrochene Strahlen weiterverfolgt. Wenn die Lichtstrahlen bei der Lichtquelle starten, treffen in der Regel nur wenige Strahlen das Auge. Daher wird das Verfahren umgekehrt (Backward Ray Tracing) und die Strahlen starten beim Auge. Durch ein Abtastgitter, dessen Position in Bezug auf den Betrachterpunkt festgelegt ist, wird durch jedes Pixel des Gitters ein Strahl in die Szene geschickt und auf seinem Weg durch die Szene nach einer Reflexion oder Refraktion weiterverfolgt.
Trifft dieser Sehstrahl (Augenstrahl) auf ein Objekt, so wird das lokale Beleuchtungsmodell am Schnittpunkt berechnet. Zu den lokalen Beleuchtungsmodellen zählen zum Beispiel das Phong-Modell und die Modelle von Schlick und Blinn. Anschließend werden zwei neue Strahlen erzeugt, die beide am aktuellen Schnittpunkt starten: der reflektierte und der gebrochene (transmittierte) Sehstrahl. Beide Strahlen werden an die Trace-Methode übergeben und durch Rekursion ebenfalls auf ihrem Weg durch die Szene verfolgt. Dann wird der Leuchtdichtebeitrag dieser beiden Strahlen berechnet und zum Beleuchtungswert für den Augenstrahl hinzugerechnet. Dieser Beleuchtungswert wird schließlich in einen Farbwert umgerechnet und im Ergebnisbild gesetzt. Der Algorithmus terminiert, wenn eine Lichtquelle getroffen wird, die auf dem Strahl transportierte Energie zu gering wird oder wenn der Sehstrahl die Szene verlässt. In der Praxis wird meist eine maximale Rekursionstiefe durch eine Zähl-Variable als Abbruchkriterium festgelegt.

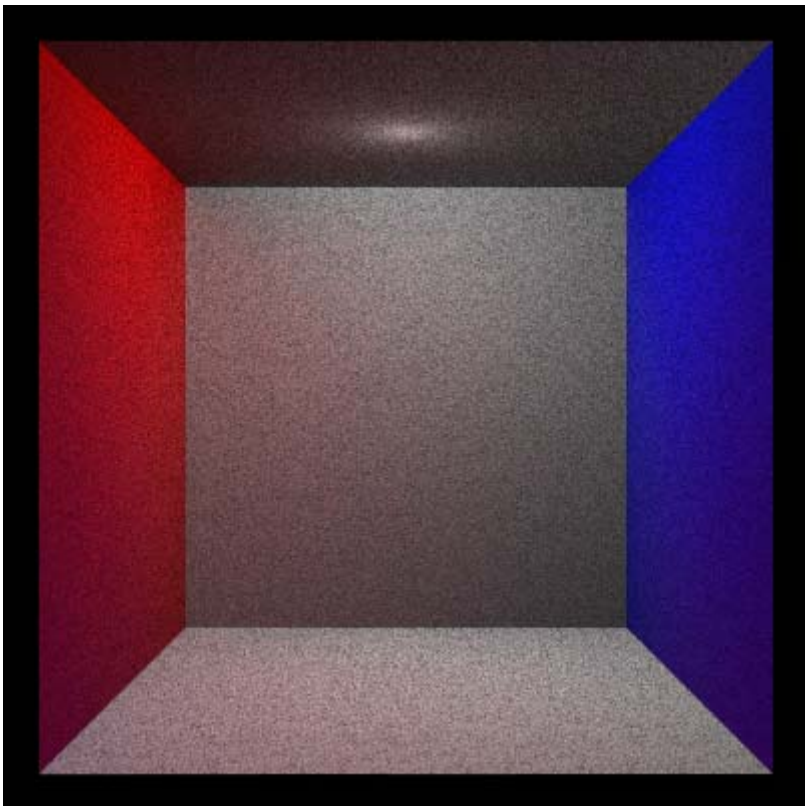
Ray Tracing löst implizit das Verdeckungsproblem. Die Schattenberechung wird mit Schattenstrahlen („Schattenfühler“) durchgeführt, die vom Schnittpunkt des Sehstrahls mit einem Objekt zu den Lichtquellen der Szene konstruiert werden. Trifft dieser Schattenstrahl ein undurchsichtiges Objekt, das zwischen Schnittpunkt und gerade betrachteter Lichtquelle liegt, dann trägt diese Lichtquelle nicht zur Beleuchtung des Schnittpunktes bei und der Schnittpunkt liegt im Schatten.
In der Regel werden beim konventionellen Ray Tracing nur ideale, d.h. scharf begrenzte Schatten auftreten. Jedoch kann der Programmierer nach eigenem Ermessen das Spektrum der Lichtquellenarten z.B. um Spotlights und Flächenlichtquellen erweitern. Schatten, die sich in Halb- und Kernschatten gliedern, lassen sich mit Flächenlichtquellen darstellen, wobei hier der Nachteil ist, dass meist aufwendige, stochastische Verfahren zur Abtastung der Flächenlichtquelle verwendet werden müssen.

Beim Ray Tracing und auch beim konventionellen sowie dem bidirektionalen Path Tracing nimmt die Schnittpunktberechnung zwischen einem Strahl und einem Objekt die meiste Berechnungszeit ein. Eine weitere Gemeinsamkeit der Algorithmen ist, dass sie betrachterabhängig berechnet werden. Sobald die Position des Augpunktes verändert werden soll, müssen Schatten bzw. die ganze Szene neu berechnet werden.

1.2 Path Tracing

Da beim konventionellen Ray Tracing nur ideal reflektierte und ideal gebrochene Strahlen weiter verfolgt werden, wird global diffuses Licht nicht berücksichtigt. Es findet also keine direkte Beleuchtung durch andere diffuse Flächen in der Umgebung statt. Dazu müsste ein diffuser Strahl rekursiv weiterverfolgt werden und sein Beleuchtungsbeitrag verrechnet werden. Ray Tracing sieht diese Option jedoch nicht vor.
Eine Erweiterung des Ray Tracing Algorithmus um die Verfolgung diffuser Strahlen führt zu Path Tracing. Beim Path Tracing werden ebenso wie beim Ray Tracing Strahlen durch Pixel eines Abtastgitters von einem Betrachterpunkt aus losgeschickt, durch die Szene verfolgt, mit Objekten geschnitten, reflektiert und rekursiv weiterverfolgt. Auch hier wird an jedem Schnittpunkt das lokale Beleuchtungsmodell berechnet, es findet Schattenberechnung statt und es gibt ein Abbruchkriterium für die Rekursion.
Der Unterschied zum Ray Tracing ist folgender: Das Path Tracing implementiert die diffuse Wechselwirkung, indem mehrere Strahlen durch jedes Pixel versendet werden, und bei der Reflexion eines Strahles an einem Schnittpunkt werden nicht rekursiv wie beim Ray Tracing ein reflektierter und transmittierter Strahl weiterverfolgt, sondern es wird nur ein einziger Strahl rekursiv verfolgt. Dieser einzelne Strahl ist entweder ein spekular reflektierter, ein diffus reflektierter oder ein refraktierter Strahl. Welcher Strahl dann weiterverfolgt wird, ob reflektierter, diffuser oder transmittierter, wird per Zufall mit Hilfe von russischem Roulette entschieden. Dabei wird mit einem Zufallsgenerator in jedem Rekursionsschritt eine gleichverteilte Zufallszahl aus einem begrenzten Intervall berechnet. In der Regel liegt diese Zufallszahl zwischen 0 und 1. Vor dem Starten des Algorithmus werden innerhalb des Intervalls [0,1] zwei Grenzen festgelegt, die das Intervall in drei Bereiche unterteilen. Je nachdem, in welchem Teilbereich die berechnete Zufallszahl liegt, wird entweder der reflektierte, der diffuse oder der transmittierte Strahl weiterverfolgt. Die Wahl dieser zwei Grenzen innerhalb des Intervalls [0,1] hat also Einfluss auf die Darstellung der Szene.

Mit Path Tracing lassen sich die gleichen optischen Effekte darstellen wie mit Ray Tracing. Weil aber jetzt auch diffuse Wechselwirkung berücksichtigt wird, kann zusätzlich Color Bleeding simuliert werden. Unter Color Bleeding versteht man das „Ausbluten“ von Farbe einer Fläche an sie benachbarte, andere Flächen. Wenn z.B. eine rote Wand in einer Cornell-Box eine gemeinsame Kante mit einer anderen, grauen Wand hat, so wird durch das direkte Beleuchten der roten Wand mit einer Lichtquelle in der Nähe einer Kante der Wand etwas rote Farbe in Richtung grauer Wand diffus reflektiert und auf ihr zu sehen sein. Dieser Effekt lässt sich auch in der Realität beobachten. Das folgende, mit Path Tracing gerenderte Bild (Abbildung 1.1) zeigt eine solche Cornell-Box mit Color Bleeding.



(Abbildung 1.1)
Cornell-Box, die mit Path Tracing gerendert wurde. Das Bild hat eine Auflösung von 400 x 400 und wurde mit 50 Strahlen pro Pixel abgetastet. Am Boden und an der hinteren Wand ist in der Nähe der roten Wand etwas Color Bleeding zu sehen.

Auch wenn Color Bleeding ein Effekt ist, der die Bilder realistischer aussehen lässt, so haben Ergebnisbilder des Path Tracing Algorithmus einen entscheidenden Nachteil gegenüber Ray Tracing: sie sind verrauscht. Daher werden beim Path Tracing mehrere Strahlen durch einen Pixel des Abtastgitters verschossen. Das kann auch beim Ray Tracing eingesetzt werden, um Aliasing-Effekte zu verringern. Bei Path Tracing jedoch hat diese Überabtastung das Ziel, das Rauschen zu reduzieren. Je mehr Strahlen pro Pixel versendet werden, desto geringer wird das Rauschen und desto angenehmer wird das Ergebnisbild für den Betrachter. Gleichzeitig steigt jedoch der Rechenaufwand.

James Kajiya führte 1986 als erster Path Tracing als eine Monte-Carlo-Methode ein [Quelle [1] im Quellenverzeichnis]. In derselben Arbeit definierte er die Rendering-Gleichung, die alle Rendering-Methoden und damit auch Ray Tracing, Path Tracing sowie bidirektionales Path Tracing zu approximieren oder zu lösen versuchen. Monte-Carlo-Methoden werden benutzt, um Integrale wie die Rendering-Gleichung zu lösen, die keine analytische oder numerische Lösung besitzen. Dabei wird der Durchschnitt zufälliger Stichproben des Integranden berechnet und anschließend der Mittelwert berechnet. Ein sichtbarer Fehler bei der Monte-Carlo-Integration ist, dass sie Varianz in den Lösungen erzeugt. In der Computergrafik wird Varianz in einem gerenderten Bild als Rauschen wahrgenommen.

1.3 Bidirektionales Path Tracing

Beim Ray Tracing und beim Path Tracing werden die Strahlen vom Betrachterpunkt aus in die Szene geschickt. Da die Wahrscheinlichkeit sehr gering ist, dass Lichtstrahlen, die von der Lichtquelle ausgesendet werden, das Auge treffen, wurde der Lichtweg umgekehrt. Ebenso gering ist jedoch die Wahrscheinlichkeit, dass Pfade, die vom Auge ausgehen, die Lichtquelle treffen.

Die Idee bei bidirektionalem Path Tracing ist, die Tatsache auszunutzen, dass bestimmte Pfade am einfachsten vom Auge aus zu samplen sind, wobei andere Pfade besser gesampelt werden können, wenn sie bei der Lichtquelle starten. Es ist ein Monte-Carlo-Rendering-Algorithmus, der die Ideen vom Aussenden und Einsammeln von Energie integriert, um photorealistische Bilder zu erzeugen. Bidirektionales Path Tracing wurde 1993 von Lafortune und Willems eingeführt [Quelle [2] im Quellenverzeichnis] und unabhängig davon 1994 von Veach und Guibas [Quelle [3] im Quellenverzeichnis] als Erweiterung zum Path Tracing Algorithmus.

Bidirektionales Path Tracing verfolgt Strahlen, die beim Auge starten, als auch Strahlen, die bei der Lichtquelle starten. Das ist ein Unterschied zum Path Tracing: Die Lichtquellen werden nicht ausschließlich beim Aufbau der Schattenstrahlen berücksichtigt, sondern als Startpunkte eines von Sehstrahlen unabhängigen Lichtstrahles betrachtet. Ebenso wie die Sehstrahlen interagieren die Lichtstrahlen mit den Oberflächen der Objekte und werden auf ihrem Weg durch die Szene weiterverfolgt. Zu diesem Zweck werden ein Startpunkt und eine Richtung auf den Lichtquellen gesampelt auf der Basis der Lichtquellenintensität. Der Startpunkt des Augenstrahles bleibt wie beim Ray Tracing und Path Tracing der Betrachterpunkt. Jetzt wird jeder Strahl (Licht- und Augenstrahl) einzeln rekursiv wie beim Path Tracing weiterverfolgt. Dabei darf die Rekursionstiefe der beiden Strahlentypen Lichtstrahl und Augenstrahl verschieden sein. Je nach Wahl dieser beiden Rekursionstiefen entstehen unterschiedliche optische Effekte im Ergebnisbild. Durch Variieren der Anzahl an Vertices, die von jeder Seite aus generiert werden, erhalten wir eine Gattung von Sampling-Techniken für Pfade jeder Länge. Jede Sampling-Technik hat verschiedene Wahrscheinlichkeits-Verteilungen über den Pfad-Raum und berücksichtigt eine andere Teilmenge der Faktoren des Integranden der Rendering-Gleichung. Samples von all diesen Techniken werden dann kombiniert durch Verwendung von Multiple Importance Sampling.

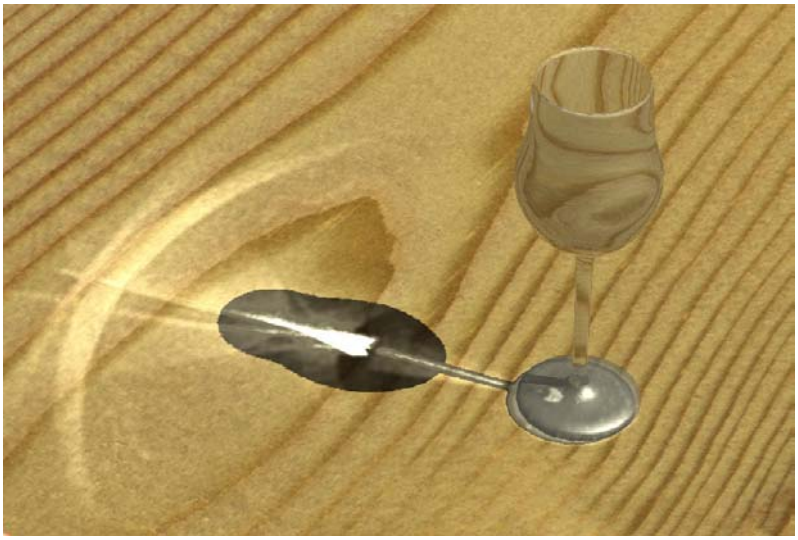
Nachdem die Zufallswege für den Lichtstrahl und den Augenstrahl konstruiert wurden, werden alle Schnittpunkte auf dem Lichtpfad (inklusive Startpunkt auf der Lichtquelle) mit allen Schnittpunkten auf dem Augenpfad (in der Regel ohne den Augenpunkt) durch Schattenstrahlen verbunden. Liegt ein undurchsichtiges Objekt zwischen den zwei den Schattenstrahl beschreibenden Punkten, so wird zum Augenstrahl kein Beleuchtungsbeitrag hinzugefügt. Trifft der Schattenstrahl kein Objekt, so wird mit einer anderen Berechnungsvorschrift als bei Ray Tracing und Path Tracing ein Beleuchtungsbeitrag berechnet. In dieser Studienarbeit wird zur Beleuchtungsberechnung die Formel von Henrik Wann Jensen realisiert: für zwei aufeinanderfolgende Schnittpunkte des Augenstrahles wird berechnet, wie groß die Leuchtdichte ist, die vom gerade betrachteten Schnittpunkt zu seinem Vorgänger gelangt [Quelle [9] im Quellenverzeichnis]. Die einzelnen Leuchtdichte-Werte zwischen je zwei aufeinanderfolgenden Schnittpunkten des Augenstrahles werden aufsummiert zu einer Gesamtleuchtdichte. Da *jeder* Stützpunkt des Augenstrahles (ohne Augpunkt selbst) mit *jedem* Stützpunkt des Lichtstrahles durch Schattenstrahlen verbunden wird, wächst die Zahl der Schattenstrahlen je nach Rekursionstiefe des Licht- bzw. Augenstrahles schnell an. Es gibt daher verschiedene Techniken, diese Zahl der

Schattenfühlertests zu reduzieren [Quelle [4] im Quellenverzeichnis]. Die Schattenstrahlen wiederum bestimmen, welche Beleuchtungsbeiträge zum endgültigen Beleuchtungswert hinzuaddiert werden müssen. Der endgültige Wert wird wie bei den anderen beiden Rendering-Algorithmen auch in einen Farbwert für das zu setzende Pixel konvertiert.

Durch bidirektionales Path Tracing werden verschiedene Beleuchtungsbeiträge berücksichtigt, nicht nur von primären Lichtquellen, sondern mit Berücksichtigung von Wahrscheinlichkeiten auch von sekundären, tertiären usw. Lichtquellen. Der Algorithmus kann indirekte Lichtprobleme wesentlich effizienter und robuster als gewöhnliches Path Tracing behandeln.

Die Bandbreite der optischen Effekte wird durch bidirektionales Path Tracing erweitert: Kaustiken. Um Kaustiken mit traditionellem Path Tracing zu sampeln, muss ein zufällig diffus reflektierter Strahl weiterverfolgt werden. Er muss dann durch eine Reihe von Spiegelreflexionen wandern, bevor er eine Lichtquelle trifft. Dieses Ereignis tritt jedoch mit einer sehr geringen Wahrscheinlichkeit ein. Wenn aber beim bidirektionalen Path Tracing die Pfade zusätzlich von der Lichtquelle aus starten, wird die Wahrscheinlichkeit größer, dass das Licht von einer spekularen Oberfläche reflektiert wird, auf eine diffuse Oberfläche trifft und dann auf die Bildebene projiziert wird. Damit ist zum Entstehen einer Kaustik folgender Weg eines Strahles notwendig: Er startet bei einer Lichtquelle, wird von spekularen Oberflächen einmal oder mehrmals ideal reflektiert, trifft dann auf eine diffuse Oberfläche und wandert danach zum Augpunkt. Wenn eine Kaustik mit dem bidirektionalen Path Tracing dargestellt werden soll, dann ist sie also auf einer diffusen Oberfläche der Szene zu sehen.

In der Physik bezeichnet der Begriff „Kaustik“ bei einem optischen System die einhüllende Fläche derjenigen Punkte, in denen sich die Bildstrahlen eines parallel einfallenden Lichtbündels schneiden. Als Katakaustik bezeichnet man die Einhüllende der an ebenen Kurven (spiegelnden Flächen) reflektierten Strahlen. Die Kaustik bei brechenden Systemen (Linsen) heißt Diakaustik. Die folgenden Bilder zeigen solche Kaustiken.



(Abbildung 1.2)

(Quelle: http://www-m2.mathematik.tu-muenchen.de/Forschungsprojekte/Sigma/Grafik/Tuffenta/glas.jpg)

Dieses Bild zeigt Kaustiken, die nach Brechung von Lichtstrahlen an einem Weinglas entstehen. Die meisten Lichtstrahlen werden viermal gebrochen, bevor sie auf die Grundfläche (Holztextur) auftreffen. Das Glas besitzt einen Brechungsindex von 1,2.

(Abbildung 1.3)

(Quelle: http://www-m2.mathematik.tu-muenchen.de/Forschungsprojekte/Sigma/Grafik/Tuffenta/wasser.jpg)

Ein komplexes Beispiel für Kaustiken ist in diesem Bild dargestellt. Die Wasseroberfläche wurde dabei als bikubische B-Spline-Fläche modelliert. Der Boden und die Seitenwände des Beckens wurden mit einer Bildtextur versehen, um ein Fliesenmuster zu erhalten.

Obwohl der Begriff Kaustik eine sehr spezifische Bedeutung in der Optik hat, wurde er in der Computergraphik bei der Bild-Synthese oft allgemeiner benutzt, um einen Bezug zu der Beleuchtung von diffusen Oberflächen durch von spiegelnden Oberflächen reflektiertes Licht herzustellen.

1.4 Konventionen in der Computergraphik

Zum Abschluss dieses Kapitels werden ein paar Konventionen erörtert, die in der Computergraphik gängig sind und auch in dieser Studienarbeit gelten sollen.

In der Computergraphik ist ein Strahl ein Liniensegment mit einer Position (Startpunkt), einem Ausmaß (einer Länge) und einer Richtung (Richtungsvektor). Ein Pfad ist eine Sammlung von Segmenten, wobei jeder Endpunkt eines Segmentes als Startpunkt für das nächste Segment dient. Die Zahl der Pfadschritte ist in der Regel begrenzt, um unendliche Rechnungen zu vermeiden.

Jeder einzelne Pfad wird in Bezug auf globale Beleuchtungsalgorithmen anders beschrieben. Paul S. Heckbert schlägt den Gebrauch von regulären Ausdrücken vor, um die Beschreibung von Pfaden zu vereinfachen [Quelle [6] im Quellenverzeichnis]. Dabei wird beschrieben, welche Wechselwirkung von Oberfläche zu Oberfläche ein Beleuchtungsalgorithmus realisiert. Das ist eine einfache und nichtmathematische Einteilung und ermöglicht den Vergleich und die Klassifizierung der Algorithmen zur globalen Beleuchtungssimulation.

Heckberts Reihenschreibweise dient dazu, alle Wechselwirkungen aufzulisten, die entlang des Weges eines Lichtstrahles von der Lichtquelle zum Auge auftreten. Hierbei wird der Weg von der Lichtquelle zum ersten Auftreffen eines Lichtstrahles „L“ genannt und bezeichnet damit die Emission des Lichtes von der Lichtquelle. „E“ bezeichnet den Weg des Lichtes vom letzten Stützpunkt des Pfades in der Szene zum Augpunkt, also die Absorption des Auges. Der Buchstabe „D“ bezeichnet die diffuse Reflexion, bei der das Licht gleichmäßig in jede Richtung zerstreut wird und „S“ kennzeichnet eine Spiegelreflexion oder Brechung. Bei der

Spiegelung gibt es bei einer gegebenen Einfallsrichtung eine einheitliche Ausfallsrichtung. Damit kann jeder besondere Pfad-Typ textuell durch einen String von Symbolen des Alphabetes repräsentiert werden, bestehend aus den Buchstaben {L, D, S, E}. Als Beispiel: Licht, das nach einer einzigen Reflexion von einer diffusen Oberfläche beim Auge ankommt, folgt dem Pfad LDE, und Licht, das nach einer einzigen Reflexion von einer spiegelnden Fläche ankommt, folgt dem Pfad LSE.
Ein vollständiger Algorithmus zur globalen Beleuchtung müsste jeden Lichtweg einschließen, der als regulärer Ausdruck L(D|S)*E geschrieben werden kann, wobei | für „oder“ steht und * für Wiederholung. Dieser Ausdruck setzt einfach fest, dass Licht, das die Quelle verlässt und das Auge erreicht, einem Pfad folgen könnte, der jede Anzahl (Null mit einbezogen) von diffusen und spekularen Reflexionen in jeder Reihenfolge beinhaltet.

Ray Tracing behandelt die Pfade LS*E und LDS*E, zusammengefasst also LD?S*E, wobei „?“ für 0 oder 1 mal steht. Da bei klassischem Ray Tracing keine diffuse Wechselwirkung berücksichtigt wird, ist hier die Berechnung der Pfade LD*E oder LS*DE nicht möglich. Diese Pfade lassen sich erst mit Path Tracing bzw. bidirektionalem Path Tracing darstellen. Dabei werden Kaustiken durch (LS*DE)-Pfade repräsentiert.

Die Ergebnisbilder bei bidirektionalem Path Tracing sind ebenso wie beim konventionellen Path Tracing verrauscht, aber in geringerem Maße. Bidirektionales Path Tracing benutzt weniger Strahlen pro Pixel als Path Tracing und erzielt im Ergebnisbild trotzdem das gleiche Maß an Rauschen. Mit anderen Worten: Wenn die gleiche Menge an Strahlen durch ein Pixel traversiert wird, so erhält man mit bidirektionalem Path Tracing bessere Bilder als mit Path Tracing. Jedoch arbeitet Path Tracing immer noch schneller, weil bei bidirektionalem Path Tracing bei jedem Strahl pro Pixel zwei Pfade kombiniert werden müssen.
Für Probleme mit kleinen Quellen von starker indirekter Beleuchtung (wie bei Kaustiken) ist bidirektionales Path Tracing besser geeignet als Path Tracing. Für Außenszenen, bei denen der Beobachter nur einen kleinen Teil des Modells sieht, ist es besser, Path Tracing zu verwenden.

In diesem Kapitel wurde bidirektionales Path Tracing als globaler Beleuchtungsalgorithmus vorgestellt und mit anderen, verwandten Algorithmen verglichen. Aufgrund der verschiedenen optischen Effekte, die er simulieren kann, kann er als Verbesserung von Ray Tracing und Path Tracing angesehen werden. Der Nachteil ist, dass verrauschte Ergebnisbilder entstehen und die Berechnungszeit größer ist als bei Path Tracing und Ray Tracing. Im nächsten Kapitel werden die einzelnen Schritte des bidirektionalen Path Tracings mathematisch beschrieben.

Kapitel 2
Teil 1
Mathematische Analyse des bidirektionalen Path Tracing Algorithmus

In diesem Kapitel wird die Mathematik erläutert, die die theoretische Grundlage für das bidirektionale Path Tracing ist. Zunächst wird die Rendering-Gleichung von James Kajiya [Quelle [1] im Quellenverzeichnis] aufgeführt, weil dies die fundamentale Gleichung zur Beschreibung der Beleuchtungsvorgänge bei globaler Beleuchtung ist. Anschließend werden Definitionen einzelner radiometrischer Größen aus der Physik genannt, die in dieser Studienarbeit gelten sollen, damit eine Basis durch physikalische Größen festgelegt ist, mit denen im weiteren Verlauf gearbeitet werden kann**.** Danach wird die Grundidee der Monte-Carlo-Methoden erläutert und in wieweit diese Techniken beim bidirektionalen Path Tracing eine Rolle spielen. Dann werden Notationen für Augen- und Lichtstrahl und ihre Strahlenabschnitte eingeführt. Dabei wird auch kurz auf die Generierung der Pfade nach Lafortune und Willems [Quelle [2] im Quellenverzeichnis] eingegangen. Diese Pfadnotationen werden für weitere Funktionen und Formeln zur Beleuchtungsberechnung verwendet; die Formeln sollen später bei der Implementation berücksichtigt werden.

2.1.1 Die Rendering-Gleichung

In diesem ersten Abschnitt wird eine mathematische Formulierung für ein Modell des globalen Beleuchtungsproblems vorgestellt. Es wurde erstmals 1986 von James Kajiya vorgestellt und ist bekannt als die Rendering-Gleichung. Sie erfasst die globale Beleuchtung, indem sie beschreibt, was an einem Punkt x der Oberfläche geschieht. Es handelt sich dabei um eine völlig allgemeine mathematische Aussage über das Problem.

Das Integral in Kajiyas Originalschreibweise ergibt sich aus:

$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_s \rho(x,x',x'')I(x,x'')dx''\right] \quad \text{(Gleichung 2.1)}$$

Dabei gilt:

- I(x, x') ist die Transportintensität oder die Intensität des Lichtes, die von Punkt x' zu Punkt x geht. Kajiya nennt das die nicht abgeschlossene Zwei-Punkt-Transportintensität.
- g(x, x') ist die Sichtbarkeitsfunktion zwischen x und x'. Wenn x und x' sich nicht „sehen" können, liegt diese bei null. Wenn sie sichtbar sind, variiert g als das umgekehrte Quadrat der Distanz zwischen ihnen oder wird vereinfachend auf eins gesetzt.
- ε(x, x') ist die Transferstrahlung von x' zu x und steht im Verhältnis zu der Intensität des Lichtes, das von Punkt x' selbst in Richtung auf x ausgestrahlt wird.
  ρ(x, x', x'') ist der Term für die Streuung in Bezug auf die Richtung x' und x''. Es ist die Intensität der Energie, die von einem Punkt oder einer Richtung x'' ausgeht und in Richtung auf x durch einen Oberflächenpunkt bei x' gestreut wird. Kajiya nennt dies den nicht abgeschlossenen Drei-Punkt-Transport-Reflexionsgrad. Im Allgemeinen wird Licht, das von einem Punkt auf der Oberfläche eines Objektes reflektiert wird, durch eine bidirektionale Verteilungsfunktion oder BRDF (Bidirectional reflection

distribution function) kategorisiert. Dieser Ausdruck betont, dass das Licht, das in einer bestimmten Richtung reflektiert wird nicht nur eine Funktion dieser Richtung ist, sondern auch der Richtung des einfallenden Lichtes. Der Drei-Punkt-Transport-Reflexionsgrad steht mit der BRDF in Verbindung durch:

$$\rho(x, x', x'') = \rho(\theta'_{IN}, \phi'_{IN}, \theta'_{ref}, \phi'_{ref}) * \cos(\theta) * \cos(\theta'_{ref}) \quad \text{(Gleichung 2.2)}$$

Dabei sind θ' und ϕ' die Azimut- und Höhenwinkel am Punkt x' und θ der Winkel zwischen der Oberflächennormalen im Punkt x und der Linie x' x.

Das Integral wird über die Szene s gebildet, also über alle Punkte auf allen Oberflächen in der Szene oder, was dasselbe bedeutet, über alle Punkte der Halbkugel, deren Mittelpunkt am Punkt x' liegt. Die Rendering-Gleichung sagt aus, dass die Transportintensität von Punkt x' zu Punkt x gleich ist zu allem Licht, das von x' zu x ausstrahlt plus dem Licht, das von x' zu x durch alle anderen Oberflächen in der Szene gestreut wird – also von einer Richtung x'' ausgeht.

Folgende wichtigen Punkte ergeben sich aus der Rendering-Gleichung:

1. Die Komplexität des Integrals bedeutet, dass es nicht analytisch berechnet werden kann, und die meisten praktischen Algorithmen reduzieren die Komplexität auf irgendeine Art und Weise. Die direkte Auswertung der Gleichung kann mithilfe von Monte-Carlo-Methoden stattfinden. Dieser Ansatz wird in einer der nächsten Abschnitte erläutert.
2. Es handelt sich um eine Beschreibung des Problems, die vom Blickwinkel unabhängig ist. Der Punkt x' ist jeder Punkt der Szene. Globale Beleuchtungsalgorithmen sind entweder blickwinkelunabhängig (z.B. Radiosity) oder blickwinkelabhängig (Ray Tracing und Path Tracing), wobei nur die Punkte x' berechnet werden, die von der Blickposition aus sichtbar sind. Blickwinkelabhängigkeit kann als Weg betrachtet werden, die innere Komplexität der Rendering-Gleichung zu reduzieren.
3. Es ist eine rekursive Gleichung – um I(x, x') zu bestimmen, muss I(x', x'') berechnet werden, wofür dieselbe Gleichung benutzt wird. Dies führt zu einer der am weitesten verbreiteten praktischen Methoden, das Problem zu lösen, die darin besteht, Licht von der Bildebene aus in der umgekehrten Richtung der Lichtausbreitung zu verfolgen und dabei einem Weg zu folgen, der von Objekt zu Objekt reflektiert wird.

2.1.2 Definition physikalischer Grundgrößen

Damit direkte und indirekte Beleuchtung berechnet werden kann, werden photometrische bzw. radiometrische Größen aus der Physik in Gleichungen der Computergraphik eingesetzt, um das Maß an Beleuchtung abzuschätzen und Beleuchtung als eine quantitative Größe zu erfassen. Diese Größen werden jetzt eingeführt.

Der Strahlungsfluss (englisch: Radiant Flux)
Der Begriff Strahlungsfluss stammt aus der Radiometrie. Der äquivalente Begriff aus der Photometrie ist „Lichtstrom".
Der Strahlungsfluss Φ ist definiert als Strahlungsmenge Q pro Einheitszeit dt

$$\Phi = \frac{dQ}{dt} \quad \text{(Gleichung 2.3)}$$

und wird gemessen in Watt [W = J * s^{-1}]. Zum Beispiel ist das die Quantität, die das Verhältnis beschreibt, mit der Energie von einer finiten Fläche S δ $|^3$ emittiert oder absorbiert wird.
Die Notation dieser Gleichung kann präziser geschrieben werden als

$$\Phi(t) = \frac{dQ(t)}{dt}.$$

Daraus wird deutlich, dass Φ und Q Funktionen der Zeit sind.

Die Bestrahlungsstärke (englisch: Irradiance)
Bestrahlungsstärke ist eine Bezeichnung aus der Radiometrie, der analoge Begriff aus der Photometrie ist „Beleuchtungsstärke". Sie ist eine Empfängergröße, weil sie den auftreffenden Strahlungsfluss auf ein Flächenelement bezeichnet.
Die Bestrahlungsstärke E ist definiert als Strahlungsfluss pro Einheitsfläche:

$$E(x) = \frac{d\Phi(x)}{dA(x)}, \qquad \text{(Gleichung 2.4)}$$

mit der Einheit [W*m^{-2}]. Es ist immer definiert mit Bezug zu einem Punkt x auf einer Oberfläche S, die eine spezifische Normale N(x) besitzt.

Die Strahlungsdichte (englisch: Radiance)
Die Strahlungsdichte L ist die fundamentale radiometrische Größe. „Leuchtdichte" ist der zugehörige Begriff aus der Photometrie. Für Licht-Transport-Berechnungen ist sie die bei weitem wichtigste Größe, definiert durch:

$$L(x,\omega) = \frac{d^2\Phi(x,\omega)}{dA^{\perp}_{\omega}(x)d\sigma(\omega)}, \qquad \text{(Gleichung 2.5)}$$

wobei $A^{\perp}_{\omega}$ die senkrechte Projektion der Fläche ist, senkrecht zu ω. D.h., um die Strahldichte bei (x, ω) zu messen, zählen wir die Zahl der Photonen pro Einheitszeit, die durch eine kleine Oberfläche $dA^{\perp}_{\omega}(x)$, senkrecht zu ω, dessen Richtungen in einem kleinen Raumwinkel $d\sigma(\omega)$ um ω umfasst werden. Strahldichte ist definiert als das Verhältnis des Strahlungsflusses dΦ, das durch diese Photonen repräsentiert wird, dividiert durch das Produkt $dA^{\perp}_{\omega}(x)d\sigma(\omega)$. Die korrespondierende Einheit ist [W * m^{-2} *sr^{-1}].

Beim Messen der Strahldichte, die eine reale Oberfläche S verlässt, ist die folgende Gleichung besser geeignet:

$$L(x,\omega) = \frac{d^2\Phi(x,\omega)}{|\omega * N(x)|dA(x)d\sigma(\omega)}, \qquad \text{(Gleichung 2.6)}$$

wobei wie vorher auch A die Flächenmessung auf S ist, und N(x) die Oberflächennormale bei x ist. Das setzt die projizierte Fläche $dA^{\perp}_{\omega}$ ins Verhältnis zur gewöhnlichen Fläche dA, gemäß

$$dA_{\omega}^{\perp}(x) = |\omega * N(x)| dA(x) \,. \qquad \text{(Gleichung 2.7)}$$

Alternativ kann der Faktor $|\omega * N(x)|$ absorbiert werden in die projizierte Raumwinkel-Messung, wie sie oben definiert wurde, was zu folgender Gleichung führt:

$$L(x,\omega) = \frac{d^2\Phi(x,\omega)}{dA(x)d\sigma_x^{\perp}(\omega)} \,. \qquad \text{(Gleichung 2.8)}$$

Das ist die gebräuchlichste Definition beim Arbeiten mit der Strahldichte auf realen Oberflächen, weil es die natürliche Flächenmessung A benutzt.

Alternativ kann die Strahldichte auch über die Strahlstärke I und den Flächeninhalt der Lichtquelle definiert werden. Je kleiner die leuchtende Fläche A im Vergleich zur Strahlstärke I ist, desto heller erscheint dem Betrachter die Lichtquelle. Die Strahldichte wird dann definiert durch die Strahlstärke I und den projizierten Flächeninhalt (dA*cos θ):

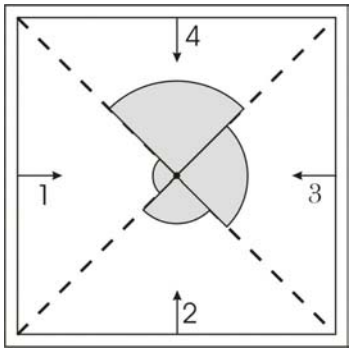
$$L = \frac{dI}{dA * \cos\theta} \,. \qquad \text{(Gleichung 2.9)}$$

Nach Einführung dieser radiometrischen Größen wird die Rendering-Gleichung umformuliert, damit diese Größen sich auch in Bezug auf die Rendering-Gleichung anwenden lassen.

2.1.3 Strahlungsdichte, Einstrahlung und die Strahlungsgleichung

Die Original-Form der Rendering-Gleichung ist in globalen Beleuchtungsmethoden nicht besonders nützlich. In diesem Abschnitt werden Definitionen eingeführt, die eine Umformulierung der Rendering-Gleichung in die Strahlungsdichte-Gleichung ermöglichen.

Die Strahlungsdichte L in einem Punkt ist eine Funktion der Richtung, und damit kann eine Funktion der Strahlungsdichte-Verteilung für einen Punkt definiert werden. Diese ist im Allgemeinen unstetig. Als Beispiel ist die Abbildung 2.1 aufgeführt. Solch eine Verteilungsfunktion besteht in allen Punkten im dreidimensionalen Raum. Die Strahlungsdichte ist daher eine fünfdimensionale Größe.



(Abbildung 2.1)
Eine zweidimensionale Strahlungsdichteverteilung für einen Punkt im Zentrum eines Raumes, wobei jede Wand eine andere Strahlungsdichte zeigt.

Die Einstrahlung E ist das Integral der einfallenden Strahlungsdichte über alle Richtungen:

$$E = \int_{\Omega} L_{IN} * \cos\theta \quad d\omega \qquad \text{(Gleichung 2.10)}$$

Dabei ist:

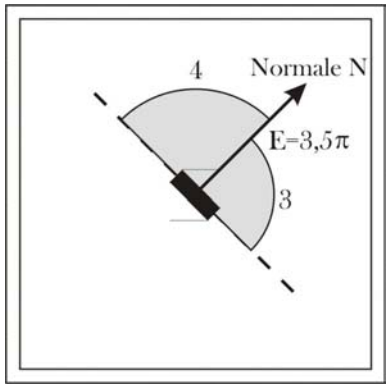
L_{IN} die einfallende oder Feld-Strahlungsdichte aus Richtung ω
θ der Winkel zwischen der Oberflächennormalen und ω

Wenn L_{IN} konstant ist, ergibt sich für eine diffuse Oberfläche:

$$L_{diffus} = \rho * E / \pi \qquad \text{(Gleichung 2.11)}$$

Die Unterscheidung zwischen diesen beiden Größen ist wichtig für globale Beleuchtungsalgorithmen, da die Form des Algorithmus als „ausstrahlend“ oder als „sammelnd“ klassifiziert werden kann. Ausstrahlend steht für die Verteilung der Strahlungsdichte von einer Oberfläche, „sammelnd“ für die Integration der Einstrahlung oder die Ansammlung (Akkumulation) des Lichtflusses an der Oberfläche.

Ein wichtiger praktischer Punkt in Bezug auf Strahlungsdichte- und Einstrahlungs-Verteilungsfunktionen besteht darin, dass die erste gewöhnlich unstetig, die letztere hingegen im Allgemeinen – mit Ausnahme von Schattengrenzen – stetig ist. Dies wird in Abbildung 2.2 dargestellt, die zeigt, dass in diesem einfachen Beispiel die Einstrahlungs-Verteilungsfunktion wegen des Effekts der Mittelwertbildung durch die Integration stetig sein wird.



(Abbildung 2.2)
Einstrahlung E ist der durch den Cosinus gewichtete Durchschnitt der Strahlungsdichte, in diesem Fall 3,5π.

Die Rendering-Gleichung kann zur Strahlungsdichte-Gleichung umgeschrieben werden, die in ihrer einfachsten Form lautet:

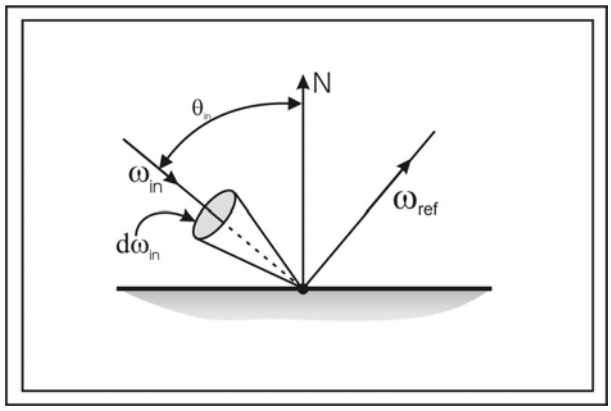
$$L_{ref} = \int \rho * L_{IN} \qquad \text{(Gleichung 2.12)}$$

Wenn man die Richtungsabhängigkeit einschließt, ergibt sich:

$$L_{ref}(x, \omega_{ref}) = L_e(x, \omega_{ref}) + \int_{\Omega} \rho(x, \omega_{in} \rightarrow \omega_{out}) L_{in}(x, \omega_{in}) \cos(\theta_{in}) d\omega_{in}$$

(Gleichung 2.13)

wobei die Symbole in Abbildung 2.3 definiert werden.



(Abbildung 2.3)
Symbole, die verwendet werden, um die Richtungsabhängigkeit zu definieren.

Dies kann so modifiziert werden, dass die Integration über alle Oberflächen stattfindet – was im Allgemeinen in praktischen Algorithmen angenehmer ist – statt über alle Einfallswinkel. Dies ergibt die Rendering-Gleichung ausgedrückt durch die Strahlungsdichte:

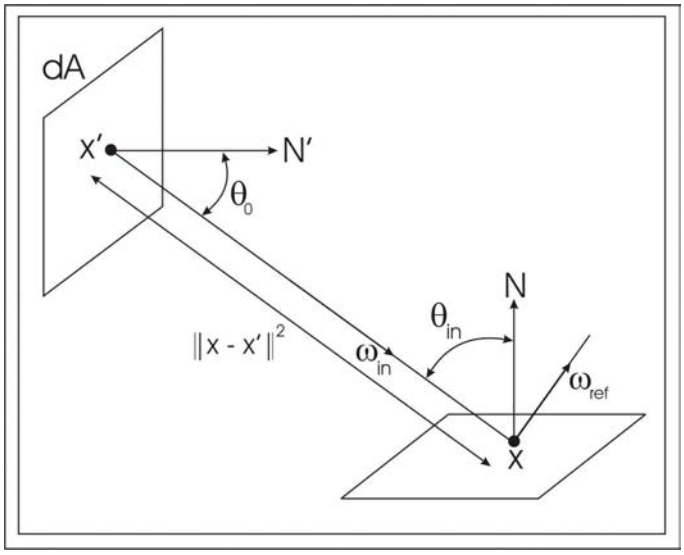
$$L_{ref}(x,\omega_{ref}) = L_e(x,\omega_{ref}) + \int_S \rho(x,\omega_{in} \to \omega_{out}) L_{in}(x',\omega_{in}) g(x,x') \cos(\theta_{in}) \frac{\cos(\theta_0) dA}{\|x-x'\|^2}$$

(Gleichung 2.14)

Darin ist nun die Sichtbarkeitsfunktion eingeschlossen. Dies kommt dadurch zustande, dass der Raumwinkel $d\omega_{in}$ durch den projizierten Bereich der differenziellen Oberflächenregion ausgedrückt wird, die in der Richtung ω_{in} sichtbar ist (Abbildung 2.4):

$$d\omega_{IN} = \frac{\cos(\theta_0) dA}{\|x-x'\|^2}$$

(Gleichung 2.15)



(Abbildung 2.4)

$\frac{\cos(\theta_0) dA}{\|x-x'\|^2}$ *ist der projizierte Bereich von dA, der in Richtung* ω_{in} *sichtbar ist.*

2.1.4 Skizzierung des Algorithmus

Nachdem nun die radiometrischen Größen festgelegt wurden und die Rendering-Gleichung in die Strahlungsgleichung zwecks besserer Implementierungsmöglichkeiten umgeformt wurde, wird jetzt der bidirektionale Path Tracing Algorithmus beschrieben. Ich habe mich dabei vorwiegend am Kapitel 3.3 aus Henrik Wann Jensens Buch [Quelle [9] im Quellenverzeichnis] orientiert und an verschiedenen Stellen Ergänzungen aus dem Paper von Lafortune und Willems „Bidirectional Path Tracing“ eingefügt.

Es wird versucht, den Ablauf des bidirektionalen Path Tracing Algorithmus' als Leitfaden zu verwenden. Dabei soll jeder Zwischenschritt gesondert betrachtet werden. Der Ablauf des Algorithmus lässt sich wie folgt skizzieren:
Zuerst wird ausgehend vom Betrachterpunkt (der Kamera) ein Augenstrahl in die Szene geschickt. Zusätzlich wird unabhängig davon ein Lichtstrahl von einer Lichtquelle aus losgeschickt. Beide Strahlentypen werden auf ihrem Weg durch die Szene verfolgt und es wird ihre Interaktion mit geschnittenen Objekten berechnet. Anschließend wird mittels Schattenfühlern der Lichtaustausch zwischen den Schnittpunkten (Stützstellen) des Augenstrahles und den Stützstellen des Lichtstrahles berechnet. Der Gesamtwert des Lichtaustausches wird in einen Farbwert umgewandelt und als Pixelfarbe im Ergebnisbild gesetzt. Dieser Prozess wird für mehrere richtungsverschiedene Augenstrahlen und Lichtstrahlen wiederholt, bis das Ergebnisbild mit der gewünschten Auflösung abgetastet ist.
Jeder einzelne Schritt dieses Algorithmus wird jetzt detailliert untersucht. Dabei werden Notationen und Formeln festgelegt, die eine formale Beschreibung der Vorgänge ermöglichen.

2.1.5 Die Kamera und die Augenstrahlen

Eine Szene von mehreren Objekten wird beim bidirektionalen Path Tracing mit Sehstrahlen abgetastet. Diese Augenstrahlen werden vom Betrachterpunkt aus losgeschickt. Er kann als Standpunkt einer Kamera angesehen werden, die von der ganzen Szene ein Standbild aufnimmt. In der Regel wird das Lochkamera-Modell aus der Physik verwendet, um die Strahlengeometrie beim Betrachterpunkt zu erklären. Es kann jedoch auch anders als beim Lochkamera-Modell der Startpunkt der Augenstrahlen als optische Linse mit bestimmter Linsendicke und eigenem Brechungsverhalten interpretiert werden. Dann ist der Startpunkt der Augenstrahlen nicht mehr ein eindeutiger Punkt, sondern die Augenstrahlen können an verschiedenen Stellen auf der Linsenoberfläche beginnen. Hier wird im weiteren Verlauf der Erklärungen ein Lochkamera-Modell vorausgesetzt.

Die Strahlen, die beim Auge starten und zuerst mit Objekten der Szene geschnitten werden, heißen „primäre“ Augenstrahlen. Die Richtung eines primären Augenstrahles wird durch einen zweiten Punkt festgelegt, der auf dem Abtastgitter liegt. Das Abtastgitter erfüllt den gleichen Zweck wie beim Ray Trracing. Es wird fortlaufend zeilen- und spaltenweise mit Primärstrahlen abgetastet. Der primäre Augenstrahl wird auf seinem Weg durch die Szene weiterverfolgt und entweder mit Objekten geschnitten, oder er trifft kein Objekt und geht an den Szenenobjekten vorbei. Liegt ein Schnittpunkt vor, wird der Strahl in Abhängigkeit vom Material des Objektes reflektiert bzw. gebrochen und rekursiv weiterverfolgt. Liegt kein Schnittpunkt vor, dann wird die vordefinierte Hintergrundfarbe gesetzt. Nach dem ersten Schnitt mit einem Objekt entsteht ein sekundärer Augenstrahl, nach dem Schnitt dieses sekundären Strahles mit einem Objekt ein tertiärer, usw. Die Bezeichnungen „primär, sekundär, tertiär, ...“ werden auch beim Lichtstrahl verwendet. Wichtig ist hier zu beachten,

dass nachdem ein Strahl ein Objekt geschnitten hat, nicht rekursiv wie beim Ray Tracing der transmittierte *und* der reflektierte Strahl rekursiv weiter verfolgt werden, sondern dass wie beim Path Tracing mit russischem Roulette entschieden wird, ob nur der diffuse, nur der reflektierte oder nur der transmittierte Strahl rekursiv weiterverfolgt werden soll.

2.1.6 Die Lichtquellen und die Lichtstrahlen

Zeitgleich zum Aufbau eines primären Augenstrahles wird ein Lichtstrahl von einer Lichtquelle aus losgeschickt. Bei mehreren Lichtquellen in der Szene wird jede Lichtquelle separat behandelt. Es werden mehrere Lichtstrahlen von einer Lichtquelle aus losgeschickt, bevor die nächste Lichtquelle berücksichtigt wird. Jeder Schnittpunkt von einem Lichtstrahl mit einem Szeneobjekt wird in einer Liste zwischengespeichert. Daher spielt es nachher bei der Implementierung keine Rolle, welche Lichtquelle zuerst behandelt wird.
Eine Lichtquelle kann unterschiedliche Ausprägungen haben. Es kann eine Punktlichtquelle sein, bei der das Licht in alle Richtungen mit gleicher Intensität ausgesendet wird. Die Beleuchtungsberechnung an den Punkten der Szenenobjekte ist hier am einfachsten: es wird der vorher für die Punktlichtquelle festgelegte Intensitätswert verrechnet.
Als Spezialisierung einer Punktlichtquelle gibt es das Spotlight, bei dem die Intensität des ausgesandten Lichtes für einen beleuchteten Punkt in Abhängigkeit zu einer Ausstrahlrichtung gewichtet wird. Bei einem Spotlight wird ein Lichtkegel festgelegt, dessen Spitze im Ausgangspunkt der Lichtquelle liegt und dessen Öffnung in eine vordefinierte Richtung weist. Die Größe der Öffnung wird dabei durch einen Raumwinkel festgelegt. Alle Strahlen, welche das Spotlight durch diesen Kegel verlassen, beleuchten die getroffenen Objekte. Verlässt ein Lichtstrahl die Lichtquelle nicht durch diesen Kegel, trägt er nicht zur Beleuchtung der Szene bei und seine Intensität wird nicht für das Endresultat verrechnet. Die Intensität der Lichtstrahlen, die den Lichtkegel verlassen, kann noch detaillierter variiert werden, wenn ein Lichtstrahl, der genau in der Mitte des Kegels in die Szene wandert eine höhere Gewichtung erhält als ein Strahl, der nahe am Rand des Kegels die Lichtquelle verlässt.
Eine weitere Lichtquellenart ist die Flächenlichtquelle. Die Startpunkte der Lichtstrahlen dieser Lichtquelle sind hier nicht eindeutig durch einen geometrischen Punkt definiert, sondern durch eine planare Fläche mit festgelegter Größe und Form. Wegen des geometrischen Ausmaßes der Fläche lassen sich verschiedene Startpunkte für die primären Lichtstrahlen festlegen. In dieser Studienarbeit werden kreisförmige, rechteckige und quadratische Flächenlichtquellen implementiert. Die Form der Fläche lässt sich jedoch beliebig wählen. Der Flächenlichtquelle wird wie den Punktlichtquellen auch eine Intensität zugewiesen. Als Startpunkt eines Lichtstrahles wird zufällig ein Punkt auf der Fläche ausgewählt und wie der Ausgangspunkt einer Punktlichtquelle behandelt. Weil mehrere Lichtstrahlen von verschiedenen Startpunkten aus die Flächenlichtquelle verlassen werden, liefert jeder Lichtstrahl nur einen Bruchteil der Gesamtintensität der Flächenlichtquelle. Bei N Lichtstrahlen wird für jeden einzelnen Lichtstrahl eine Intensität von (I / N) verrechnet, wenn I die Gesamtintensität bezeichnet.
Die Flächenlichtquelle besitzt neben ihrer Form und Größe eine Normale, die die Ausrichtung der Fläche anzeigt. Eine Flächenlichtquelle teilt durch die Ebene, in der die Fläche liegt, den dreidimensionalen Raum in zwei Halbräume. Die Normale der Flächenlichtquelle zeigt nur in einen dieser Halbräume und daher leuchtet die Flächenlichtquelle nie in zwei entgegen gesetzten Richtungen, sondern nur in den Halbraum, in den die Normale zeigt. Die Intensität der Lichtstrahlen kann in Abhängigkeit zum Winkel zwischen Lichtstrahl und Normale gewichtet werden, damit wie bei der Definition der Strahldichte (vgl. Gleichung 2.5) die

senkrechte Projektion der Fläche berücksichtigt wird, falls ein Lichtstrahl die Fläche nicht parallel zur Normalen verlässt.

Die Richtung der primären Lichtstrahlen ist also beim Spotlight stärker eingeschränkt als bei idealer Punktlichtquelle und Flächenlichtquelle. Es ist aber sinnvoll, bei idealer Punkt- und Flächenlichtquelle in Bezug auf realistische und ausreichende Beleuchtung der Szene die Lichtstrahlen vorwiegend in Richtung der Szenenobjekte zu senden. D.h. die Ausrichtung der Flächenlichtquelle und die Richtung der Lichtstrahlen der Punktlichtquelle sind hier geeignet zu wählen. Dann ist auch eine Interaktion der Lichtstrahlen mit den Objekten der Szene wahrscheinlicher.

2.1.7 Monte Carlo-Methoden beim bidirektionalen Path Tracing

In diesem Abschnitt werden Monte-Carlo-Methoden kurz angesprochen und erläutert, welche Rolle sie beim bidirektionalen Path Tracing spielen.
Monte-Carlo-Methoden werden benutzt, um Integrale wie die Rendering-Gleichung zu lösen, die keine analytische oder numerische Lösung besitzen. Dabei wird der Durchschnitt zufälliger Stichproben des Integranden berechnet. Der sichtbare Effekt dieses Prozesses in dem endgültig gerenderten Bild ist Rauschen.
Bei Monte-Carlo-Methoden werden Stichproben aus der Geometrie der Szene und anhand der optischen Eigenschaften der Oberfläche genommen. Das Problem bei den Monte-Carlo-Methoden besteht darin, Techniken zu erfinden, mit denen eine genaue Schätzung des Integrals oder eine Schätzung mit geringer Abweichung schnell erzielt werden kann.

Die beiden Methoden zur Auswahl der Stichprobe eines Integranden sind das Verfahren der geschichteten Stichproben (Stratified Sampling) und das der Wichtigkeitsstichproben (Importance Sampling). Die einfachste Form geschichteter Stichproben teilt den Integrationsbereich in gleiche Schichten und schätzt jedes Teilintegral mit einer oder mehreren zufälligen Stichproben. So erhält jeder Teilbereich die gleiche Zahl an Stichproben.

Wie der Name andeutet, wählt das Verfahren der Wichtigkeitsstichproben (Importance Sampling) tendenziell Stichproben in wichtigen Bereichen des Integranden. Wichtigkeitsstichproben sind in Algorithmen globaler Beleuchtung, welche die Monte-Carlo-Ansätze verwenden, von großer Bedeutung. Denn obwohl die Rendering-Gleichung die globale Beleuchtung für jeden einzelnen Punkt einer Szene beschreibt, wird keine Lösung gebraucht, die gleich bleibend genau ist. Ein genaueres Ergebnis wird eher für eine hell erleuchtete spekulare Oberfläche als für eine schwach beleuchtete, diffuse Wand benötigt. Um zum Beispiel einen Monte-Carlo-Ansatz für das Ray Tracing einer spekularen Oberfläche zu nutzen, würde man die reflektierten Strahlen auswählen, die sich um die spekulare Reflexionsrichtung anhäufen, um auf diese Weise die (bekannte) BRDF in Bereichen zu erfassen, in denen sie wahrscheinlich einen hohen Wert zurückgibt. Die Stichproben werden im Allgemeinen so verteilt, dass ihre Dichte in den Bereichen am höchsten ist, wo die Funktion einen hohen Wert hat oder wo sie sich schnell und in bedeutendem Maße verändert.

Beim bidirektionalen Path Tracing werden die Richtungen der primären Lichtstrahlen so bestimmt, dass die Lichtstrahlen in Bereiche fallen, die vom Szenenaufbau her stark beleuchtet sein müssen. Bei einer Flächenlichtquelle zum Beispiel liefert ein Lichtstrahl, der fast parallel zur Normalen in die Szene startet, einen wichtigeren Beitrag zur Beleuchtung als ein Strahl, der fast rechtwinklig zur Normalen die Lichtquelle verlässt. Bei der Generierung

von primären Lichtstrahlen wird später bei der Implementierung genau darauf geachtet werden.
Ein weiteres Beispiel für die Auswahl von Stichproben ist das Bestimmen von Abtastpunkten in einem Pixel. Wenn ein Augenstrahl durch das Abtastgitter verfolgt wird und mit Objekten geschnitten wird, dann trägt dieser Strahl nur zu einem Bruchteil zur Gesamtbeleuchtung bei und ist damit nur eine Stichprobe für die Rendering-Gleichung. Beim bidirektionalen Path Tracing müssen daher wie beim normalen Path Tracing auch mehrere Strahlen pro Pixel versendet werden, um mehr Stichproben zu erhalten und die Rendering-Gleichung besser approximieren zu können.

Ein sichtbarer Fehler bei der Monte Carlo Integration ist, dass sie Varianz in den Lösungen erzeugt. In der Computergrafik wird Varianz in einem gerenderten Bild als Rauschen wahrgenommen.
Die Varianz ist eine mathematische Variable, die oft in der Statistik verwendet wird. Es ist definiert als das Quadrat der Standardabweichung, σ, und zeigt an, wie groß der Fehler sein kann. Die Approximation für die Abweichung mit n Abtaststellen ist:

$$\sigma \propto \frac{1}{\sqrt{n}}$$

d.h. Sigma ist proportional zum Kehrwert der Wurzel aus n.

Das Entwickeln der Approximation durch Inkrementieren der Anzahl der Abtaststellen n ist ein sehr kostspieliger Weg. Halbieren des Fehlers erfordert vier Mal mehr Abtaststellen. Die Methoden zur Reduzierung der Varianz sind Stratified Sampling und Importance Sampling.
Das heißt, je mehr Augenstrahlen n durch ein einzelnes Pixel gesendet werden, desto geringer wird die Varianz und desto geringer wird das Rauschen im Ergebnisbild. Das bidirektionale Path Tracing liefert eine geringere Varianz für Szenen, die indirekt beleuchtet sind. Denn falls beim Path Tracing die zu berechnende Szene indirekt beleuchtet wird, könnten nur wenige Schattenstrahlen eine Lichtquelle erreichen, was zu einer großen Varianz der Ergebniswerte führt. Bidirektionales Path Tracing versucht dieses Problem zu beheben, indem nicht nur Strahlen vom Auge aus, sondern auch von der Lichtquelle aus gesampelt werden.

Beim Monte-Carlo-Ansatz werden also Stichproben der Rendering-Gleichung benutzt, um einen Schätzwert für die Intensität eines Pixels zu berechnen, und daher müssen viele Strahlen pro Pixel versendet werden, die durch die Szene reflektieren. James Kajiya benutzte zum Beispiel für seinen Path Tracing Algorithmus [Quelle [1] im Quellenverzeichnis] 40 Strahlen pro Pixel.

2.1.8 Die Pfadnotationen

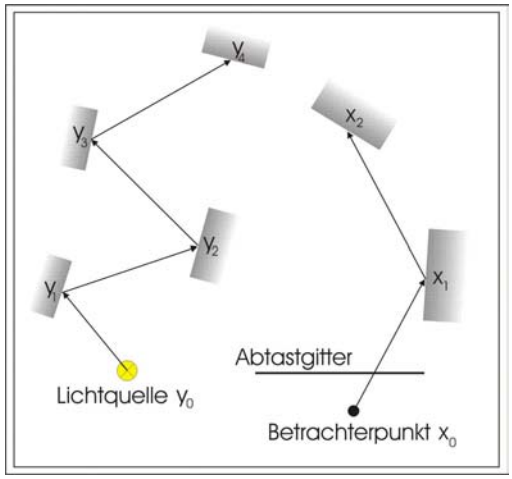
Es werden jetzt Bezeichnungen für die Augenstrahlen und Lichtstrahlen und ihre Schnittpunkte mit den Szenenobjekten eingeführt. Die Gesamtheit des Weges vom ersten bis zum letzten Schnittpunkt wird sowohl in Bezug auf den Betrachter (Augpunkt) als auch in Bezug auf die Lichtquelle als Strahl bezeichnet. Die einzelnen Abschnitte, in die der Strahl unterteilt wird, wenn man die Schnittpunkte als Stützstellen („Vertices“) betrachtet, werden Pfade genannt. Jeder Pfad wird durch zwei Stützstellen beschrieben. Ein Strahl setzt sich also aus mehreren Pfaden zusammen.

2.1.8.1 Notationen für den Augenstrahl

Ein vollständiger Augenstrahl wird im Folgenden mit „x“ bezeichnet. Der Startpunkt des Augenstrahles beim Betrachter wird mit „x_0“ bezeichnet. Jeder weitere Schnittpunkt des Augenstrahles wird mit „x_i“ bezeichnet bei aufsteigendem Index i, i 0 {1, 2, 3, … n}. An jedem Schnittpunkt x_i wird die Interaktion des Augenstrahles mit der Oberfläche des getroffenen Objektes in Abhängigkeit der entsprechenden Materialeigenschaften berechnet. Dabei ist n die Gesamtzahl der Stützstellen des Augenstrahles in der Szene. Der Wert von n ist endlich und kann variiert werden.

2.1.8.2 Notationen für den Lichtstrahl

Ein vollständiger Lichtstrahl wird im Folgenden mit „y“ bezeichnet. Der Startpunkt des Lichtstrahles auf der Lichtquelle wird mit „y_0“ bezeichnet. Analog zum Augenstrahl werden weitere Schnittpunkte des Lichtstrahles mit Objekten bei aufsteigendem Index j mit „y_j“ bezeichnet, j 0 {1, 2, 3, … m}. Trifft ein Lichtstrahl auf eine Oberfläche, so wird die Intensität, die der Lichtstrahl transportiert, in Abhängigkeit von den Materialeigenschaften verrechnet und der nach der Reflexion verbleibende Anteil weitertransportiert zum nächsten Schnittpunkt.



(Abbildung 2.5)
Die Abbildung zeigt einen Licht- und einen Augenstrahl mit den jeweiligen Schnittpunkten mit durch Schraffur angedeuteten Objekten.

Die maximale Rekursionstiefe setzt den Werten n und m eine obere Grenze. Diese Werte können beliebig variiert werden. Dadurch erhält man verschiedene Effekte in den Ergebnisbildern.

Die Schnittpunkte werden einfach durch einen Punkt mit drei Koordinaten beschrieben. Die Strahlen, die nach einer Reflexion an einem Objekt entstehen, wurden je nach Materialeigenschaft entweder spekular oder diffus reflektiert oder gebrochen. Bei einer spekularen Reflexion gilt das einfache Reflexionsgesetz: Einfallswinkel gleich Ausfallwinkel. Die Winkel werden gegen die Normale aufgetragen. Bei diffuser Reflexion wird zufällig eine Richtung für den weiterführenden Strahl ausgewählt, weil durch diffuse Streuung kein Reflexionsgesetz wie bei spekularer Reflexion gilt. Bei einer Refraktion wird das Brechungsgesetz aus der Physik angewendet.

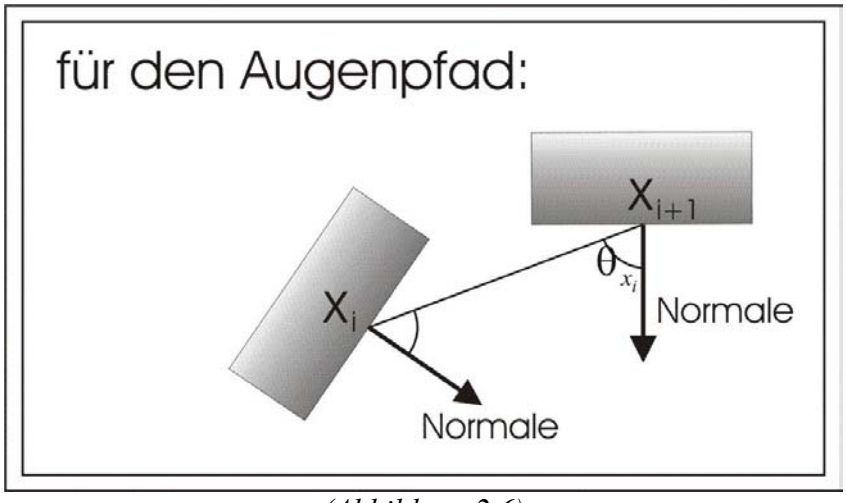
2.1.9 Pfadgenerierung nach Lafortune und Willems

Nachdem die Notationen für Strahlen und Schnittpunkte eingeführt wurden, wird jetzt die Generierung der Strahlen bzw. der Pfade untersucht. Der Ansatz dazu stammt aus einer wissenschaftlichen Arbeit von Lafortune und Willems [Quelle [2] im Quellenverzeichnis].

Lafortune und Willems führten als erste bidirektionales Path Tracing ein. Die Grundidee war hier zu Jensens Ansatz identisch: Strahlen werden zur selben Zeit vom Auge (dem Betrachterpunkt) und von einer Lichtquelle aus losgeschickt. Ebenso werden alle Stützstellen des Augenpfades mit allen Stützstellen des Lichtpfades durch Schattenstrahlen verbunden, um den Lichtaustausch zu berechnen.

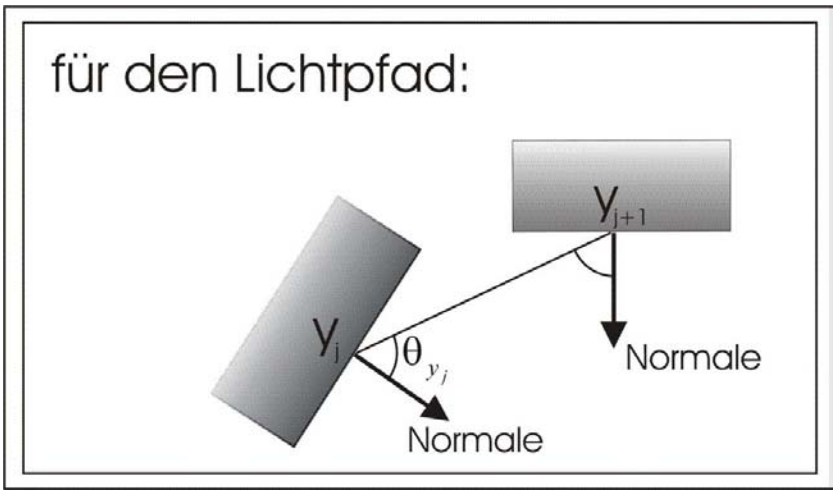
Beim Erstellen der einzelnen Pfade für Licht- und Augenstrahlen verwenden Lafortune und Willems folgenden Ansatz: die Pfadwege werden als Zufallswege betrachtet und diese und die zugehörigen Abtastpunkte werden anhand von Wahrscheinlichkeits-Verteilungs-Funktionen, so genannten pdf's (probability distribution functions) gewählt. Dann werden die effektiven Beiträge zum Strahlungsfluss berechnet. Es kann mathematisch bewiesen werden, dass dies zu einer korrekten Lösung der Rendering-Gleichung führt.

Neben den bereits eingeführten Notationen x_i und y_j für den Augen- bzw. Lichtpfad, die hier beibehalten werden, ergänzen Lafortune und Willems die Beschreibung der Pfade um die Winkel, die zwischen den Normalen an den Stützstellen und den eintreffenden bzw. ausfallenden Pfaden gemessen werden.
Bei Lafortune wird der Winkel θ_{x_i} für Stützstellen des Augenstrahles zwischen den Normalen im Punkt x_{i+1} und dem Pfad gemessen, der zum Punkt x_i zurückführt (vgl. Abb. 2.6).



(Abbildung 2.6)
Der Winkel θ_{x_i} wird beim Punkt x_{i+1} gemessen, zwischen Normale und dem Strahl, der zum Punkt x_i weiterführt. Lafortune und Willems verfolgen den Augenstrahl vom letzten Punkt des Augenstrahles zum Auge zurück und nicht vom Auge aus, daher diese Winkelbezeichnung mit Index x_i.

Der Winkel θ_{y_j} ist der Winkel, der auf dem Lichtstrahl am Punkt y_j gemessen wird und zwischen der Normalen und dem Pfad aufgetragen wird, der zum Punkt y_{j+1} weiterführt (vgl. Abbildung 2.7). Sowohl beim Lichtstrahl, als auch beim Augenstrahl wird der Winkel zwischen Normale und weiterführendem Pfad gemessen.



(Abbildung 2.7)
Der Winkel θ_{y_j} wird beim Punkt y_j gemessen, weil der Lichtstrahl vom Punkt y_j zum Punkt y_{j+1} wandert.

2.1.9.1 Erstellen der Zufallswege mit pdf und spdf

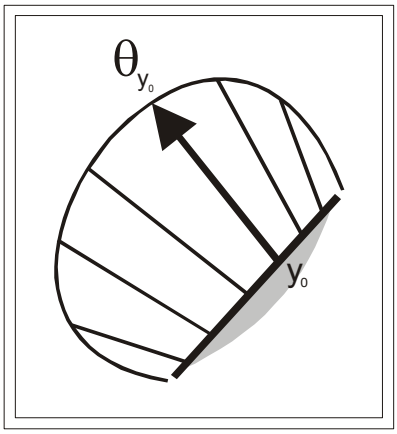
Jetzt werden die Wahrscheinlichkeits-Verteilungs-Funktionen vorgestellt, mit denen Lafortune und Willems die Zufallswege bestimmen. Zuerst werden die Richtungen und Teilstrecken des primären Licht- und Augenstrahles anhand einer pdf ermittelt.
Die Werte y_0 und θ_{y_0} für den ersten Lichtpfad von der Lichtquelle aus werden gemäß der folgenden pdf gesampelt, die auf dem Prinzip des Importance Sampling basiert:

$$\text{pdf}(y, \theta_y) = \frac{L_e(y, \theta_y) * \left|\theta_y * N_y\right|}{L_{Norm}} \qquad \text{(Gleichung 2.16)}$$

mit dem Normalisierungsfaktor L_{Norm} der pdf

$$L_{Norm} = \int_A \int_{\Omega_y} L_e(y, \theta_y) * \left|\theta_y * N_y\right| * d\omega_y d\mu_y \quad \text{(Gleichung 2.17)}$$

wobei $L_e(y, \theta_y)$ die aussendende Strahldichte beim Punkt y in Richtung θ_y ist, und $\left|\theta_y * N_y\right|$ der Absolut-Wert des Kosinus des Winkels zwischen θ_y und dem Normalen-Vektor bei y ist.
Das Importance Sampling hebt Abtastpunkte hervor, die mehr Gewicht auf dem Ergebnis haben werden, und es stellt sicher, dass mehr Lichtpartikel von hellen Emittern in helle Richtungen verschossen werden, anstatt die Lichtpartikel einheitlich zu verteilen und ihre Beiträge zum Flux später zu gewichten.



(Abbildung 2.8)
Sampling von y_0 und θ_{y_0} gemäß der austretenden Strahldichte der Lichtquelle. Der Punkt y_0 ist ein zufällig gewählter Punkt auf der Fläche der Lichtquelle, die hier angedeutet wird durch die Linie mit der grauen Schraffur.

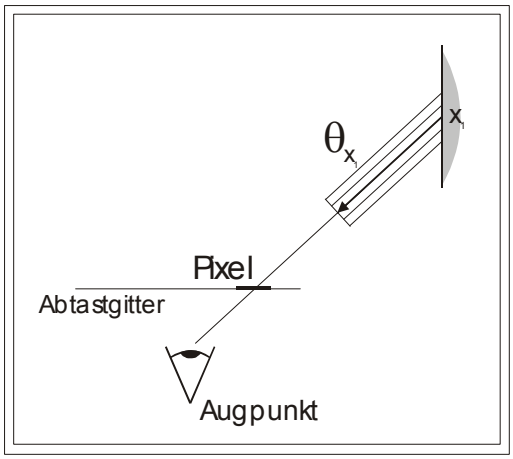
Der Vorteil ist hier also, dass Lichtstrahlen vorwiegend in die Richtungen geschossen werden, in die das meiste Licht fällt. Dadurch wird die Szene besser beleuchtet, als wenn die Lichtstrahlen gleichmäßig in alle Richtungen verteilt werden und anschließend gemittelt werden.
Analog werden für den ersten Augenpfad, der beim Betrachterpunkt startet, die Größen x_0 und θ_{x_0} gemäß der folgenden pdf gesampelt (vgl. Abbildung 2.9):

$$\text{pdf}(x, \theta_x) = \frac{g(x, \theta_x) * |\theta_x * N_x|}{G} \qquad \text{(Gleichung 2.18)}$$

mit dem Normalisierungsfaktor G dieser pdf

$$G = \int_A \int_{\Omega_x} g(x, \theta_x) * |\theta_x * N_x| * d\omega_x d\mu_x \,, \qquad \text{(Gleichung 2.19)}$$

wobei die Funktion $g(x, \theta_x)$ eins ist für alle Punkte-Paare und Richtungen (x, θ_x) auf den Oberflächen, die zum Strahlungsfluss beitragen und ansonsten null ist.



(Abbildung 2.9)
Der erste Punkt x_0 ist der Augpunkt selbst. Lafortune und Willems verfolgen die Augenstrahlen zum Auge hin, anstatt vom Auge auszugehen. Der Pfeil von x_1 aus kann daher umgedreht werden.

2.1.9.2 Die spdf

Nachdem die Werte für y_0 und x_1 und die zugehörigen Winkel für die ersten Pfade des Licht- und Augenstrahles bestimmt wurden, werden jetzt alle weiteren Werte für die Stützstellen auf dem Lichtstrahl und dem Augenstrahl und die zugehörigen Winkel $\theta_{y_{j+1}}$ und $\theta_{x_{i+1}}$ bestimmt, die eindeutig die nächsten Stützstellen festlegen. Zu ihrer Berechnung wird die „subcritical pdf (spdf)“ verwendet. Für den Wert $\theta_{y_{j+1}}$ ergibt sich auf der Basis von Importance-Sampling:

$$pdf(\theta) = f_r(y_{j+1}, \theta_{y_j}, \theta) * \left| \theta_y * N_{y_{j+1}} \right| \qquad \text{(Gleichung 2.20)}$$

wobei $f_r(y, \theta_{IN}, \theta_{OUT})$ die bidirektionale Reflexions-Verteilungs-Funktion (BRDF) ist. Die pdf ist hier „subcritical“ weil nicht bis eins integriert wird über alle möglichen Winkel, zumindest für physikalisch gültige BRDF’s. Der aktuelle Wert der Integration gibt die Wahrscheinlichkeit an, mit der der Zufallsweg fortgesetzt wird, wodurch sichergestellt wird, dass der Zufallsweg terminiert. Diese Technik entspricht russischem Roulette.

Die subcritical pdf für $\theta_{x_{i+1}}$ wird auf ähnliche Weise gewählt:

$$pdf(\theta) = f_r(x_i, \theta, \theta_{x_i}) * \left| \theta_x * N_{x_i} \right| \qquad \text{(Gleichung 2.21)}$$

Die Zufallswege von Augen- und Lichtpfaden sind unabhängig voneinander. Die spdfs für die Richtungen entsprechen sich gegenseitig, wenn man die Variablen umbenennt und den Lichtweg bei der BRDF umkehrt. Dies ergibt aus der Reziprozität der BRDF: der Strahlungsfluss in der einen Richtung muss identisch sein mit dem in der umgekehrten Richtung. Das bedeutet, dass nach einer Umbenennung der Variablen der eine Zufallsweg mit dem gleichen Algorithmus berechnet werden kann wie der andere Zufallsweg.
Die pdf und die spdf nehmen beide Werte zwischen 0 und 1 an und geben damit Wahrscheinlichkeiten an. Je höher also der Wert dieser Funktionen bei Einsetzen eines Wertes für den Winkel theta ist, desto höher ist der Beleuchtungsbeitrag für die Szene. Es ist also eine Art Optimierungsproblem, den Winkel theta zu finden, der einen pdf-Wert nahe eins ergibt.

2.1.10 Die Visibilitätsfunktion und Beleuchtungsberechnung nach Jensen

Nachdem die Generierung der Pfade für Licht- und Augenstrahl von Lafortune und Willems vorgestellt wurde, geht es jetzt um die Beleuchtungsberechnung für das Endresultat. Die folgenden Formeln sind aus Jensens Buch [Quelle [9] im Quellenverzeichnis] entnommen und werden später bei der Implementierung berücksichtigt. Sie beziehen sich nicht auf die Generierung der Pfade anhand der pdfs, die gerade eingeführt wurden. Der Vollständigkeit wegen wird aber die Beleuchtungsberechnung von Lafortune nach der Betrachtung der Formeln von Jensen ergänzt.

Da jetzt die Schnittpunkte der Strahlen mit den Szenenobjekten bekannt sind, ist der nächste Schritt, sie zu verknüpfen, um den Beleuchtungsaustausch zwischen ihnen zu berechnen. Jeweils ein Schnittpunkt x_i des Augenpfades wird mit allen Schnittpunkten y_j des Lichtpfades

durch Schattenstrahlen verknüpft. Das wird für jede Stützstelle des Augenstrahles wiederholt, inklusive dem Auge selbst.
Für jedes Schnittpunkt-Paar x_i und y_j wird geprüft, ob ein undurchsichtiges Objekt zwischen den zwei Punkten liegt, das den Lichtaustausch zwischen diesen beiden Punkten verhindern könnte. Diese Sichtbarkeit wird mit Hilfe der Visibilitätsfunktion $V(x_i, y_j)$ berechnet. Dabei wird ein Schattenstrahl eingesetzt, der die gleiche Funktion wie beim Raytracing hat. Schneidet dieser Schattenstrahl keines der Objekte der Szene, dann findet Lichtaustausch zwischen den beiden betrachteten Punkten x_i und y_j statt.

Die Visibilitätsfunktion verbindet so die Informationen des Licht- und des Augenstrahles.
Sie berechnet die individuellen Beiträge der Lichtpfad-Vertices zum Augenpfad. Falsch wäre, einfach alle Punkte x_i und y_j zu verbinden und den Wert des Licht-Vertex zu nehmen. Stattdessen wird die reflektierte Strahldichte bei x_i, die bezogen wird auf y_j, berechnet als:

$$L_{i,j}(x_i \to x_{i-1}) = f_r(y_j \to x_i \to x_{i-1}) * V(x_i, y_j) * \frac{\left|(y_j \to x_i) * \vec{n}_{x_i}\right|}{\left\|x_i - y_j\right\|^2} * I(y_j \to x_i)$$

(Gleichung 2.22)

Hier bezieht sich x_{i-1} auf die Stützstelle des Augenstrahles, die vor x_i liegt. Der Vektor $\vec{n}_{x_i}$ ist die Normale im Punkt x_i. $I(y_j \to x_i)$ ist die Strahlungsintensität, die von y_j ausgeht in die Richtung von x_i.
Die Strahlungsintensität $I(y_j \to x_i)$ für einen Vertex auf dem Lichtpfad wird berechnet als:

$$\mathrm{I}(\mathrm{y_j} \to x_i) = \Phi_i(y_j) * \left|(y_j \to x_i) * \vec{n}_{y_j}\right| * f_r(y_{j-1} \to y_j \to x_i)$$

(Gleichung 2.23)

f_r bezeichnet die BRDF. Sie bezieht sich auf drei Stützpunkte. In Gleichung 2.22 bezieht sie sich auf zwei Stützstellen des Augenstrahles und eine des Lichtstrahles, und in Gleichung 2.23 werden zwei Stützstellen des Lichtpfades mit einer Stützstelle des Augenpfades verknüpft. Die BRDF wird hier als das Verhältnis von einfallender Beleuchtungsstärke und ausfallender Strahldichte verstanden. Die ersten zwei Parameter von f_r dienen zur Bestimmung der einfallenden Größe und mit dem zweiten und dritten Parameter wird die ausfallende Größe bestimmt. $\Phi_i(y_j)$ ist der Strahlungsfluss des einfallenden Photons bei y_j.
Für die Augen- und Lichtstrahlen gibt es Sonderfälle, die dann eintreten, wenn die Indices i und j die Werte 0 annehmen. Bei gegebener Gleichung 2.22 (mit Berücksichtigung des Spezialfalles i = 0), können wir die gewichtete Summe der Beiträge von allen Pfaden berechnen:

$$L_P = \sum_{i=0}^{N_i} \sum_{j=0}^{N_j} w_{i,j} * L_{i,j}$$

(Gleichung 2.24)

wobei L_P eine Abschätzung für den Pixel ist. Diese Gleichung setzt voraus, dass wir perfektes Importance Sampling für alle BRDF's benutzt haben (anderenfalls müsste man Teilpfade aus dem Augenpfad ablösen, die nicht x_0 enthalten.)

2.1.11 Beleuchtungsberechnung nach Lafortune und Willems

Da bereits die Pfadnotation von Lafortune und Willems erläutert wurde, wird hier jetzt auch beschrieben, wie sie das Endergebnis für die Beleuchtung berechneten. Bei diesem Ansatz werden wie bei Jensen auch Stützstellen auf dem Lichtpfad mit Stützstellen auf dem Augenpfad durch Schattenstrahlen verbunden, um den Lichtaustausch zu berechnen.

Lafortune und Willems zeigten in einer anderen Arbeit „Reducing the Number of Shadow Rays“ [Quelle [4] im Quellenverzeichnis], wie die Anzahl der Schattenstrahlen reduziert werden kann. Wenn zum Beispiel 5 Stützstellen für den Augenstrahl vorliegen und 5 Stützstellen für den Lichtstrahl, dann werden insgesamt 25 Schattenstrahlen benötigt, um für dieses eine Strahlenpaar aus Licht- und Augenstrahl den Beleuchtungsaustausch zu berechnen. Wenn das Abtastgitter zum Beispiel eine Auflösung von 400 Zeilen mal 400 Spalten hat, dann werden 160 000 Augenstrahlen vom Augpunkte aus durch das Gitter wandern. Im einfachsten Fall gehört zu jedem Augenstrahl, der rekursiv weiterverfolgt wird, genau ein Lichtstrahl. Das bedeutet, dass bei 160 000 Strahlenpaaren aus Licht- und Augenstrahl – mit jeweils 5 Stützstellen für jeden Strahl – insgesamt 5 * 5 * 160 000 = 4 Millionen Schattenstrahlen bestimmt werden müssen, um den Austausch an Beleuchtung zwischen zwei verbundenen Punkten zu berechnen. Wenn jetzt noch Supersampling verwendet wird, steigt diese Zahl um einen weiteren Faktor.
Das Problem ist hier nicht das Erstellen der Schattenstrahlen: es ist einfach, zwischen zwei Punkten eine Verbindungslinie zu ziehen. Das Problem ist, dass jeder Schattenstrahl mit den Szenenobjekten geschnitten werden muss. Die Schnittpunktberechnung zählt beim Ray Tracing und bei Path Tracing Algorithmen zu den aufwendigsten und zeitintensivsten. Lafortune und Willems zeigten in ihrer Arbeit „Reducing the Number of Shadow Rays“, wie die Anzahl der Schattenstrahlen verringert werden kann, um den Berechnungsaufwand zu reduzieren. Hier wird nur die Grundidee gezeigt.
Als Beispiel dient folgendes Szenario:
Der Augenstrahl x hat N Stützstellen und der Lichtstrahl y hat M Stützstellen. Dann werden N*M Schattenstrahlen aufgebaut. Lafortune und Willems schlugen drei verschiedene Optimierungsmöglichkeiten vor, um die Anzahl der Schattenstrahlen zu verringern. Jede ist eine stochastische Prozedur, die Schattenstrahlen durch Importance Sampling anhand ihrer Beleuchtungsbeiträge aussucht.

1. Die Beleuchtungsbeiträge pro Stützstelle y_j auf dem Lichtstrahl werden gruppiert. Dann wird zufällig ein Schattenstrahl aus dieser Gruppe ausgewählt und mit den Szenenobjekten geschnitten. Sein Ergebnis steht repräsentativ für die ganze Gruppe.
2. Die Beleuchtungsbeiträge pro Stützstelle x_i auf dem Augenstrahl werden gruppiert, ein Schattenstrahl aus dieser Gruppe wird stochastisch ausgesucht und steht repräsentativ für die Gruppe.
3. Die dritte Möglichkeit sieht vor, dass alle Beleuchtungsbeiträge von Augen- und Lichtstrahl gruppiert werden und nur ein zufällig gewählter Schattenstrahl ausgesucht wird.

In der Regel steigt durch Ignorieren von Schattenstrahlen die Varianz an. Weil aber der Arbeitsaufwand geringer geworden ist, können mehr Licht- bzw. Augenstrahlen beim Supersampling eingesetzt werden.

Jetzt wird gezeigt, wie Lafortune und Willems die radiometrischen Größen berechnen, die nach Umwandlung in einen Farbwert ins Ergebnisbild gesetzt werden sollen. Lafortune und Willems unterscheiden zwischen einem „primary estimator“ und einem „secondary

estimator“. Der erste Begriff meint den Wert, der berechnet wird bei Einsatz von einem einzigen Augenstrahl pro Pixel. Wenn kein Supersampling des Abtastgitters eingesetzt wird, dann ist der „primary estimator“ der Wert, der als Endresultat in einen Farbwert umgerechnet wird und für den aktuellen Pixel gesetzt wird. Der „secondary estimator“ wird durch Mittelung von mehreren „primary estimators“ berechnet und ist daher der nach Supersampling berechnete Wert für einen Pixel. Es werden jetzt die Formeln dafür gezeigt und auch die Sonderfälle betrachtet, die bei einer bestimmten Anzahl von Stützstellen auftreten.

2.1.11.1 Der Hauptwert (primary estimator) und der Nebenwert (secondary estimator)

Der „primary estimator“ für den Strahlungsfluss kann dann als Summe von gewichteten partiellen Werten berechnet werden (die eckigen Klammern kennzeichnen einen skalaren Wert):

$$\langle \Phi \rangle = \sum_{j=0}^{m} \sum_{i=0}^{n} \omega_{ij} * \langle C_{ij} \rangle \qquad \text{(Gleichung 2.25)}$$

wobei n die Anzahl der Stützstellen auf dem Augenstrahl ist und m die Anzahl der Stützstellen auf dem Lichtstrahl.
Die Faktoren $\langle C_{ij} \rangle$ sind die Werte für den Strahlungsfluss, der durch j Reflexionen auf dem Lichtpfad und i Reflexionen auf dem Augenpfad berechnet wurde.
Für den Fall (i = 0 und j = 0) ist die Lichtquelle direkt sichtbar. Das ist ein Unterschied zum Raytracing. Für den Fall (i = 0 und j > 0) entsteht ein Lichtstrahl, der das Auge erreicht.

Hier müssen drei Sonderfälle unterschieden werden:

1. i = 0, j = 0: $\langle C_{00} \rangle$ = G x $L_e(x_0, \theta_{x_0})$. Dieser Term berechnet den Strahlungsfluss, der von einer Lichtquelle empfangen wurde, die direkt durch das aktuelle Pixel auf dem Abtastgitter zu sehen ist.
2. j = 0, i > 0: $\langle C_{0i} \rangle = L' \times G \times L_e(y_0, \theta_{y_0 \to x_{i-1}}) \times f_r(x_{i-1}, \theta_{y_0 \to x_{i-1}}, \theta_{x_{i-1}}) \times \frac{\left|\theta_{y_0 \to x_{i-1}} * N_{y_0}\right| * \left|\theta_{y_0 \to x_{i-1}} * N_{x_{i-1}}\right|}{\left\|y_0 - x_{i-1}\right\|^2} * v(y_0, x_{i-1})$

   mit $L' = \frac{L}{\int_{\Omega_y} L_e(y_0, \theta_y) * \left|\theta_y * N_{y_0}\right| d\omega_y}$,

wobei $\theta_{y \to x}$ der Winkel ist, der die Richtung vom Punkt y zum Punkt x festlegt. Der Faktor v(y, x) ist die bereits angesprochene Visibilitätsfunktion, die den Wert eins annimmt, wenn der Punkt y den Punkt x „sieht“, wenn also kein undurchsichtiges Objekt zwischen y und x den kürzesten Weg von y nach x versperrt. Ansonsten ist diese Funktion null. Der Wert für die Funktion wird durch Mittelwert-Bildung des Schattenstrahles zwischen Punkt x und y gefunden.
Der obige Term ist eine Wertberechnung für den Strahlungsfluss, den das Auge von der Lichtquelle über den Augenpfad erreicht, wie bei klassischem Path Tracing.

3. i > 0, j > 0: $\left\langle C_{ij} \right\rangle = L \times G \times f_r(y_j, \theta_{y_{j-1}}, \theta_{y_j \to x_{i-1}}) \times f_r(x_{i-1}, \theta_{y_j \to x_{i-1}}, \theta_{x_{i-1}}) \times \frac{\left|\theta_{y_j \to x_{i-1}} * N_{y_j}\right| * \left|\theta_{y_j \to x_{i-1}} * N_{x_{i-1}}\right|}{\left\|y_j - x_{i-1}\right\|^2} * v(y_j, x_{i-1})$

Dieser Term ist eine Wertberechnung für den Strahlungsfluss, der das Auge von der Lichtquelle aus erreicht, über i Reflexionen auf dem Augenpfad und j Reflexionen auf dem Lichtpfad.

Der „primary estimator“ des Strahlungsflusses besitzt noch eine große Varianz, die im Ergebnisbild in Form von Rauschen sichtbar wird. Wie bei allen Monte-Carlo-Methoden berechnen Lafortune und Willems daher einen zweiten Wert durch Mittelung der Ergebnisse von verschiedenen „primary estimators“ für ein einzelnes Pixel. Die Varianz des nun entstandenen Ergebnisses wird durch einen Faktor $\sqrt{N}$ reduziert, wobei N die Zahl der primary estimators ist. Für Standard-Path Tracing liegt N typischerweise zwischen 40 und 500, aber die optimale Anzahl hängt größtenteils von der Komplexität der Szene und der gewünschten Genauigkeit ab. Einige heuristische adaptive Sampling-Techniken werden gewöhnlich angewendet, um einen Ausgleich zwischen dem Berechnungsaufwand und der Qualität der Resultate zu finden.

2.1.12 Die Gewichtungen

Sowohl Jensen als auch Lafortune und Willems berücksichtigen bei der Berechnung der Strahldichte Gewichtungsfaktoren w_{ij} (vgl. Gleichungen 2.24 und 2.25). Der Einsatz von Gewichtungsfaktoren soll sicherstellen, dass Beiträge, die wichtiger oder gravierender in Bezug auf das Endergebnis sind, auch stärker berücksichtigt werden und mit einem größeren Gewichtungsfaktor verrechnet werden.
Die Gewichtungen $w_{i,j}$ müssen normalisiert werden:

$$\sum_{i=0}^{N} w_{i,N-i} = 1 \qquad \text{für N = 0, 1, 2, 3, …} \quad \text{(Gleichung 2.26)}$$

Diese Bedingung garantiert, dass die Gewichtungen für Pfade der Länge 0, 1, 2, 3, … sich zu 1 summieren. Die physikalische Bedeutung, die dahinter steckt, ist die, dass die Menge der Gewichtungen für die Werte des Strahlungsflusses, die über eine bzw. zwei, drei, vier, … Reflexionen beim Auge ankommen, sich alle zu eins summieren müssen.

Es ist einfach ersichtlich, dass mit

$$w_{i,j} = \begin{cases} 1 & \text{für i = 0} \\ 0 & \text{sonst} \end{cases} \qquad \text{(Gleichung 2.27)}$$

der einfache Standard Path Tracing Algorithmus gemeint ist.

Diese Wahl der Gewichtungen benutzt jedoch nicht vollständig die Information des Sampling-Prozesses. Die folgende Alternative benutzt beide Pfadwege effizienter:

$$\omega_{ij} = \begin{cases} \prod_{k=0}^{i-2} W_k & \text{für} \quad j = 0 \\ 0 & \text{für} \quad i = 0, \text{und} \\ (\prod_{k=0}^{i-2} W_k) * (1 - W_i) & \text{sonst}, \end{cases} \qquad \text{(Gleichung 2.28)}$$

wobei die Gewichtungen W_i (i=0, 1, 2, … n-1) noch ein paar Freiheitsgrade übriglassen.

Die Idee bei dieser besonderen Wahl ist, dass jeder Punkt auf dem Augenpfad, die Werte für die indirekte Beleuchtung über den Rest des Augenpfades und über den Lichtpfad gewichtet werden. Für spekulare Oberflächen würde man eher auf dem Wert aufbauen, der durch Verfolgen des Augenpfades gefunden wird. Für diffuse Oberflächen ist der durch die Beiträge des Lichtpfades gefundene Wert wahrscheinlich wichtiger. Daher wird die Gewichtung W_i proportional zu einer Messung des Grades der Spekularität einer Oberfläche am Punkt x_i auf dem Augenpfad gewählt. Für stark spekulare Oberflächen ist es fast eins, für diffuse Oberflächen geht es auf null zurück. Tests bei realistischen Szenen haben gezeigt, dass diese Technik stark die Qualität von Bildern verbessert, besonders beim Rendern von Szenen, die Spiegel enthalten.

Die Wahl der Gewichtungen hat starken Einfluss auf die Varianz der kombinierten Messgröße. Veach und Guibas führten eine Technik zum Kombinieren von Messgrößen ein, basierend auf verschiedenen Sampling-Techniken [Quelle [13] im Quellenverzeichnis]. Sie zeigten, dass die „Power Heuristik“ gute Resultate für bidirektionales Path Tracing ergibt. Mit der Power Heuristik werden die Gewichtungen berechnet als:

$$w_{i,j} = \frac{p_{i,j}^{\beta}}{\sum_{k=0}^{i+j} p_{k,j+i-k}^{\beta}} \qquad \text{mit der Potenz } \beta = 2. \quad \text{(Gleichung 2.29)}$$

$p_{i,j}$ ist die Wahrscheinlichkeitsdichte für das Generieren des Pfades $x_0, x_1, \ldots, x_i, y_j, y_{j-1}, \ldots,$ y0. Diese Pfadwahrscheinlichkeit kann berechnet werden durch Multiplizieren der Wahrscheinlichkeit mit der jeder Vertex auf dem Pfad generiert wird:

$$p(x_i \to x_{i+1}) = p_{f_r}(x_i \to x_{i+1}) * \frac{\left|(x_i \to x_{i+1}) * \vec{n}_{x_i}\right| * \left|(x_i \to x_{i+1}) * \vec{n}_{x_{i+1}}\right|}{\left\|x_i - x_{i+1}\right\|^2}$$

(Gleichung 2.30)

Hier ist $p_{f_r}(x_i \to x_{i+1})$ die Wahrscheinlichkeits-Dichte für das Sampeln der Richtung x_i -> x_{i+1} mit Bezug zum projizierten Raumwinkel. Wichtig ist hier, dass die Wahrscheinlichkeiten für die Startpunkte der Pfade verschieden sind; sie hängen von den Sampling-Techniken ab, die für die Lichtquelle und für den Pixel benutzt werden.

Ein anderes Problem tritt auf, wenn die Distanz zwischen Augen-Vertex und Licht-Vertex sehr kurz ist (so wie bei Ecken). Das Quadrat dieser Distanz ist der Nenner in Gleichung (2.22) und der Wert kann unendlich groß werden. Dieses Problem kann glücklicherweise behoben werden durch Verwendung der „Power-Heuristik“ (Gleichung 2.29), weil die Gewichtungen, die mit diesem kurzen Pfaden assoziiert werden, sehr niedrig sein werden.

2.1.13 Der Farbwert

Da jetzt alle notwendigen Berechnungen angesprochen wurden, von der Pfadgenerierung bis zur Strahldichte, bleibt nur noch die Umwandlung in einen Farbwert, der im Ergebnisbild gesetzt werden soll. Mit einem Ausgabegerät wie dem Bildschirm kann jedoch ein einzelner Bildpunkt nicht mit einer Strahldichte-Verteilung angesprochen werden, sondern es ist die Umrechnung in einen Farbwert notwendig. Die Aufgabe ist also jetzt, den Strahldichte-Wert in einen Farbwert zu konvergieren.
Die Farbgebung bei einem Monitor geschieht auf der Grundlage der additiven Farbmischung. Die Farbe setzt sich hier aus einer roten, einer grünen und einer blauen Komponente zusammen (RGB-Monitor). Für jede Komponente können unterschiedliche Intensitätsstufen gewählt werden, in der Regel 256 pro Komponente. Farben können also wie ein dreidimensionaler Vektorraum aufgefasst werden. Die Vektoren des Farbraumes heißen „Farbvalenzen“. Die Länge eines Vektors ist ein Maß für die Strahldichte und heißt „Farbwert“, seine Richtung ist die „Farbart“.
Die genaue Berechnung der Farbe anhand eines Beispiels wird im zweiten Teil dieses Kapitels noch mal angesprochen bei der Dokumentation des Quellcodes des Ray Tracers.

In diesem Teilkapitel wurde in groben Zügen die Mathematik erläutert, die beim bidirektionalen Path Tracing von Bedeutung ist. Dabei wurden hauptsächlich die Ansätze und Methoden von Lafortune und Willems sowie von Henrik Wann Jensen herangezogen. Im nächsten Kapitel wird zu Beginn der Ray Tracer dokumentiert, den ich in dieser Studienarbeit erweitert habe und es werden meine eigenen Ansätze und Ideen zur Realisierung der mathematischen Formeln und Methoden erklärt.

Kapitel 2, Teil 2
Spezifikation der Algorithmen für bidirektionales Path Tracing

Dieses Teilkapitel ist eng mit dem ersten Teil des zweiten Kapitels verbunden, weil es jetzt um die Realisierung der mathematischen Formeln und Ansätze aus dem ersten Teil von Kapitel 2 geht. Zunächst werden die Eigenschaften und Klassen des Ray Tracers beschrieben, der die Grundlage für den bidirektionalen Path Tracer ist und in dieser Studienarbeit erweitert werden soll. Ich beschränke mich bei dieser Dokumentation auf die Klassen des Ray Tracers, die wichtig sind im Zusammenhang mit bidirektionalem Path Tracing. Die Dokumentation wird immer wieder durch Auszüge aus dem C++-Quellcode des Ray Tracers ergänzt. Anschließend werden die neu hinzugefügten Klassen bzw. die Ergänzungen in den vorhandenen Klassen beschrieben. Abschließend wird dann der eigentliche bidirektionale Path Tracing-Algorithmus mit Methoden-Aufrufen der Klassen dokumentiert.

2.2.1 Dokumentation wichtiger Klassen des Ray Tracers

2.2.1.1 Eigenschaften des Ray Tracers

Der Ray Tracer zeichnet sich durch die folgenden Eigenschaften aus:
Er wurde in C++ implementiert, ist plattformunabhängig und in der Lage, parallel auf verschiedenen Rechnern zu arbeiten. Je nachdem, wie er kompiliert wird, kann der Ray Tracer auf nahezu jedem Linux/Unix-Cluster gestartet werden durch Verwendung des Message Passing Interface (MPI). Der Ray Tracer parallelisiert das Ray Tracing auf folgende Weise: Das Bild wird von einem Manager-Prozess in Teilbilder zerlegt. Ein Worker Process fordert ein Teilbild an und bearbeitet es. Am Schluss werden alle resultierenden Teile vom Manager wieder eingesammelt und zusammengefügt.
Der Ray Tracer realisiert verschiedene lokale Beleuchtungsmodelle wie die von Schlick, Blinn und Phong. Das Phong-Beleuchtungsmodell kombiniert ideal diffuse mit empirisch verteilter spekularer Reflexion. Blinn entwickelte ein Modell der spekularen Reflexion auf physikalischer Grundlage. Das Modell von Schlick ist nicht abgeleitet von einer physikalischen Theorie von Oberflächenreflexion. Stattdessen ist es eine Mischung aus Approximationen zu existierenden theoretischen Modellen und intuitivem Reflexionsverhalten.
In den Ray Tracer wurden verschiedene Beschleunigungsalgorithmen wie BSP-Tree, Octree, Uniform Grid und Bounding-Volume-Hierarchien integriert. Der Ray Tracer kann zwischen verschiedenen Kamera-Modellen (Pinhole-Kamera und Thinlens-Kamera), verschiedenen Lichtquellenarten (Punktlichtquelle und Spotlicht) und unterschiedlichen Szenenobjekten (Kugel, Dreieck, Würfel, Quader) auswählen. Des Weiteren besteht die Möglichkeit, neben Szenenbeschreibungen, die im Quelltext stehen, extern über VRML-Dateien Szenenbeschreibungen einzulesen und zu rendern.

Zusätzlich kann mit dem vorhandenen Ray Tracer eine Szene mit Path Tracing gerendert werden. Daher bietet es sich an, den Ray Tracer jetzt so zu erweitern, dass eine Szene mit bidirektionalem Path Tracing gerendert werden kann. Darüber hinaus ist der Ray Tracer in der Lage, eine Szene mit Supersampling abzutasten. Die Generierung zusätzlicher primärer Augenstrahlen pro Pixel kann wahlweise durch Radial- oder Poisson-Verteilung erzielt werden.

2.2.1.2 Die wichtigsten Elemente des Ray Tracers

In diesem Abschnitt werden die Klassen hervorgehoben, die grundlegend für den Ray Tracer sind. Der Ray Tracer besteht insgesamt aus 51 verschiedenen Klassen, wobei außer der Datei main.cpp jede Klasse durch eine Header-Datei und die zugehörige Cpp-Datei beschrieben wird. Die Klassen, die sich auf Beschleunigungsmethoden, VRML-Parsing, Texturdarstellung oder die eigentliche Bilddarstellung beziehen, werden hier nicht beschrieben, weil das den Rahmen der Studienarbeit sprengen würde und diese Klassen nicht unmittelbar zum Thema dieser Studienarbeit passen.
Wichtig sind im Zusammenhang mit der Studienarbeit folgende Klassen:

- mathematische Elemente, die zur Darstellung und Berechnung beitragen wie Punkte (Klasse „Point“) und Vektoren (Klasse „Vector3D“)
- die geometrischen Primitive, die dargestellt werden sollen wie Quader (Klasse „Box“), Kugel („Sphere“) und Dreieck („Triangle“); sie werden zusammengefasst zu einer Oberklasse „GeomObject“
- die Klassen, die durch die mathematischen Elemente beschrieben werden wie Strahlen (Klasse „Ray“) und die Klasse „LocalGeometry“, die wichtige Informationen zu einem Schnittpunkt speichert;
- die Klassen, die die Lichtquellen und die Kameras beschreiben (Klasse „Light“ mit der Unterklasse „Spotlight“ und die Klasse „Camera“ mit den Unterklassen „PinholeCamera“ und „ThinlensCamera“)
- die Materialklassen „Material“ und die zugehörigen Unterklassen „HallMaterial“ und „PathTracMaterial“ (wobei eine weitere Unterklasse „PathandHallMaterial“ inhaltlich gesehen die Synthese aus den beiden Unterklassen ist)
- die Klasse „Loader“, in der die Szenenbeschreibungen stehen und eine VRML-Datei importiert werden kann;
- die Klasse „Raytracer“, die natürlich zu den wichtigsten zählt und
- die Klasse main, die die zentrale Rolle spielt und in der der Algorithmus gestartet wird;

In Anlehnung an diese Aufzählung werden die Klassen jetzt der Reihe nach dokumentiert. Die Dokumentation wird durch Auszüge aus dem Quelltext der Klassen ergänzt.

2.2.1.3 Dokumentationen einzelner Klassen

Point und Vector3D

Als erstes werden die Klassen Point und Vector3D beschrieben. Mit der Klasse Point werden nicht nur Eckpunkte von Objekten wie Dreiecken und Quadern, sondern auch der Mittelpunkt von Kugeln und Schnittpunkte von Strahlen mit Objekten beschrieben. Die Klasse Point ist wie folgt aufgebaut:

(Quellcode aus point.h)

```
class Point
{
        protected:
        Real m_data[3];

        public:
                ///Konstruktoren
                ///Standard-Konstruktor.
                Point();
                ///Konstruktor mit Wertübergabe.
```

```
        Point(Real x, Real y, Real z);

        void getXYZ(Real &x, Real &y,Real &z)const;
        ///Leseoperationen
        Real getX() const{ return m_data[0]; }
        Real getY() const{ return m_data[1]; }
        Real getZ() const{ return m_data[2]; }

        ///Schreiboperationen
        void setX(Real xWert){ m_data[0] = xWert; }
        void setY(Real yWert){ m_data[1] = yWert; }
        void setZ(Real zWert){ m_data[2] = zWert; }

        ///Operator-Überladung
        void operator= (const Point &p);
        void operator+= (const Vector3D &p);
        void operator-= (const Vector3D &p);
        Vector3D operator+ (const Point &p)const;
        Vector3D operator- (const Point &p)const;
        Point operator+ (const Vector3D &p)const;
        Point operator- (const Vector3D &p)const;
};
```

Ein Punkt im dreidimensionalen Raum wird durch drei Koordinaten beschrieben. Daher wird ein Array m_data[3] angelegt, der diese drei Koordinaten speichert und auf die über einen Index zugegriffen werden kann. Einfache get- und set-Methoden ermöglichen den Zugriff und das Verändern von einzelnen Punktkoordinaten. Die Methoden, die hier noch wichtig sind, sind die zur Operator-Überladung von Additions-, Subtraktions- und Zuweisungs-Operator. Des Weiteren wurden Methoden eingesetzt, mit denen ein Punkt durch einen Vektor verschoben werden kann und der verschobene Punkt zurückgegeben wird.

Die Klasse Vector3D ist ähnlich aufgebaut: Ein Vektor im dreidimensionalen Raum wird durch drei Komponenten beschrieben. Deshalb wird auch hier ein Array m_data[3] eingesetzt, um eine Datenstruktur für einen Vektor zu erhalten. Verschiedene get- und set-Methoden ermöglichen den Aufruf einzelner Vektor-Komponenten bzw. ihre Manipulation. Zusätzlich gibt es Methoden, die zwei Vektoren miteinander verknüpfen durch einfache Addition, Subtraktion oder einfache Zuweisung.

(Auszug aus der Datei vector3d.h)

```
(…)
        ///Operator-Überladung
        void operator= (const Vector3D &v);
        void operator+= (const Vector3D &v);
        void operator-= (const Vector3D &v);
        void operator*= (const Real scalar);
        Vector3D operator+ (const Vector3D &v)const;
        Vector3D operator- (const Vector3D &v)const;
        Vector3D operator* (const Real scalar)const;//Skalierungs-Operator

        ///Methoden
        Vector3D crossprod(const Vector3D &v);///Kreuzprodukt zweier Vektoren
        Real dotprod(const Vector3D &v);///Skalarprodukt zweier Vektoren
        void normalize();  //Vektor-Normalisierung
(…)
```

Die meisten Methoden in der Klasse Vector3D dienen der Operator-Überladung. Im Gegensatz zum Punkt kann ein Vektor durch eine skalare Größe in seiner Länge verändert werden. Darüber hinaus können zwei Vektoren durch Skalar- und Vektorprodukt („Kreuzprodukt“) verknüpft werden. Zur Realisierung dieser beiden Produkte wurden die Methoden „dotprod(…)“ und „crossprod(…)“ implementiert. Beiden Methoden wird als Parameter ein Vektor übergeben. Die Methode „dotprod(…)“ berechnet das Skalarprodukt aus zwei Vektoren und gibt eine skalare Größe zurück. Mit diesem Ergebnis lassen sich Aussagen über den Winkel zwischen den beiden verrechneten Vektoren machen.
Ein Beispiel für den Aufruf dieser Methode ist in der Datei „box.cpp“ in der Methode „getNormVector()“ zu finden (vgl. nächsten Textauszug), in der geprüft wird, auf welcher der sechs Flächen des Quaders sich der Schnittpunkt zwischen Augenstrahl und Quader befindet. Es wird hier das Skalarprodukt zwischen zwei Vektoren gebildet, von denen angenommen wird, dass sie senkrecht zueinander stehen. Das Skalarprodukt wäre in diesem Fall null (wobei das Ergebnis wegen Rundungsfehlern geringfügig von null verschieden sein kann und daher meist die Abfrage auf den Vergleich mit einem winzig kleinen Wert „epsilon“ geändert wird).

(aus der Datei box.cpp)

```
(…)
        // Abfrage für die Ebene_left;
        // dotprod() erwartet einen Vektor als Parameter!
        // daher erst aus den zwei Punkten einen machen;
        temp = intersect_point - point1;
        skalar = normal_left.dotprod(temp);

        if ((skalar < epsilon) || (skalar > -epsilon))
        {
                // dann liegt unter Berücksichtigung von Rundungsfehlern das
                // Skalarprodukt nahe bei null;
                // und der Schnittpunkt liegt auf Ebene_left;
                // daher wird der zugehörige Normalenvektor zurückgegeben;
                return normal_left;
        }
```

Weil die Variable „normal_left“ vom Typ „Vector3D“ ist, kann über sie die Methode dotprod(…) aufgerufen werden. Analog funktioniert der Aufruf der Methode „crossprod(…)“ zur Berechnung des Kreuzproduktes aus zwei linear unabhängigen Vektoren. Hier wird der Rückgabewert jedoch keine skalare Größe sein, sondern ein Vektor, der zu den beiden verrechneten Vektoren senkrecht steht.

Geometrische Objekte

Die Klassen Triangle, Box und Sphere dienen der geometrischen Beschreibung der Szenenobjekte beim Ray Tracing. Diese Klassen sind deshalb so wichtig, weil das Rendern einer Szene die Abbildung einer geometrischen Beschreibung auf ein Bild ist und daher ein Ray Tracer eine geometrische Beschreibung von Szenenobjekten braucht. Ein Dreieck wird durch seine drei Eckpunkte beschrieben. Eine Box wird durch zwei Eckpunkte beschrieben. Wichtig ist hier zu beachten, dass die zwei Eckpunkte nicht zur selben Fläche eines Quaders gehören, sondern das Quadervolumen aufspannen. Eine Kugel wird durch ihren Mittelpunkt und durch einen Radius beschrieben. In jeder dieser Geometrie-Klassen gibt es eine Methode, die den Schnittpunkt eines Strahles mit dem jeweiligen Objekt berechnet und einen Zeiger auf eine LocalGeometry-Instanz zurückgibt. Zusätzlich kann jede Klasse die Normale einer

Fläche aus den vorgegebenen Daten berechnen und als Vektor an die aufrufende Stelle übergeben. Es werden jetzt kurze Quellcode-Ausschnitte aus den Klassen für die geometrischen Objekte gezeigt. Als Oberklasse für alle Szenenobjekte dient die Klasse „GeomObject“, die ihrerseits eine Unterklasse von „SceneItem“ ist.

(aus der Datei box.h)

```
class Box : public GeomObject  // ..weil GeomObject Oberklasse ist
{
   public:

   /** der Würfel und auch ein Quader werden hier über 2 Punkte beschrieben,
   * die raumdiagonal gegenüber liegen;
   *
   * zuerst werden die Ebenengleichungen der sechs Quaderseiten aufgestellt
   * in Hessescher Normalenform;
   * dann wird jeweils der intersection point eingesetzt und die
   * Ebenengleichung überprüft;
   * der Normalenvektor der passenden Ebenengleichung wird zurückgegeben;
   */

(…)
   // Methode zur Normalenberechnung; gibt den Normalenvektor zurück;
   virtual Vector3D getNormVector(Point intersect_point);

   // Bestimmt den Schnittpunkt des Objekts mit einem geg. Ray
   virtual LocalGeometry*  intersect(Ray*  ray);

   // Methode, die prüft, ob ein Schnittpunkt innerhalb eines Objektes liegt;
   virtual bool inInterior(Point viewpoint);

   //Methode, die prüft, ob Schatten vorhanden ist
   virtual bool shadowIntersect(Ray *ray );

   /**
   * Methode zum Erzeugen einer BoundingBox für einen Quader;
   * BoundingBox wird zurückgegeben
   * \return BoundingBox
   */
   virtual BoundingBox* inquireBounds();

   //Methode, die LowerLeftFront zurückgibt
   Point getLLF();

   //Methode, die UpperRightBack zurückgibt
   Point getURB();

   //Methode, die LowerLeftFront setzt
   void setLLF(Point llf);

   //Methode, die UpperRightBack setzt
   void setURB(Point urb);

(…)
```

Wichtig sind in der Klasse „Box“ die Methoden getNormVector(…) und intersect(…), sowie für den Beschleuniger BVH die Methode inquireBounds(). Die anderen Methoden sind hauptsächlich get- und set-Methoden zum Festlegen der Eckpunkte URB (upper right back) und LLF (lower left front), die den Quader definieren. Die Klassen Triangle und Sphere sind ähnlich aufgebaut. Die Unterschiede von Triangle und Sphere im Vergleich zur Klasse Box

bestehen in der Arbeitsweise der Intersect-Methode, die den Schnittpunkt zwischen Strahl und Objekt berechnet und in der Definition dieser geometrischen Objekte. Das Dreieck wird durch drei Punkte (Klasse Point) beschrieben, die die Eckpunkte repräsentieren und die Kugel wird durch Mittelpunkt (Klasse Point) und Radius (skalare Größe) beschrieben. Durch set- und get-Methoden können auch hier die Punkte und skalaren Werte aufgerufen bzw. verändert werden.

Ray und LocalGeometry

Die Klasse „Ray“ spielt natürlich in Bezug auf Ray Tracing und Path Tracing eine zentrale Rolle, weil es bei diesen Beleuchtungsalgorithmen um das Verfolgen von Strahlen geht. Ein Strahl wird durch einen Ausgangspunkt und eine Richtung beschrieben. Der Ausgangspunkt ist vom Typ „Point“ und die Richtung wird durch einen Vektor festgelegt (Klasse „Vector3D“). Ausgangspunkt und Richtung können durch get-Methoden aufgerufen werden:

(aus der Datei ray.cpp)

```
const Point Ray::getOrigin() const {
        return origin;
}

const Vector3D Ray::getDirection() const {
        return direction;
}

void Ray::setDirection(Vector3D &dir) {
  direction = dir;
}
```

Zusätzlich kann in einem der Konstruktoren der Klasse Ray dem Strahl eine maximale Länge zugewiesen werden. Für das Traversieren der Strahlen vom Augpunkt aus oder von einem Schnittpunkt aus zum nächsten Schnittpunkt wird diese Länge nicht benötigt. Das Begrenzen eines Strahles durch eine festgelegte Länge macht erst bei Verwendung eines Strahles als Schattenfühler Sinn. Ein Schattenfühler wird von einem Schnittpunkt zur Lichtquelle bzw. speziell beim bidirektionalen Path Tracing von einem Stützpunkt des Augenstrahles zu einem Stützpunkt des Lichtstrahles aufgebaut. Wenn die Länge des Schattenfühlers begrenzt wird, werden nur Objekte bzw. Stützpunkte berücksichtigt, die in der näheren Umgebung des Schnittpunktes liegen, von dem aus der Schattenfühler startet. Wie weit diese Umgebung ausgedehnt wird, hängt dann von der festgelegten Länge des Schattenfühlers ab.

Die Klasse „LocalGeometry“ speichert Informationen zu einem Schnittpunkt. Eine Schnittpunkt-Methode intersect() aus einer der Klassen für die geometrischen Objekte gibt einen Zeiger auf ein LocalGeometry zurück. Über dieses LocalGeometry kann jederzeit die gewünschte Information zu einem Schnittpunkt abgerufen werden.
In LocalGeometry werden folgende Informationen gespeichert:

- die aktuelle Farbe am Schnittpunkt
- einen Zeiger auf das aktuelle SceneItem (SceneItem ist die Oberklasse zu allem, was in der Szene dargestellt wird, d.h. alle geometrischen Objekte, außer Lichtquellen und Kameras)
- die Entfernung zum vorigen Schnittpunkt
- den eigentlichen Schnittpunkt
- die Normale am aktuellen Schnittpunkt
- der Strahl, der den Schnittpunkt verursacht hat (= der ankommende Strahl)

Für diese Größen gibt es die zuständigen Kapselmethoden zum Setzen und Aufrufen der Attribute für diese Membervariablen (set- und get-Methoden).

(aus der Datei localgeometry.h):

```
///Leseoperationen
        Color* getColor(){ return mColor; }
        SceneItem* getSceneItem(){ return mItem; }
        Real& getDistance(){ return mDistance; }
        Point* getIPoint(){ return iPoint; }
        Vector3D* getNormal(){ return mNormal; }
        Ray* getIRay(){ return r; }

///Schreiboperationen
        void setColor(Color *c){  *mColor = *c; }
        void setSceneItem(SceneItem *s){ *mItem = *s; }
        void setDistance(Real d){ mDistance = d; }
        void setIPoint(Point *p){ iPoint = p; }
        void setNormal(Vector3D *v){ mNormal = v; }
```

Die Informationen aus LocalGeometry werden in erster Linie in der Materialklasse benötigt, wenn es darum geht, das Endergebnis für die Farbe bzw. den reflektierten Strahl zu berechnen. Im Material wird zum Beispiel für das Berechnen des reflektierten Strahles die Normale am Schnittpunkt benötigt sowie der Strahl, der den Schnittpunkt verursacht hat, um den Ausfallswinkel für den weiterführenden Strahl korrekt berechnen zu können.

Die Kamera und die Lichtquellen

Die Kamera-Klasse realisiert ein Kamera-Modell. Beim Ray Tracing und bei den Path Tracing-Algorithmen spielt sie eine zentrale Rolle, weil von der Kamera aus Primärstrahlen gesendet werden und durch das Abtastgitter in die Szene wandern. Diese Primärstrahlen werden als erstes mit den Objekten der Szene geschnitten. Es gibt als Oberklasse zu allen Kameratypen die Klasse „Camera“. Sie enthält unter anderem Methoden zum Festlegen des Abtastgitters (setImagePlane()) und zum Setzen der Öffnungswinkel der Kamera in vertikaler und horizontaler Richtung. Die Öffnungswinkel werden in der Klasse „Loader“ beim Aufruf der Konstruktoren der Unterklasse von Camera übergeben. Die Unterklassen sind „PinholeCamera“, die die Geometrie eines Lochkamera-Modells verwendet, und die Klasse „ThinslensCamera“, bei der die Strahlen von einer gedachten Linsenoberfläche aus starten und dadurch die primären Augenstrahlen verschiedene Startpunkte auf dieser Linsenoberfläche haben können. Mit der ThinlensCamera ist es möglich, Tiefenschärfe in einem Ergebnisbild zu simulieren.

(aus camera.cpp):

```
// Setzt neue ImagePlane.
//  \param ip Neue ImagePlane
void Camera::setImagePlane(ImagePlane* ip) { m_ip = ip; }

// Setzt Öfffnungswinkel in vertikaler Richtung.
// \param alpha Öffnungswinkel in vertikaler Richtung.
void Camera::setAlpha(Real alpha)            { m_alpha = alpha; }

// Setzt Öfffnungswinkel in horizontaler Richtung.
// \param beta Öffnungswinkel in horizontaler Richtung.
void Camera::setBeta(Real beta)    { m_beta = beta; }
```

Im Header der Klasse Camera werden folgende Größen festgelegt:
(aus der Datei camera.h)

```
  Point *m_position;   /**Position der Kamera              */
  Point* m_lookAt;    /**Blickpunkt der Kamera         */
 Vector3D* m_up;      /**Vektor senkrecht zur Blickrichtung  */
  ImagePlane* m_ip;

  Real m_alpha;              /**Öffnungswinkel vertikal */
  Real m_beta;               /**Öffnungswinkel horizontal */

  vector<Ray*> raylist;  /**Liste mit Primärstrahlen */
  vector<RayInfo> infolist;   /**Liste mit Information von Primärstrahlen*/
```

Die Liste der Primärstrahlen wird erst in der Unterklasse von Camera gefüllt. Dazu steht die Methode „getPrimaryRays(…)“ zur Verfügung, die eine Liste von Strahlen zurückgibt, den primären Augenstrahlen. Es werden zunächst zeilenweise auf dem Abtastgitter Abtastpunkte ermittelt. Bei einem Strahl pro Pixel wird der Mittelpunkt des Pixels als Abtastpunkt gewählt, bei Supersampling je nach Zufallsverteilung (Poisson- oder Radialverteilung) entsprechend mehr Abtastpunkte pro Pixel. Zu jedem Abtastpunkt wird ein Strahl erzeugt und in die Liste der Primärstrahlen („m_raylist“) eingefügt. Gleichzeitig werden Zusatzinformationen der Primärstrahlen wie Pixelzugehörigkeit im Ergebnisbild und Gewichtung des Strahles in eine separate Liste „m_infolist“ geschrieben. Die primären Lichtstrahlen werden zum ersten Mal in der Klasse „Raytracer“ verwendet, wenn sie an die rekursiv arbeitende Methode „raytrace(Ray* r, int m_recDepth)“ übergeben werden und auf ihrem Weg durch die Szene traversiert werden.

Eine Lichtquelle wird bei diesem Raytracer durch eine Intensität und eine Position im Raum beschrieben. Die Intensität ist hier keine skalare Größe, sondern eine Farbe (Typ „Color“). Von dieser Farbe wird immer ein Anteil eines Rot-, Grün oder Blau-Kanals verrechnet, wenn in Materialklassen das jeweilige lokale Beleuchtungsmodell implementiert wird. Die Klasse „light“ dient als Oberklasse zu der Klasse „spotlight“. Ein Spotlight ist eine Spezialisierung einer Punktlichtquelle, daher macht die Vererbungsstruktur zwischen Light und Spotlight Sinn. Ein Spotlight versendet Licht nur innerhalb eines Kegels, der durch einen Raumwinkel beschrieben wird und dessen Öffnung in eine bestimmte Richtung weist. Soll eine Punktlichtquelle realisiert werden, dann wird der Konstruktor der Klasse „light“ aufgerufen. Wenn ein Spotlight implementiert werden soll, wird der Konstruktor der Klasse Spotlight aufgerufen. Näheres zu den Lichtquellenarten wird später bei der Spezifikation der Flächenlichtquelle erläutert. In den Klassen Light und Spotlight werden lediglich Werte über Parameter des Konstruktors oder der jeweiligen set-Methoden den Membervariablen zugewiesen, die zur Beschreibung einer Lichtquelle durch Position und Intensität notwendig sind.

Die Material-Klassen

Die Materialklassen sind bei diesem Ray Tracer von großer Bedeutung. Oberklasse für alle Materialtypen ist die Klasse „Material“, in der die ambienten, diffusen und spiegelnden Anteile eines Materials gespeichert werden. In den Unterklassen HallMaterial und PathTracMaterial werden in der Methode getColor() die Farbanteile nach einer Reflexion eines Strahles an einer Oberfläche berechnet anhand des lokalen Beleuchtungsmodells. Es kann zwischen den Beleuchtungsmodellen Schlick, Blinn und Phong gewählt werden.

Neben der Berechnung der direkten Beleuchtung an einem Schnittpunkt werden in der Methode getColor() außerdem die nach einer Reflexion entstandenen Strahlen berechnet. Der rekursive Aufruf der raytrace-Methode für den reflektierten Strahl steht in dieser getColor()-Methode, weil es eine Frage des Materials ist, wie ein Strahl reflektiert wird. Der Unterschied zwischen den Klassen PathTracMaterial und HallMaterial ist der, dass in PathTracMaterial der Path Tracer realisiert wurde und hier mit russischem Roulette entschieden wird, ob nur der diffus reflektierte, nur der spekular reflektierte oder nur der refraktierte Strahl weiterverfolgt wird. Beim Path Tracing wird nur einer dieser Strahlen weiterverfolgt und beim rekursiven Aufruf der Trace-Methode mit übergeben. HallMaterial dagegen realisiert den Standard Ray Tracing-Algorithmus. In der Methode getColor() in HallMaterial werden daher nach der Reflexion sowohl der spekular reflektierte als auch der refraktierte Strahl rekursiv weiterverfolgt: für beide Teilstrahlen wird daher die raytrace-Methode aufgerufen. Weil der rekursive Aufruf in den Materialklassen steht, wird für den bidirektionalen Path Tracer eine neue Materialklasse geschrieben werden, die die Reflexion der Strahlen je nach lokalem Beleuchtungsmodell berechnet, die Ergebnisfarbe berechnet und rekursiv einen Augenstrahl bzw. einen Lichtstrahl analog zum Path Tracing weiterverfolgt.
Die wichtigsten Zeilen aus dem Quellcode von HallMaterial aus der Methode getColor() werden hier kurz erwähnt, damit später die neue Materialklasse für den bidirektionalen Path Tracer besser verständlich ist:

(aus der Datei HallMaterial.cpp, Methode getColor(); die Erklärungen an einigen Stellen sind fett unterlegt)

```
Color* HallMaterial::getColor(Ray* R, LocalGeometry* LG, int recursionDepth)
{
  Color* result = new Color();
  short light=raytracer->getLight();   // gibt die Zahl an, die das gewählte Beleuchtungsmodell bezeichnet;

// Liste der Lichter aus der Szene holen und Iterator initialisieren
  vector<Light*> lightList = *(scene->getLightList());
  vector<Light*>::iterator it;

  // ambientaler Term
  *result =  m_ambientColor->getColor(/*Texturkoordinaten*/) * scene->getAmbientColor();

  int cnt = 0;

// hier werden die Schnittpunktinformationen über den Parameter von LocalGeometry LG geholt:
  Point IPoint = *(LG->getIPoint());
  Vector3D viewVec = LG->getIRay()->getDirection() * (-1);
  Vector3D normal = *(LG->getNormal());
  normal.normalize();

(... Definition weiterer lokaler Variablen)

  // wenn normale vom betrachter wegzeigt: umdrehen
```

```
  if (viewVec.dotprod(normal) < 0.0) normal *= -1;

// Iteration über alle Lichtquellen
for (it = lightList.begin(); it != lightList.end(); it++)
{
        // benötigte Vektoren errechnen
        lightVec = lightList[cnt]->getPosition() - IPoint;
        shadowRay.setDirection(lightVec);

        lightVec.normalize();

        //falls aktuelles Licht Spotlight und der Schnittpunkt im Lichtkegel oder kein Spotlight, tue:

        if (lightList[cnt]->isSpotlight()&&inCone(lightList[cnt],lightVec) || !(lightList[cnt]->isSpotlight()))
        {
                // wenn der Schattenfühler kein Szenenobjekt schneidet, berechne Lichtbeitrag:
                if( scene -> shadowIntersect ( &shadowRay ) == false )
                {
                        Color intens = lightList[cnt]->getIntensity();
                        (...)
                /*
                        Je nach gewähltem Beleuchtungsmodell ( = Variable „light“) wird
                        jetzt das Endergebnis für „result“, die zu setzende Pixelfarbe
                        berechnet.
                */
                        (...)
                }
        }
    cnt++;   // Countervariable wird hochgezählt, damit die nächste Lichtquelle behandelt werden kann;
}       // ende der for-Schleife über alle Lichtquellen;

// Die folgende Bedingung prüft, ob die Rekursion abbricht:
// Trace-Rekursion
   // Abbruchbedingung
   if (recursionDepth <= 0)
     return result;

   viewVec *= (-1);

(…)
// Als nächstes wird bei einem Refraktions-Index größer 0 der Fresnel-Term berechnet.
(…)
// Danach werden die an der Materialoberfläche reflektierten Strahlen berechnet:
   // Reflexionsstrahl erzeugen und losschicken
   if ((m_reflectionColor->getColor().getR() +
        m_reflectionColor->getColor().getG() +
        m_reflectionColor->getColor().getB()) > 0.01)
   {
      Vector3D reflectVec = viewVec - (normal * ( viewVec.dotprod(normal))) * 2;
      Ray* reflectedRay = new Ray (IPoint, reflectVec);
      Color* reflecColor = raytracer->raytrace(reflectedRay, --recursionDepth);

      // Ergebnis der Reflexionsstrahlen muss noch verrechnet werden.
      Color koeff = Color(1,1,1) - m_reflectionColor->getColor();
      *result = (*result) * koeff + (*reflecColor) * m_reflectionColor->getColor();

       delete reflecColor;
   }
```

```
// Refraktionsstrahl erzeugen und losschicken
   // diese Rechnung funktioniert vorerst nur für Strahlen die von außen auf ein Object
   // treffen und dann im inneren gebrochen werden
   // wir aber bei allen Refraktionanforderungen aufgerufen und errechnet (Ergebnis wäre dann falsch)
   // und es wird angenommen das man von Vakuum/Luft in ein anderes Material übergeht
   if ((m_refractionColor->getColor().getR() +
        m_refractionColor->getColor().getG() +
        m_refractionColor->getColor().getB()) > 0.01)
   {
      Vector3D partOne = viewVec * (1/m_refractionIndex);
      Real partTwo = (viewVec.dotprod(normal) * (1/m_refractionIndex));
      Real partThree = 1-(((1/m_refractionIndex) * (1/m_refractionIndex)) *
                              (1-(viewVec.dotprod(normal)*(viewVec.dotprod(normal)))));

      Vector3D transmittVec = partOne -(normal * (partTwo + sqrt(partThree)));

      Ray* transmittRay = new Ray(IPoint,transmittVec);
      Color* refractColor = raytracer -> raytrace(transmittRay, --recursionDepth);

      // Ergebnis der Refraktionsstrahlen verrechnen
      //Color koeff = Color(1,1,1) - m_refractionColor->getColor();
      *result = *result + m_refractionColor->getColor() * (*refractColor);

      //delete transmittRay;
      delete refractColor;
   }
```

// Nach der Berechnung der reflektierten Strahlen wird die zu setzende Farbe berechnet:

```
   // die Farbe auf [0,1] beschränken
   Real red = result->getR();
   Real gre = result->getG();
   Real blu = result->getB();

   if (red < 0.0) result->setR(0.0);
   if (gre < 0.0) result->setG(0.0);
   if (blu < 0.0) result->setB(0.0);
   if (red > 1.0) result->setR(1.0);
   if (gre > 1.0) result->setG(1.0);
   if (blu > 1.0) result->setB(1.0);

   return result;
}
```

Der Loader

In der Klasse „Loader“ wird die Szenenbeschreibung in C++-Quellcode oder in Form einer importierten VRML-Datei aufgenommen. Auf die Bearbeitung einer importierten Datei möchte ich hier nicht eingehen. Stattdessen wird ein sehr einfacher Szenenaufbau aus dem Konstruktor der Klasse Loader gezeigt. Die Variable „sceneNr“ wurde vor dem Aufruf des Ray Tracers in der Kommadozeile mit einem Wert belegt. In diesem Beispiel beträgt ihr Wert 8.

(aus der Datei loader.cpp):

```
else if (sceneNr == 8)
{
        ///erste Punktlichtquelle festlegen:
        Point* light_point_2 = new Point(-10.0, 2.0, -10.0);
        Color* light_col_2 = new Color(1.0, 1.0, 1.0);
        Light* light_2 = new Light((*light_point_2), (*light_col_2));
        m_scene->addLight(light_2);


// (… weitere Lichtquellen können festgelegt werden)

        // jetzt das Material festlegen:
        // grau ohne Highlight:
        Material* hall_8 = new HallMaterial(new ColorMap(Color(0.4, 0.4, 0.4)),
                                            new ColorMap(Color(0.4, 0.4, 0.4)),
                                            new ColorMap(Color(0.0, 0.0, 0.0)), 0.0);
        m_scene->addMaterial(hall_8);


        // die Objekte der Szene: 2 Dreiecke als Grundfläche;
        // 2 Dreiecke, als Rechteck, das zwischen Lichtquelle und Grundfläche liegt
        // und für Schatten sorgt; auf der y-Höhe 1.0 bis 1.5

        /// Grundfläche (Boden) aus zwei Dreiecken
        ///erstes Dreieck
        Point* edge_1 = new Point(-10.0, 0.0, 10.0);
        Point* edge_3 = new Point(-10.0, 0.0,-10.0);
        Point* edge_2 = new Point( 10.0, 0.0, 10.0);
        Triangle* triangle_1 = new Triangle((*edge_1), (*edge_2), (*edge_3));
        triangle_1->setMatPointer(hall_8);
        m_itemList->push_back(triangle_1);
```

Analog zu „triangle_1“ werden weitere Dreiecke definiert und zu Rechtecken zusammengesetzt. In dieser Szene liegt eine quadratische Grundfläche der Größe 10 mal 10 in der xz-Ebene des Koordinatensystems. Darüber liegt ein kleineres Rechteck, das einen Schatten auf die Grundfläche werfen soll. Die Punktlichtquelle liegt über dem kleinen Rechteck.

Die Kamera wird durch folgende Zeilen festgelegt:
(weiter aus Loader.cpp):

```
(…)
// hier die Kamera-Definition:
        /// Kamera
        Real alpha = 23; Real beta = 23;

        PinholeCamera* cam = new PinholeCamera(new Point(0.0, 2.0, -6.0),
                                               new Vector3D(0.0, 1.0, 0.0),
                                               new Point(0.0,0.0,0.0), alpha, beta);
        m_scene->setCamera(cam);

        ///Einfuegen der ItemList in die Scene
        m_scene->addSceneItemList(m_itemList);

        // Kontrollanweisung:
}       // end of else if (sceneNr == 8)
```

Der Zeiger “m_scene” speichert dann nach Aufnahme aller Szenenobjekte mit Materialien, der Lichtquellen und der Kamera den kompletten Szenenaufbau. Der Zeiger m_scene wird in der Klasse Ray Tracer zum ersten mal aufgerufen, wenn im Konstruktor des Ray Tracers die Kamera aus der Szene geholt wird und einer Membervariablen aus Raytrace.cpp zugewiesen wird. Über diesen Zeiger kann dann auch auf die primären Augenstrahlen zugegriffen werden.

Die Klasse Raytracer

Diese Klasse darf natürlich in einem Raytracer nicht fehlen. Die Klasse Raytracer behandelt die Tracing-Aufrufe und liefert das berechnete (Teil-) Bild zurück. In dieser Klasse werden vorwiegend Variablen verwendet, die schon in anderen Klassen deklariert wurden, wie z.B. Zeiger auf die Szene, Variablen für die Rekursionstiefe oder Datenstrukturen, die die Strahlen bzw. die Liste der Lichtquellen zwischenspeichern.

(aus der Datei raytracer.h):

```
(…)

private:
  ///Szene
  /**Der Zeiger auf die zu berechnende Szene*/
  Scene *m_scene;
  ///Hoehe des insgesamt zu berechnden Bereichs
  /** Die Höhe des zu berechnenden Bildes*/
  int m_height;
  ///Breite des insgesamt zu berechnden Bereichs
  /** Die Breite des zu berechnenden Bildes */
  int m_width;
  ///Rekursionstiefe
  /**Legt fest wie oft ein Strahl maximal reflektiert/transmittiert wird. */
  int m_recDepth;
  ///SuperSampling
  /**Gibt an wieviel Strahlen pro Pixel verschossen werden. Wird an Kamera weitergegeben*/
  int m_sSampling;

(…)
```

Der Zeiger auf die Szene wird schon im Konstruktor der Klasse Loader definiert und wird in der Klasse Raytracer nur aufgerufen. Höhe und Breite des Bildes werden in der Klasse „Image“ festgelegt, die Rekursionstiefe ist eine globale Variable und der Wert von m_sSampling gibt die Anzahl der Strahlen pro Pixel beim Supersampling an. Die Variable, die die Anzahl der Strahlen pro Pixel angibt, wurde schon global definiert und in der main-Methode festgelegt. Die Klasse Raytracer muss die Werte dieser Variablen verarbeiten bzw. an Methoden anderer Klassen übergeben. Im Konstruktor der Klasse Raytracer werden folgende Werte der Parameter den Membervariablen der Raytracer-Klasse zugewiesen:

(aus der Datei raytracer.cpp):

```
// Konstruktor:
Raytracer::Raytracer(Scene *s,int w,int h, int recDepth, int sSampling, int lightModel)
{
  m_scene=s;
  m_width=w;
  m_height=h;
  m_recDepth=recDepth;
  m_camera=m_scene->getCamera();
```

```
 m_sSampling=sSampling;
 m_light=lightModel;
 m_ip = new ImagePlane(m_width, m_height); //->~raytracer
 m_camera->setImagePlane(m_ip);
 m_camera->adeptAngles(); //Bild entzerren

}
```

„m_scene“ ist ein Zeiger auf die Szene. Über diesen Zeiger kann auf jedes Item zugegriffen werden, dass im Loader bei der Szenenbeschreibung deklariert wurde. Allgemein ist alles, was in der Szene an Objekten verwendet wird, ein Item, außer den Lichtquellen und der Kamera.
Im Konstruktor wird über „m_camera=m_scene->getCamera();” der Membervariablen „m_camera“ das Kameramodell zugewiesen. Mit der Integer-Variablen „m_light“ wird später zwischen den lokalen Beleuchtungsmodellen von Phong, Schlick und Blinn in der Materialklasse unterschieden.

Wichtig in dieser Klasse ist vor allem die Methode „raytrace“, die eine Farbe zurückgibt und an die ein Strahl und die Rekursionstiefe als Parameter übergeben werden. Beide Parameter der Methode werden an die getColor()-Methode der Materialklasse weitergegeben. Hier wird die zu setzende Farbe verrechnet und weil der rekursive Aufruf in der Materialklasse passiert, wird hier auch die Rekursionstiefe dekrementiert. Die Rekursion des gesamten Algorithmus bricht ab, wenn die Rekursionstiefe null beträgt. Die Methode raytrace wird in der getColor()-Methode rekursiv aufgerufen. Die raytrace-Methode selbst sieht wie folgt aus:

(aus raytracer.cpp)

```
Color* Raytracer::raytrace(Ray *r, int recursionDepth)
{
 LocalGeometry *lg=m_scene->intersect(r);
 if (lg == NULL){
   delete r;
   return new Color(0,0,0);//->getImage()/getPartOfImage()
 }
 GeomObject* go= (GeomObject*) lg->getSceneItem();
 Color* col;
 Material* mat = go->getMatPointer();
 col = mat->getColor(r, lg, recursionDepth);
 delete (lg);
 delete r;

 return col;//->getImage()/getPartOfImage()
}
```

Zuerst wird ein Strahl r mit der Szene geschnitten. Die Intersect-Methode liefert einen Zeiger auf ein LocalGeometry-Objekt zurück, über den auf alle wichtigen Informationen des Schnittpunktes aufgerufen werden können. Wenn kein Schnittpunkt zwischen dem Strahl und einem Objekt existiert, dann wird die Hintergrundfarbe schwarz (=Color(0, 0, 0)) zurückgegeben. Anderenfalls existiert ein Schnittpunkt und das LocalGeometry wird zusammen mit dem Strahl und der Rekursionstiefe an die getColor()-Methode übergeben, die die berechnete Farbe zurückgibt, die dann auch von der Methode raytrace zurückgegeben wird.

Die Datei main.cpp

Es fehlt jetzt noch die Cpp-Datei, die den Raytracer aufruft, die Klassen zum Abspeichern des Ergebnisbildes sowie den Loader, der die Szene laden soll. Die Datei main.cpp ist die umfangreichste in diesem Ray Tracer. Hier wird daher nur das wichtigste aus main.cpp hervorgehoben. Sie beinhaltet Methoden zum Parsen der Kommandozeile und der Weiterverarbeitung der Werte der Kommadozeilenparameter. In ihr werden globale Variablen wie die Rekursionstiefe, die Größe des Ergebnisbildes, die Anzahl der Strahlen pro Pixel beim Supersampling, Variablen für die Beschleuniger und viele weitere Variablen mit Werten belegt. Weil der Ray Tracer die Berechnung eines Bildes parallel auf mehreren Rechnern ausführen kann, lässt sich die Klasse main.cpp grob in einen sequentiell arbeitenden Teil und einen parallel arbeitenden Teil gliedern.

Jetzt wird kurz die main-Methode aus der Datei main.cpp dokumentiert, die zum sequentiell arbeitenden Teil gehört und die den Algorithmus startet.

(aus main.cpp, die main-Methode):

```
///main Funktion
/**Viel Parsing und erstellen von Raytracer, Loader und Saver
*/
int main(int argc,const char **argv)
{
  timer=new Timer(SERIAL_TIMER);
  timer->start("Total");
  parseCommandline(argc,(char **)argv);
  Loader *loader;

// Hier wird unterschieden, ob eine VRML-Datei oder eine Szene aus der Datei
// Loader.cpp mit der Szenenbeschreibung geladen werden soll:

  if(strlen(infile)>0)
  { loader=new Loader(infile,accel,accelOpts); }  //->main()
  else
  { loader=new Loader(sceneNr,accel,accelOpts);}  //->main()

// Die folgenden 6 Zeilen sind die wichtigsten:
  scene=loader->getScene();
  raytracer=new Raytracer(scene,width,height, recDepth, sSampling, lightModel);//->main()
  PPMImage* img=(PPMImage *)(raytracer->getImage());
  if(saveSwitch){
    timer->start("Saving");
    img->saveImage(outfile);

(... weiterer Quellcode zum Bearbeiten einer ppm-Datei folgen)
     timer->stop();              // Zeitmessung anhalten
  }
  timer->stop();
(…)

// Löschen der Zeiger, die vorher verwendet wurden:
delete (loader);
delete (img);
delete (outfile);
  delete (infile);
  delete (timer);
  return EXIT_SUCCESS;
}
```

Nachdem nun die wichtigsten Klassen des Ray Tracers dokumentiert wurden, werden jetzt die einzelnen Ansätze und Spezifikationen erläutert, die zusammengefügt den bidirektionalen Path Tracer ausmachen. Die folgenden Erklärungen sind rein theoretisch. Wie diese Ansätze dann in C++ implementiert wurden, wird im dritten Kapitel dieser Arbeit erläutert.

2.2.2 Realisierung der theoretischen Ansätze aus Kapitel 2, Teil 1

Zunächst wird erklärt, wie primäre Lichtstrahlen für die einzelnen Lichtquellen generiert wurden. Ich habe eine Flächenlichtquelle im Ray Tracer ergänzt, die die Darstellung von Halbschatten in der Szene ermöglicht und mit der sich andere Kaustiken darstellen lassen, als mit einer Punktlichtquelle oder einem Spotlight. Es kann wahlweise eine runde oder rechteckige bzw. quadratische Flächenlichtquelle beim Rendern verwendet werden. Die Flächenlichtquelle ließ sich dann gut in den bidirekionalen Path Tracer integrieren.

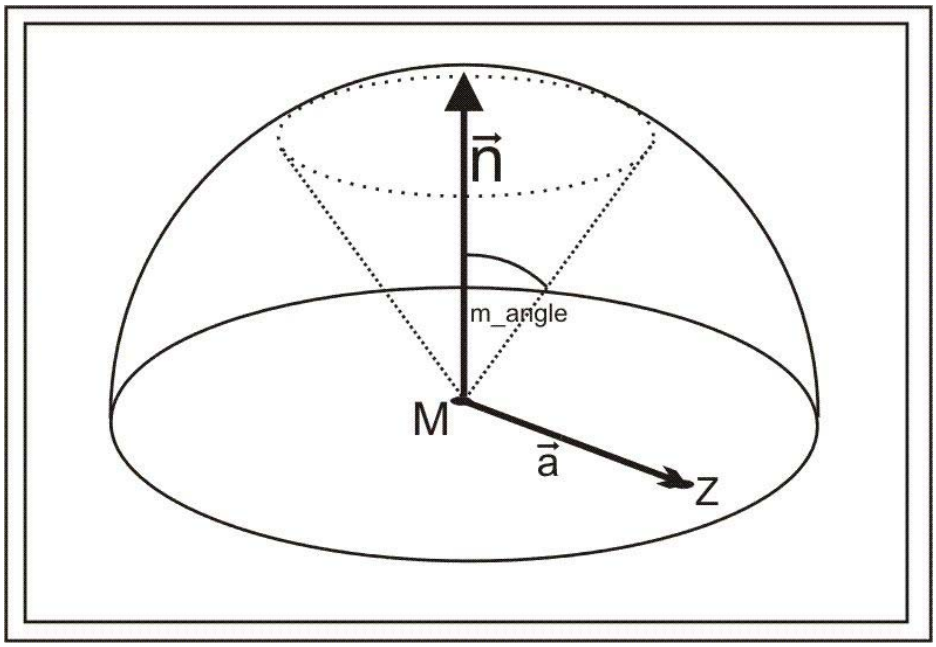
2.2.2.1 Generierungen der primären Lichtstrahlen

2.2.2.1.1 Primäre Lichtstrahlen für die Punktlichtquelle

Die primären Lichtstrahlen starten bei einer Punktlichtquelle alle im Ausgangspunkt der Lichtquelle. Wenn kein Objekt verhindert, dass Lichtstrahlen von der Lichtquelle direkt ins Auge des Betrachters gelangen, dann ist eine Punktlichtquelle von jeder beliebigen Position im Raum aus zu sehen. Daher kann ein primärer Lichtstrahl von der Punktlichtquelle aus in jede beliebige Richtung versendet werden. Der Startpunkt ist also für alle Primärstrahlen gleich, nur die Richtungsvektoren unterscheiden sich. Der Richtungsvektor im dreidimensionalen Raum besteht aus drei Komponenten. Daher werden drei Werte zufällig bestimmt und als Vektorelemente verwendet. Weil die Richtung der Strahlen beliebig ist, liegen diese Zufallswerte im Bereich [-1...1], so dass auch Vektoren in negative Achsenrichtung des dreidimensionalen Koordinatensystems zeigen können.

2.2.2.1.2 Primäre Lichtstrahlen für das Spotlight

Nun wird die Generierung von Lichtstrahlen beim Spotlight beschrieben. Die Richtungen für die primären Lichtstrahlen werden hier direkt durch den Lichtkegel des Spotlights eingeschränkt. Der Kegel wird durch einen Winkel festgelegt, der am Ausgangspunkt zwischen dem Kegelmantel und der Hauptachse des Kegels aufgetragen wird. Dieser Winkel beschreibt die Größe des Lichtkegels, durch den die Lichtstrahlen das Spotlight verlassen sollen. Die Richtung bzw. die mittlere Achse dieses Kegels wird durch einen Vektor beschrieben. Das Spotlight ist also im Gegensatz zur Punktlichtquelle eine gerichtete Lichtquelle. Startpunkt aller Primärstrahlen ist der Ausgangspunkt des Spotlights. Es muss nur dafür gesorgt werden, dass die primären Lichtstrahlen innerhalb des Kegels in die Szene wandern.



(Abbildung 2.10)
Der Winkel m_angle legt die Größe des Lichtkegels fest, der das Spotlight mit definiert. Der Punkt Z wurde zufällig generiert.

Zunächst wird eine imaginäre runde Fläche konstruiert, deren Mittelpunkt der Ausgangspunkt des Spotlights ist und die Ausrichtung der Achse des Lichtkegels stimmt mit der Richtung der Normalen $\vec{n}$ dieser runden Fläche überein. Der Radius der Fläche entspricht der Länge 1. Das heißt, die Normale $\vec{n}$ wird vorher auf die Länge 1 normalisiert. Hier kann also im Grunde eine Hemisphäre mit Radius 1 um die runde Fläche aufgebaut werden, wobei die Normale auf den Zenit der Hemisphäre zeigt (vgl. Abbildung 2.10). Jetzt wird zufällig ein zu dieser Normalen senkrecht stehender Vektor $\vec{a}$ berechnet: zwei Komponenten von $\vec{a}$ werden zufällig bestimmt, die dritte muss die Bedingung „Skalarprodukt zwischen $\vec{a}$ und Normale muss null sein“ erfüllen. Der Vektor $\vec{a}$ liegt dann genau auf der Fläche, egal welche Zufallswerte die ersten zwei berechneten Komponenten von $\vec{a}$ hatten.

Beispiel:
Gegeben: zwei per Zufallsgenerator berechnete Komponenten a1 und a2
Gesucht: Komponente a3

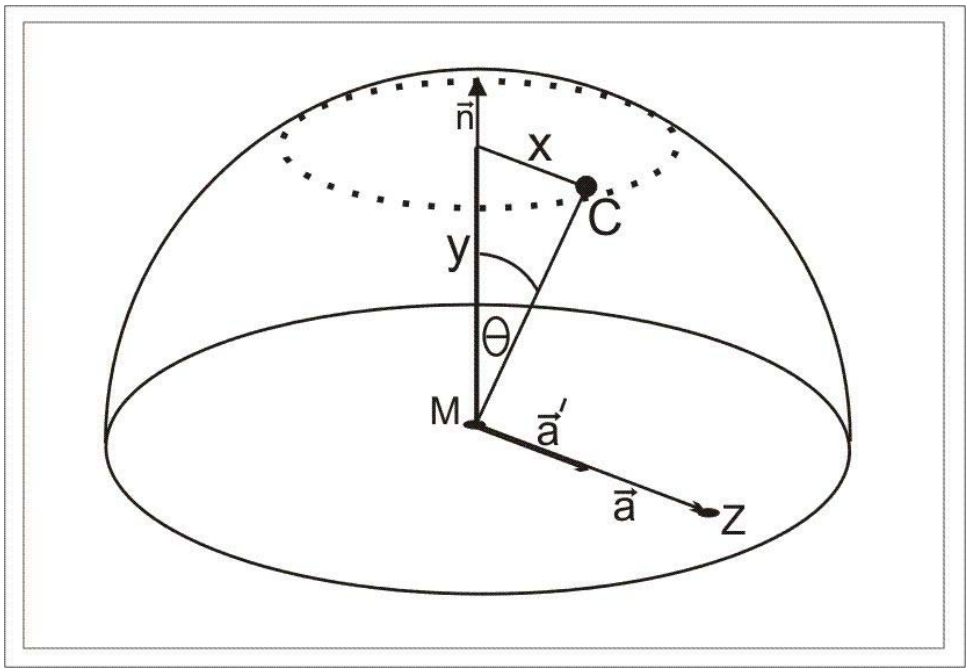
Ansatz: Bedingung für die Vektoren $\vec{a}$ und $\vec{n}$: Skalarprodukt muss null sein, weil $\vec{n}$ zu $\vec{a}$ senkrecht stehen muss;

$$0 = a1 * n1 + a2 * n2 + a3 * n3;$$

a1 und a2 sind bekannt, daraus folgt für a3:

$$a3 = \frac{-(a1 * n1) - (a2 * n2)}{n3}$$

Diese Berechnung ist jedoch nur möglich, wenn n3 nicht null ist. Daher wird später bei der Implementierung abgefragt, welche der Komponenten von $\vec{n}$ ungleich null ist und nach der jeweiligen Komponente von $\vec{a}$ freigestellt, ohne versehentlich durch null zu teilen.
Der zufällig bestimmte Vektor $\vec{a}$ führt vom Ausgangspunkt des Spotlights zu einem Zufallspunkt Z auf der Kreisfläche.



(Abbildung 2.11)
Der Winkel theta wird zwischen Normale und dem Vektor g= MC aufgetragen. Die Strecke x skaliert den Vektor a auf a'. Die Strecke y skaliert die Normale.

Jetzt wird ein Winkel theta zufällig bestimmt, der zwischen der Normalen $\vec{n}$ und einer gedachten Linie g aufgetragen wird, die vom Mittelpunkt M zu einem Punkt C auf der Hemisphäre gezogen wird, der noch nicht bekannt ist (vgl. Abbildung 2.11). Der Winkel theta liegt im Bereich [0..angle]. Die obere Intervallgrenze (hier „angle" genannt), entspricht dem Winkel, der bei der Definition des Spotlights für die Größe des Lichtkegels festgelegt wurde.

Punkte auf einer Hemisphäre werden meist durch Polarkoordinaten beschrieben. Dazu werden zwei Winkel benötigt. Der Winkel theta wäre der eine. Für den anderen Winkel, der auf der Kreisfläche am Mittelpunkt M aufgetragen wird, braucht man eine Bezugsachse auf der Kreisfläche, gegen die der Winkel aufgetragen wird. Weil diese Bezugsachse aber fehlt, wurde der Zufallspunkt Z auf der Kreisfläche bestimmt.
Über den Winkel theta werden die Strecken x und y mit Sinus und Cosinus berechnet. Dadurch wird sozusagen der Punkt C auf die Normale projiziert. Die Hypothenuse des Dreiecks aus projiziertem Punkt, C und M hat in jedem Fall die Länge des Radius' der Hemisphäre. Die Länge des Vektors $\vec{a}$ = MZ wird mit x skaliert, die Länge der Normalen $\vec{n}$ wird von der Länge 1 auf y skaliert. Die neue Richtung des Lichtstrahles ergibt sich durch Vektoraddition dieser beiden skalierten Vektoren $\vec{a}$' und $\vec{n}$'. Durch die Skalierung wird sichergestellt, dass der Lichtstrahl die Lichtquelle durch den festgelegten Lichtkegel verlässt. Weil der Winkel theta zufällig bestimmt wurde, ist die Strahlenrichtung ebenfalls zufällig innerhalb des Lichtkegels bestimmt worden.

2.2.2.1.3 Primäre Lichtstrahlen für die Flächenlichtquelle

Jetzt wird beschrieben, wie die Primärstrahlen für eine Flächenlichtquelle generiert werden.
Eine Flächenlichtquelle hat eine geometrische Ausdehnung. Das kann eine Fläche beliebiger Form und Größe sein. Hier in dieser Studienarbeit werden nur runde, rechteckige und als Spezialfall der rechteckigen quadratische Flächenlichtquellen implementiert. Theoretisch ist

es aber möglich, die Form der Fläche beliebig zu variieren: z.B. Polygone oder Flächen mit gekrümmten Kanten. Neben der Form und der Größe für die Fläche wird eine Normale festgelegt, die die Ausrichtung der Fläche festlegt und bestimmt, in welchen Halbraum die Flächenlichtquelle die Strahlen aussendet.
Weil die Flächenlichtquelle eine geometrische Ausdehnung hat, haben die primären Lichtstrahlen verschiedene Startpunkte auf der Fläche. Wenn zwischen einer runden und einer eckigen Lichtquelle unterschieden werden soll, dann lässt sich der Unterschied am besten anhand der Generierung von Startpunkten auf der Fläche verdeutlichen. Das ist also jetzt der erste Schritt: Bestimmung von Zufallspunkten auf der Flächenlichtquelle. Danach werden erst die Richtungsvektoren für die Lichtstrahlen berechnet.

Startpunkte auf einer runden Flächenlichtquelle

Zunächst möchte ich die Bestimmung von Zufallspunkten auf einer runden Fläche erklären.
Eine runde Flächenlichtquelle wird durch einen Mittelpunkt, einen Radius und eine Normale beschrieben, die im Mittelpunkt ansetzt und in den Halbraum zeigt, in den die Lichtstrahlen versendet werden sollen.
Zuerst wird zufällig ein zur Normalen der Lichtquelle senkrecht stehender Vektor $\vec{a}$ berechnet. Zwei der drei Vektorkomponenten von $\vec{a}$ werden mit einem Zufallsgenerator bestimmt. Wenn der Vektor $\vec{a}$ senkrecht zur Normalen stehen soll, muss das Skalarprodukt aus Lichtquellennormale und Vektor $\vec{a}$ null sein. Weil die dritte Komponente des Vektors $\vec{a}$ noch fehlt, wird sie nach der Bedingung „Skalarprodukt muss null sein“ bestimmt. Dieser Ansatz ist ähnlich zur Bestimmung des Punktes Z beim Spotlight.

Nachdem der Vektor $\vec{a}$ berechnet wurde, wird er normalisiert: er hat jetzt die Länge 1. Nun wird seine Länge per Zufall auf einen Wert zwischen 0 und dem Radius der runden Fläche gebracht. Weil der Vektor $\vec{a}$ in der Ebene der Fläche der Lichtquelle liegt und seine Länge jetzt entweder kleiner oder gleich dem Radius des Kreises ist, führt er vom Mittelpunkt aus zum eigentlichen Zufallspunkt auf der Kreisfläche (vgl. Abbildung 2.12). Die einzelnen Zufallspunkte auf der Kreisfläche werden zwischengespeichert und bei der Erstellung der Primärstrahlen verwendet.

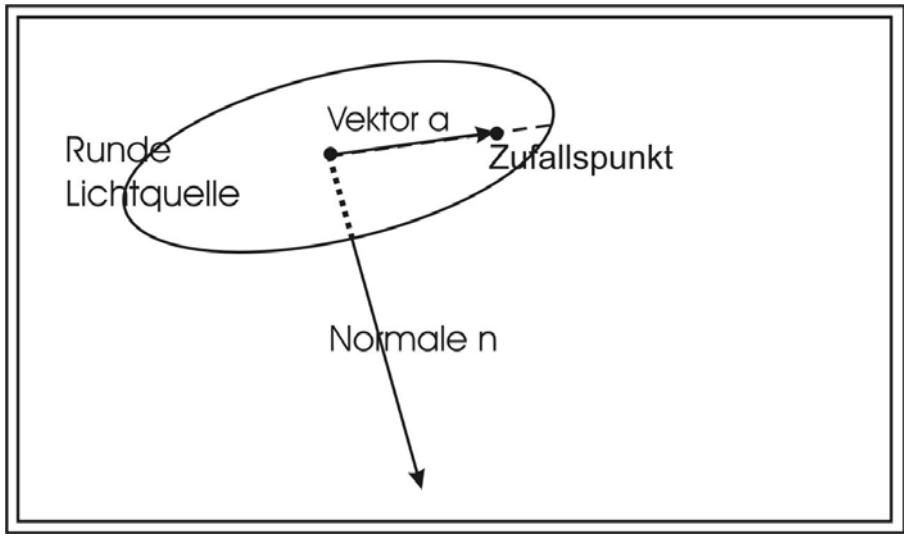


Abbildung 2.12: runde Flächenlichtquelle in der Vergrößerung

Startpunkte auf einer eckigen Flächenlichtquelle

Neben einer runden Flächenlichtquelle kann auch eine rechteckige Flächenlichtquelle spezifiziert werden. Sie ist ebenfalls durch einen Mittelpunkt und eine Normale definiert. Der Mittelpunkt auf der Fläche ist der Schnittpunkt der beiden Flächendiagonalen und die Normale zeigt in den Halbraum, der beleuchtet werden soll. Die Größe der Fläche wird durch eine Breite und eine Länge festgelegt. Wenn die Länge gleich der Breite ist, dann entsteht logischerweise eine quadratische Flächenlichtquelle.
Im Gegensatz zu einer runden Fläche ist bei einer viereckigen Fläche das Festlegen eines Zusatzpunktes von Bedeutung. Er legt die Ausrichtung des Rechtecks fest. Der Zusatzpunkt wurde im Voraus so gewählt, dass er auf der Mitte der kürzeren Seite des Rechtecks liegt (vgl. Abbildung 2.13). Bei einem Quadrat ist es natürlich egal, auf welcher Seite der Zusatzpunkt liegt. Wird jetzt ein Vektor $\vec{t}$ vom Mittelpunkt M der Lichtquelle zu diesem Zusatzpunkt gezogen, dann steht dieser Vektor senkrecht zur Normalen der Lichtquelle. Anschließend wird mit Hilfe des Vektorproduktes (Kreuzprodukt) aus Normale und $\vec{t}$ ein Vektor $\vec{s}$ berechnet, der sowohl zur Normalen, als auch zum Vektor $\vec{t}$ senkrecht steht. Die Vektoren $\vec{t}$ und $\vec{s}$ liegen beide auf der Fläche der Lichtquelle und werden zunächst normalisiert. Dann werden ihre Längen zufällig auf einen Wert zwischen 0 und der Hälfte der Seitenlänge der Seite gebracht, zu der sie parallel sind. In der folgenden Skizze entspricht die Länge des Vektors $\vec{t}$ der Hälfte der Länge y des Rechtecks. Die Länge des Vektors $\vec{s}$ wurde zufällig auf einen Wert zwischen 0 und der Hälfte der Breite x gebracht.

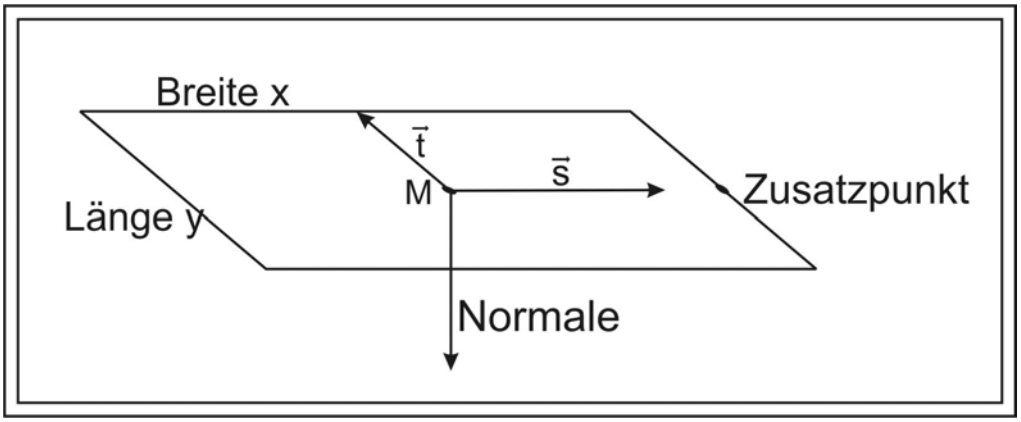
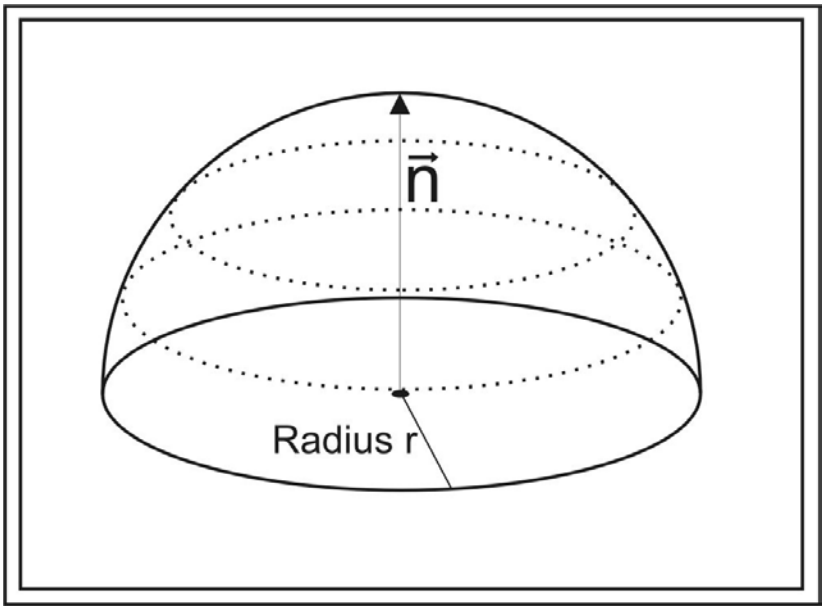


Abbildung 2.13: rechteckige Flächenlichtquelle in der Vergrößerung

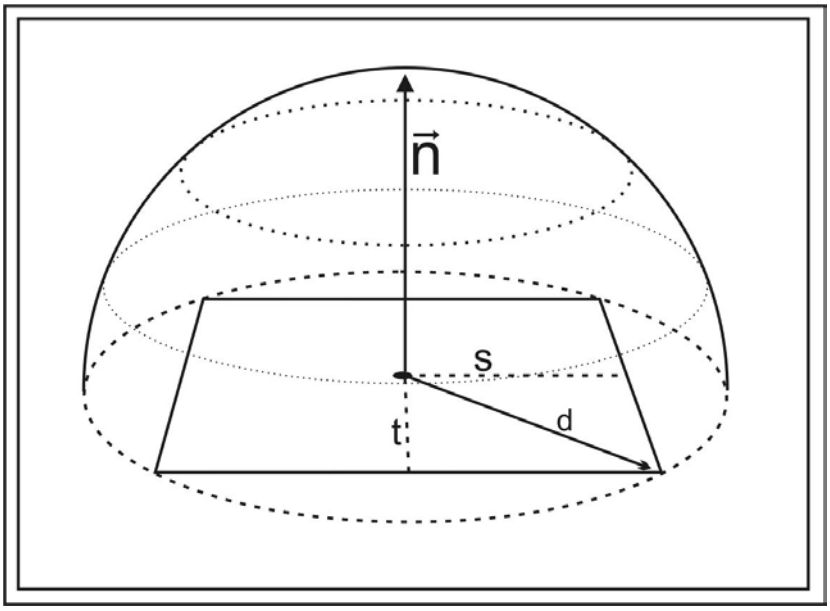
Durch Vektoraddition von $\vec{t}$ und $\vec{s}$ erhält man vom Mittelpunkt M der Lichtquelle aus einen neuen Punkt, der wegen der zufällig variierten Längen von $\vec{t}$ und $\vec{s}$ selbst zufällig generiert wurde und auf der rechteckigen Fläche liegt. Um eine möglichst gleichmäßige Verteilung der Punkte auf der Fläche zu erhalten, werden die Längen der Vektoren mit Faktoren aus dem Bereich [-1...1] variiert.
Der Vorgang dieses Punktegenerierens wird wiederholt, bis die gewünschte Punktanzahl erreicht ist. Die Punkte werden zwischengespeichert und bei der Erstellung der Primärstrahlen aufgerufen.

Richtungen für die primären Lichtstrahlen bei der Flächenlichtquelle

Nachdem die Startpunkte auf der Fläche generiert wurden, werden jetzt die Richtungen für die Primärstrahlen zufällig bestimmt. Dazu wird um die Fläche der Lichtquelle eine Hemisphäre aufgebaut, wobei die Normale der Flächenlichtquelle genau auf den Zenit dieser Hemisphäre zeigt.



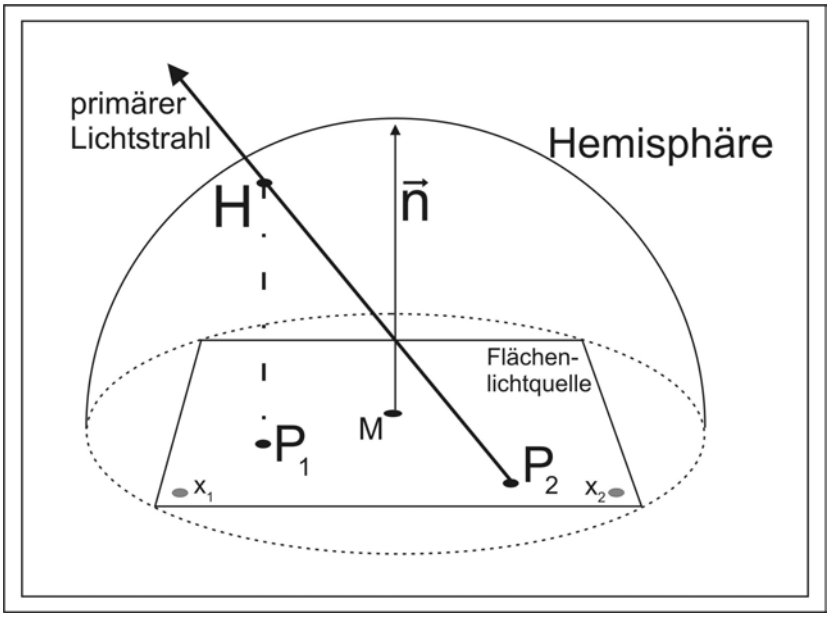
(Abbildung 2.14)
Die Hemisphäre wird um die runde Lichtquelle herum aufgebaut.
Der Radius der Hemisphäre entspricht dem Radius r des Kreises.



(Abbildung 2.15)
Die Hemisphäre wird hier um die rechteckige Lichtquelle herum aufgebaut.
Der Radius der Hemisphäre entspricht der Länge d, die sich
nach Pythagoras aus den Längen s und t berechnen lässt.

In der folgenden Erklärung wird eine eckige Fläche vorausgesetzt, die Überlegungen und Ansätze zur Implementierung gelten aber auch für die runde Flächenlichtquelle.

Auf der Hemisphäre werden jetzt Zufallspunkte bestimmt. Zunächst wird ein Zufallspunkt P_1 auf der Flächenlichtquelle ausgewählt. Er wurde bereits bei der Bestimmung von Zufallspunkten auf der Fläche generiert. Er wird parallel zur Normalen der Flächenlichtquelle auf die Hemisphäre projiziert (vgl. Abbildung 2.16). Dies sei der Punkt H. Er wird folgendermaßen berechnet:
Gegeben sind der Punkt P_1 und Mittelpunkt M. Ebenso ist die Normale bekannt, deren Länge auf den Radius der Hemisphäre skaliert wird. Des Weiteren lässt sich die Länge des Vektors MP_1 berechnen. Die Länge der Strecke MH entspricht dem Radius der Hemisphäre, auch wenn der Punkt H selbst noch nicht bekannt ist. Das Dreieck MP_1H ist ein rechtwinkliges Dreieck. Daher kann über den Satz des Pythagoras die Länge der Strecke P_1H berechnet werden über MH und MP_1. Im nächsten Schritt wird die Normale, die stets länger oder genau so lang wie P_1H ist, mit der Länge der Strecke P_1H skaliert. Mit diesem verkürzten Normalenvektor gelangt man von P_1 aus zum Punkt H auf der Hemisphäre.



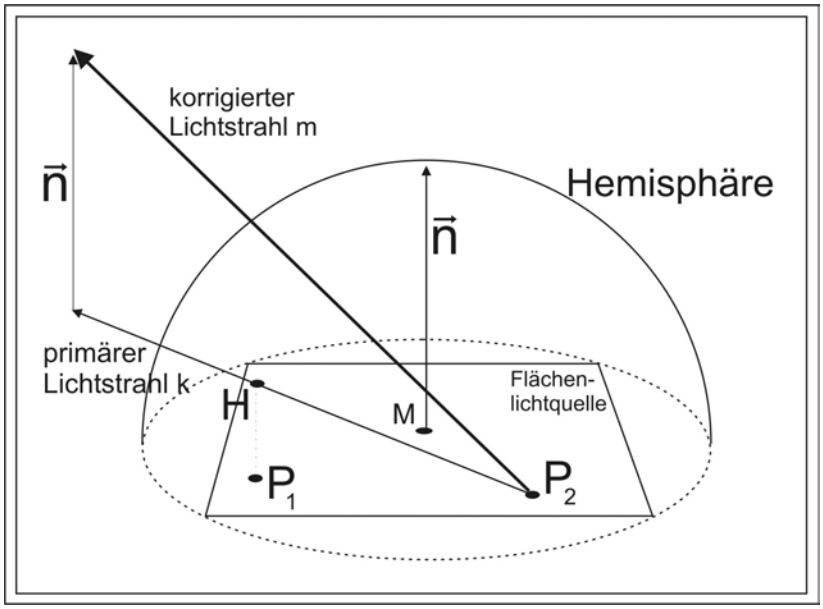
(Abbildung 2.16)
Zufallspunkte P_1 und P_2 werden auf der Flächenlichtquelle generiert. P_1 wird auf die Hemisphäre projiziert. Startpunkt für primären Lichtstrahl ist P_2, Richtungsvektor ist P_2H.

Wenn jetzt ein anderer Zufallpunkt P_2 auf der Flächenlichtquelle ausgewählt wird, dann ist der Richtungsvektor für den primären Lichtstrahl durch P_2H gegeben. Wenn der Punkt P_1 als Startpunkt des Lichtstrahles gewählt würde und durch H geht, dann hätte der Lichtstrahl die gleiche Richtung wie die Normale und alle Primärstrahlen würden parallel zur Normalen die Lichtquelle verlassen. Um verschiedene primäre Lichtstrahlen zu erhalten, werden immer wieder neue Punkte für P_1 genommen sowie ein neuer, zweiter Punkt P2 ausgesucht und dieser als Startpunkt des Lichtstrahles verwendet.

Diese Art der Strahlengenerierung hat einen „Schönheitsfehler“: Angenommen, der erste Zufallspunkt P_1 liegt sehr nahe am Rand der Hemisphäre (wie der Punkt x1 in Abbildung 2.16). Dann wird der projizierte Punkt von x1 auf die Hemisphäre nicht weit von x1 selbst entfernt liegen, sondern sehr nah oberhalb von x1. Wenn jetzt der zweite Zufallspunkt sehr weit weg liegt von x1 (wie zum Beispiel der Punkt x2), dann entsteht in diesem Fall ein Lichtstrahl, der fast parallel zur Flächenlichtquelle verläuft. Das ist vielleicht ein Ereignis, dass selten eintritt, weil ein Zufallsgenerator in der Regel sehr verschiedene Werte liefert und daher die Punkte gleichmäßig über die Flächenlichtquelle verteilt liegen. Aber wenn man

diesen „worst case“ weiterdenkt und *alle* primären Lichtstrahlen die Lichtquelle nahezu senkrecht zur Normalen $\vec{n}$ verlassen würden, dann könnte ein kleines Objekt direkt vor der Lichtquelle platziert werden und es würde nicht beleuchtet werden.

Da dies nicht der Realität entspricht und um diesen „worst case“ auszuschließen, werden die möglichen Richtungen der primären Lichtstrahlen etwas eingeschränkt:
Zunächst wird der primäre Lichtstrahl wie oben beschrieben berechnet: Zufallspunkte P_1 und P_2 wählen, den einen Punkt auf die Hemisphäre projizieren und den Richtungsvektor P_2H ziehen. Jetzt wird der Winkel zwischen diesem neuen Strahl und der Normalen berechnet. Ist dieser Winkel größer als ein Grenzwert (zum Beispiel 45°), dann wird die Richtung des Lichtstrahles etwas „in Richtung der Normalen“ korrigiert. Das heißt, es wird auf den Richtungsvektor des Lichtstrahles, der einen zu großen Winkel zur Normalen hat, die Normale vektoriell hinzuaddiert (vgl. Abbildung 2.17). Am meisten wird ein Lichtstrahl natürlich durch diese Vektoraddition korrigiert, wenn die Länge der Normalen genau eins ist oder sogar größer als eins. Alternativ dazu könnte die Länge der Normalen je nach Größe des Winkels proportional verkürzt oder verlängert und dann erst hinzuaddiert werden. Bei der Implementierung wurde die Normale auf die Länge eins normiert und erst dann zum Richtungsvektor des primären Lichtstrahles addiert, wenn der Winkel zwischen Lichtstrahl und Normale größer als 45° war.



(Abbildung 2.17)
Der Winkel zwischen Lichtstrahl k und Normale n ist zu groß.
Daher wird durch Vektoraddition der Strahl k korrigiert.
Ergebnis der Vektoraddition ist der Lichtstrahl m.

Mit einfachen Worten wird also ein Punkt, der zu „weit unten“ liegt auf der Hemisphäre, etwas in Richtung Zenit der Hemisphäre verschoben, damit der Lichtstrahl nicht zu parallel zur Fläche der Lichtquelle in die Szene wandert.

2.2.2.2 Die Schattenfühler

Nachdem die primären Lichtstrahlen generiert wurden, können sie jetzt verwendet werden. Das Licht einer Lichtquelle wird beim Ray Tracer durch eine Intensität beschrieben. Wenn also die Lichtstrahlen beim bidirektionalen Path Tracing ähnlich wie die Augenstrahlen beim Ray Tracing in die Szene gesendet und mit Objekten geschnitten werden, dann müssen sie diese Intensität weitertransportieren. Wenn es nicht ausschließlich ideale Spiegel in der Szene gibt, dann nimmt die Intensität des Lichtstrahles an jedem weiteren Schnittpunkt etwas ab.
Die an den Schnittpunkten gespeicherten Intensitätswerte tragen mit zum Ergebnis der Beleuchtung bei. Bei der Implementierung des bidirektionalen Path Tracings werden diese Intensitätswerte an den Stützstellen des Lichtstrahles zwischengespeichert und für jede Stützstelle des Augenstrahles eingerechnet. Das heißt, dass für jeden aktuellen Stützpunkt eines Augenstrahles in einer Schleife über alle zwischengespeicherten Intensitätswerte der Lichtstrahlen-Vertices iteriert wird und ihre Werte verrechnet werden zur Gesamtfarbe. Der Intensitätswert eines Lichtstützpunktes wird nur dann verrechnet für den Augenstrahl-Vertex, wenn die Vertices von Licht- und Augenstrahl sich „sehen“. Das heißt, es darf kein Objekt zwischen den beiden Stützstellen verhindern, dass Licht vom Lichtstrahl-Vertex zum Augenstrahl-Vertex gelangt.
Wenn zum Beispiel der Lichtstrahl eine Rekursionstiefe von 5 hat, dann entstehen zusammen mit der Lichtquelle selbst 6 Lichtvertices, die alle für jede Stützstelle eines Augenstrahles berücksichtigt werden. Wenn der Augenstrahl zum Beispiel ebenfalls die Rekursionstiefe 5 hat, dann werden insgesamt 25 verschiedene Schattenfühler aufgebaut. Weil für jeden dieser Schattenfühler ein Schnittpunkttest durchgeführt werden muss und ein Schnittpunkttest zu den aufwendigsten Operationen beim Ray Tracing und Path Tracing zählt, ist der Aufwand beim bidirektionalen Path Tracing sehr groß. Bidirektionales Path Tracing hat die Eigenschaft, dass mehr Schattenstrahlen als in klassischem Path Tracing verschossen werden.

Eric Lafortune und Yves Willems beschreiben in ihrer Arbeit „Reducing the Number of Shadow Rays in Bidirectional Path Tracing“ [Quelle [4] im Quellenverzeichnis] einen Weg, um die Zahl der Schattenfühler und damit die Zahl der Schnittpunkttest zu verringern. Diese Idee wurde bereits im ersten Teil des zweiten Kapitels erklärt.
Natürlich kann das Ignorieren der Masse von Schattenstrahlen die Varianz ansteigen lassen. Es wird Lichtinformation, die eigentlich eingerechnet werden sollte, ignoriert. Die zunehmende Varianz ist im Ergebnisbild durch erhöhtes Rauschen erkennbar. Sie kann verringert werden durch Wählen von mehr Augenstrahlen pro Pixel. Wenn weniger Schattenfühlertests durchgeführt werden müssen, können mehr Augenstrahlen pro Pixel versendet werden, ohne dass der Berechnungsaufwand gravierend ansteigt. Lafortune und Willems zeigen anhand von Beispielen, dass diese Vermutung zutrifft.
Die zweite Option der gerade beschriebenen Optimierung wird später implementiert werden.

Jetzt möchte ich abschließend noch auf die Formel zur Beleuchtungsberechnung eingehen, nach der die Intensitäten der Lichtstrahl-Vertices verrechnet werden zum Endresultat.

2.2.2.3 Beleuchtungsberechnungen nach Jensen

Die Formel zur Berechnung der Leuchtdichte stammt von Jensen [Quelle [9] im Quellenverzeichnis, Seite 44f.].
Die reflektierte Strahldichte bei x_i, die bezogen wird auf y_j, wird hier berechnet als:

$$L_{i,j}(x_i \to x_{i-1}) = f_r(y_j \to x_i \to x_{i-1}) * V(x_i, y_j) * \frac{\left|(y_j \to x_i) * \vec{n}_{x_i}\right|}{\left\|x_i - y_j\right\|^2} * I(y_j \to x_i)$$

(Gleichung 2.22 aus Kap 2, Teil 1)

In dieser Gleichung bezog sich x_{i-1} auf die Stützstelle des Augenstrahles, die vor x_i liegt. Der Vektor $\vec{n}_{x_i}$ ist die Normale im Punkt x_i. $I(y_j \to x_i)$ ist die Strahlungsintensität, die von einem Stützpunkt des Lichtstrahles y_j ausgeht in Richtung Stützpunkt des Augenstrahles x_i. Diese Strahlungsintensität $I(y_j \to x_i)$ für einen Vertex auf dem Lichtpfad wird berechnet als:

$$\mathrm{I}(\mathrm{y_j} \to x_i) = \Phi_i(y_j) * \left|(y_j \to x_i) * \vec{n}_{y_j}\right| * f_r(y_{j-1} \to y_j \to x_i)$$

(Gleichung 2.23 aus Kap 2, Teil 2)

f_r bezeichnet in den Gleichungen 2.22 und 2.23 eine BRDF. Die BRDF wird hier auf zwei Dimensionen beschränkt und als das Verhältnis von einfallender Beleuchtungsstärke E und ausfallender Strahldichte L verstanden. Die ersten zwei Parameter von f_r dienen zur Bestimmung der einfallenden Größe und mit dem zweiten und dritten Parameter wird die ausfallende Größe bestimmt. $\Phi_i(y_j)$ ist der Strahlungsfluss des einfallenden Photons bei y_j.

In den folgenden Abschnitten wird erklärt, wie sich die Beleuchtungsstärke E und die Leuchtdichte L berechnen lassen. Die Intensität $I(y_j \to x_i)$ aus Gleichung 2.23 wird beim bidirektionalen Path Tracer für jeden Stützpunkt des Lichtstrahles zwischengespeichert. Wenn dann die Schattenstrahlen zu den Stützstellen auf dem Augenstrahl geschickt werden, wird die Beziehung ($y_j \to x_i$) deutlich. Das Problem bei der Implementierung der Intensitätsberechnung nach Gleichung 2.23 ist, dass kein Strahlungsfluss $\Phi_i(y_j)$ gegeben ist. Der Strahlungsfluss ist laut Gleichung 2.3 aus dem ersten Teil des zweiten Kapitels der Quotient aus Strahlungsmenge Q und der Zeit t. Weil aber beim Ray Tracer und beim bidirektionalen Path Tracer keine Zeitmessung berücksichtigt wird und auch keine Strahlungsmenge Q gegeben ist, lässt sich der Strahlungsfluss nicht berechnen. Stattdessen wird ein anderer Ansatz verfolgt, um die Intensität an jedem Schnittpunkt zu berechnen.

Beim Ray Tracer werden die Lichtquellen durch ihre Intensität I charakterisiert. Daher muss sie für Gleichung 2.23 nicht immer neu berechnet werden. Beim bidirektionalen Path Tracer wird für ein Augen-Lichtstrahl-Vertex-Paar die Intensität verrechnet, die beim Lichtstrahl-Vertex zwischengespeichert wurde. Diese Intensität wird verwendet, um die Beleuchtungsstärke E zu berechnen, die am Augenstrahl-Vertex ankommt. Sie lässt sich nach folgender Formel ermitteln:

$$E = \frac{I * \cos\alpha}{r^2} \ ,$$

wobei I die Intensität ist, die zwischengespeichert wurde, der Winkel Alpha wird zwischen der Normale am Augenstrahlvertex und dem ankommenden Schattenfühler gemessen und r der Abstand ist zwischen dem Augenstrahlvertex und dem Lichtstrahlvertex. Alle diese Größen lassen sich berechnen bzw. sind schon vorhanden und müssen daher nur verrechnet werden für E.
Als nächstes wird die Leuchtdichte L berechnet. Weil hier die Beleuchtung am Augenstrahlvertex berechnet wird, wird L nach einem lokalen Beleuchtungsmodell berechnet. Wenn zum Beispiel das Phong-Modell realisiert wird, dann ergibt sich für die Leuchtdichte:

$$L_{Phong} = L_{amb} + \sum_{i=1}^{n} (L_{diff,i} + L_{spec,i}) ,$$

wobei

L_{amb} das ambiente Licht bezeichnet,
L_{diff} das ideal diffus reflektierte Licht und
L_{spec} gerichtet diffus reflektiertes Licht (beim Phong-Modell) bezeichnet.

Für jeden einzelnen Summanden L_i muss die Leuchtdichte L des einfallenden Lichtes, also des Lichtes, das beim bidirektionalen Path Tracing von einem Lichtstrahlvertex ausgeht, berücksichtigt werden.
Jetzt sind Beleuchtungsstärke E und L bekannt und der Wert für die BRDF in Gleichung 2.22 kann angenähert werden durch:

$$f_r(y_j \to x_i \to x_{i-1}) = \frac{dL_{out}(d\vec{\omega}_{out})}{dE_{in}(d\vec{\omega}_{in})} .$$

Damit ist der erste Faktor aus Gleichung 2.22 berechnet. Der Faktor $V(x_i, y_j)$ ist eins, wenn der Schattenfühlertest positiv war und kein Objekt zwischen den Punkten x_i und y_j liegt. Der Faktor ist null, wenn ein Objekt dazwischen liegt und den Lichtaustausch verhindert. Dieser Faktor wird für die Gleichung 2.22 nicht berechnet, sondern es wird vorher der Schattenfühler aufgebaut und der intersect-Methode als Strahl übergeben. Wenn festgestellt wird, dass ein Objekt den Lichtaustausch zwischen zwei Punkten verhindert, dann wird die Gleichung für $L_{i,j}$ erst gar nicht berechnet, sondern es wird das nächste Punkte-Paar aus Lichtstrahlvertex und Augenstrahlvertex untersucht.
Damit bleibt noch die Berechnung des Quotienten übrig, in dem die Normale am Stützpunkt x_i und die Punkte x_i und y_j zu verrechnen sind. Diese Größen sind bekannt und daher kann der Quotient berechnet werden.
Nachdem alle einzelnen Faktoren bestimmt wurden, werden sie nach Gleichung 2.22 miteinander multipliziert und damit ist $L_{i,j}$ bestimmt. Bei jeder Rekursionsstufe wird dieses $L_{i,j}$ berechnet und zu einem Gesamtwert L_{gesamt} aufsummiert.

2.2.2.4 Ablauf des bidirektionalen Path Tracing Algorithmus'

In diesem Abschnitt wird kurz erläutert, wie der bidirektionale Path Tracing Algorithmus abläuft:
Erst werden die primären Augenstrahlen von der Kamera aus Richtung Abtastgitter aufgebaut und nach ihrer Erstellung in einer Liste abgespeichert. Anschließend werden die primären Lichtstrahlen von allen Lichtquellen aufgesammelt und in einer anderen Liste abgespeichert. Weil es mehrere Lichtquellen in der Szene geben kann, läuft eine Schleife über die

Lichtquellen der Szene und sammelt immer wieder primäre Lichtstrahlen auf. Die Lichtstrahlen aller Lichtquellen werden in einer einzigen Liste abgelegt. Wie viele Lichtstrahlen pro Lichtquelle aufgesammelt werden, kann der Programmierer selbst festlegen durch Parameter.

Nun wird in einer Schleife über alle primären Augenstrahlen iteriert. Dazu wird immer ein Augenstrahl aus der Liste der primären Augenstrahlen genommen. Bevor der Augenstrahl jedoch in die Szene geschickt wird, wird pro Lichtquelle zufällig ein Lichtstrahl aus der Gesamtliste aller Lichtstrahlen ausgesucht und sein Weg durch die Szene nachvollzogen. Wenn die Lichtstrahlen der Lichtquellen auf Objekte treffen, dann wird der Schnittpunkt zwischen Lichtstrahl und Szenenobjekt in einer separaten Liste zwischengespeichert und es wird der reflektierte Lichtstrahl berechnet. An jedem Schnittpunkt des Lichtstrahles wird eine Lichtmenge zwischengespeichert. Die Verfolgung eines Strahles wird abgebrochen, wenn die maximale Rekursionstiefe des Lichtstrahles erreicht ist oder wenn ein primärer oder reflektierter Lichtstrahl kein Szenenobjekt trifft und ins Leere wandert.
Nach der Traversierung eines Lichtstrahles jeder Lichtquelle liegt eine Liste mit Schnittpunkten von Lichtstrahlen vor, wobei jede Lichtquelle genau einen Strahl versendet hat. Jetzt wird erst ein Augenstrahl durch das Abtastgitter in die Szene traversiert. Diesem Augenstrahl wird die aktuelle Liste aller Schnittpunkte der vorher ausgesuchten Lichtstrahlen mitgegeben.
Beispiel: Die Lichtstrahlen werden bis zu einer Rekursionstiefe von 4 traverisiert. In der Szene gibt es zwei Lichtquellen. Daher werden ein Lichtstrahl für die eine Lichtquelle ausgesucht (nach seiner Generierung) und ein Lichtstrahl für die andere Lichtquelle. Unter der Annahme, dass beide Lichtstrahlen bis zur maximalen Rekursionstiefe verfolgt werden, entstehen pro Lichtstrahl fünf Schnittpunkte, weil für die beiden Lichtquellen selbst auch ein „Schnittpunkt“ mit Lichtmenge berechnet wird. Das heißt, die Liste der Lichtstrahlenschnittpunkte enthält zehn Elemente. Diese Liste wird bei der Verfolgung eines Augenstrahles berücksichtigt.

Ebenso wie der Lichtstrahl wird der Augenstrahl durch die Szene verfolgt und mit Objekten geschnitten. Der Augenstrahl startet aber bei der Kamera. An den Schnittpunkten wird ein Reflexionsstrahl berechnet und es gibt eine maximale Rekursionstiefe, die von der des Lichtstrahles verschieden sein darf.
Bei jedem Schnittpunkt des Augenstrahles mit einem Objekt wird die Liste der Lichtstrahl-Schnittpunkte durchlaufen und berechnet, welche Lichtmenge von jedem dieser Stützstellen beim aktuellen Schnittpunkt des Augenstrahles ankommt. Aus der Summe aller Lichtbeiträge lassen sich die direkte und die indirekte Beleuchtung für den Schnittpunkt des Augenstrahles berechnen. Durch die rekursive Verfolgung des Augenstrahles wird die Lichtmenge pro Stützstelle aufsummiert und schließlich in einen Farbwert für den Pixel konvertiert, durch den der primäre Augenstrahl gesendet wurde.

Dieser Vorgang wird nacheinander für jeden primären Augenstrahl wiederholt: zuerst pro Lichtquelle einen Lichtstrahl verfolgen, die Stützstellen aufsammeln, einen Augenstrahl durch das Abtastgitter senden, die Stützstellen mitgeben und den Beleuchtungsaustausch zwischen den Stützstellen der Lichtstrahlen und des Augenstrahles berechnen. Nachdem die Liste der Schnittpunkte eines Lichtstrahles für einen Augenstrahl verwendet wurde, wird sie natürlich wieder geleert und mit Schnittpunktinformationen von anderen Lichtstrahlen neu aufgefüllt.

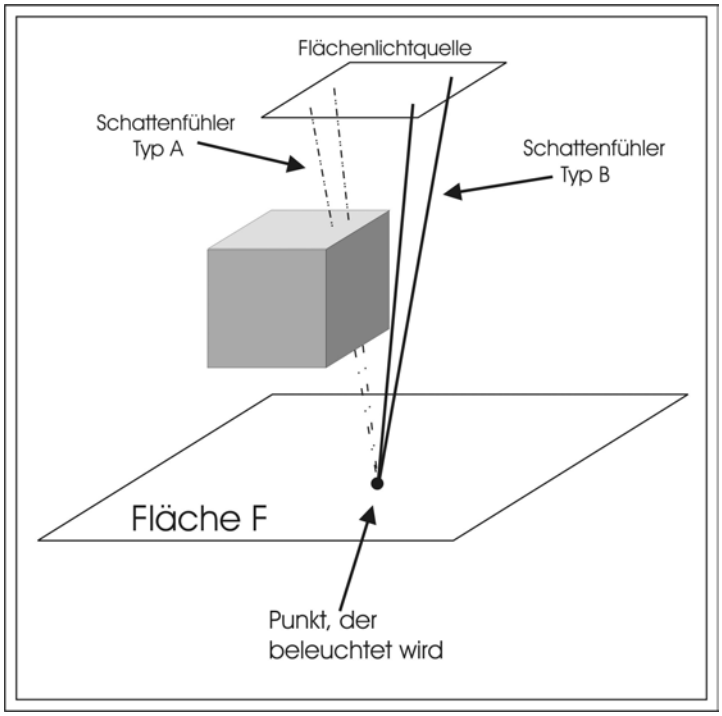
In diesem Teilkapitel wurde der vorhandene Ray Tracer dokumentiert. Anschließend wurden die Ansätze zur Generierung von Lichtstrahlen bei verschiedenen Lichtquellen vorgestellt. Schließlich bin ich noch kurz auf die eigentliche Beleuchtungsberechnung beim bidirektionalen Path Tracing eingegangen und habe den zentralen Ablauf des Algorithmus näher erläutert. Im nächsten Kapitel werden die Ergebnisse der Programmierung dokumentiert und gezeigt, inwiefern sich die bisherigen Überlegungen umsetzen ließen.

Kapitel 3
Ergebnisse

In diesem Kapitel werden die Implementierungsergebnisse gezeigt. Dabei wird in erster Linie Wert auf selbst gerenderte Bilder und Auszüge aus dem Quellcode des Programms gelegt. Zunächst habe ich den Ray Tracer um eine Flächenlichtquelle erweitert, um zu sehen, wie sie arbeitet und ob die Generierung von Zufallspunkten gut ist. Die Ergebnisbilder dazu werden zuerst gezeigt. Anschließend zeige ich aus dem Quellcode der geänderten Klassen die wichtigsten Ausschnitte, die sich auf die Ansätze und Ideen aus dem zweiten Teil von Kapitel 2 beziehen. Abschließend werden ein paar Ergebnisbilder vorgestellt und versucht, verschiedene Szenenbeschreibungen mit dem bidirektionalen Path Tracer zu rendern.

3.1 Die Flächenlichtquelle im Ray Tracer

Beim Ray Tracing werden die Augenstrahlen in die Szene geschickt und mit Objekten geschnitten. Von jedem Schnittpunkt aus werden Schattenfühler zur Lichtquelle aufgebaut, um die Beleuchtung am aktuellen Schnittpunkt korrekt berechnen zu können. Wenn eine Punktlichtquelle oder ein Spotlight vorliegen, dann wird der Schattenfühler vom Schnittpunkt zum Ausgangspunkt der Lichtquelle aufgebaut. Bei einer Flächenlichtquelle dagegen gibt es mehrere Möglichkeiten, einen Ausgangspunkt für den Schattenfühler zu finden. Die folgende Abbildung 3.1 verdeutlicht diesen Sachverhalt.



(Abbildung 3.1)
Schattenfühler bei einer Flächenlichtquelle

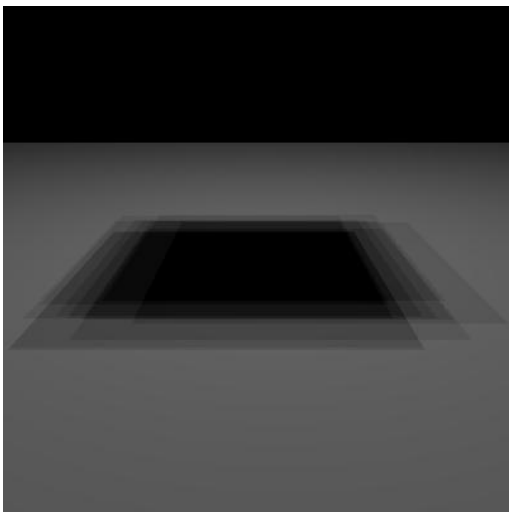
In der Abbildung 3.1 wird ein Punkt auf der Fläche F beleuchtet. Der graue Würfel verdeckt aus der Sicht des beleuchteten Punktes teilweise die rechteckige Flächenlichtquelle. Sie hat eine Gesamt-Intensität von I. In diesem Beispiel werden vom Punkt auf der Fläche F aus vier Schattenfühler Richtung Lichtquelle gesendet. Dabei gibt es zwei Möglichkeiten: entweder sie treffen ein dazwischen liegendes, undurchsichtiges Objekt (hier der grau gefärbte Würfel),

wie die Schattenfühler vom Typ A, oder sie treffen ungehindert die Lichtquelle, wie hier die Schattenfühler vom Typ B. Da in diesem Beispiel vier Schattenfühler eingesetzt werden und davon zwei ein Objekt treffen und zwei davon nicht, wird der Punkt mit 0.5 * I beleuchtet.
Das ergibt für die endgültige Intensität:

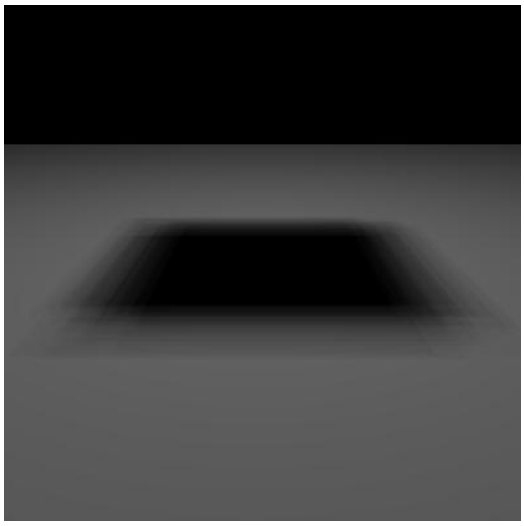
$I_{neu} = 0 + 0 + ¼ * I + ¼ * I.$

Es ist also möglich, ein Objekt so zwischen die Flächenlichtquelle und einen Punkt, der beleuchtet werden soll, zu platzieren, dass die Flächenlichtquelle aus Sicht des Objektpunktes halb verdeckt wird. Der Objektpunkt wird dann nur noch mit halber Intensität der Flächenlichtquelle beleuchtet. Das ist bei Punktlichtquellen nicht möglich: hier wird ein Punkt entweder beleuchtet oder nicht.

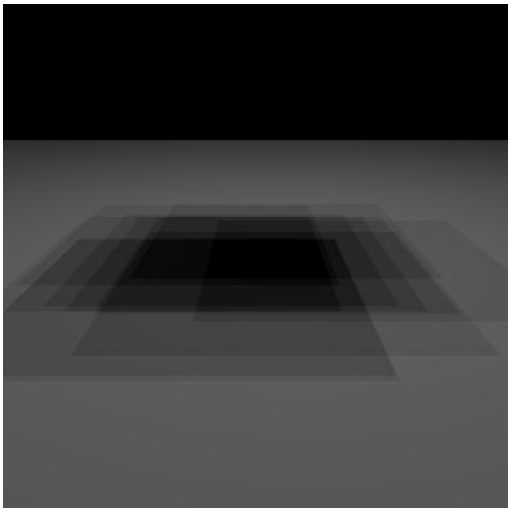
Wegen dieser besonderen Eigenschaft von Flächenlichtquellen ist es möglich, Halbschatten darzustellen. Die Grenze des Schattens ist dann nicht als scharfe Kante sichtbar, sondern der Schattenbereich auf dem beleuchteten Objekt geht kontinuierlich in den beleuchteten Bereich über. Die Größe dieses Übergangsbereiches hängt von der Größe der Flächenlichtquelle ab. Der Schatten wird dann wie in der Physik in Halb- und Kernschatten unterteilt. Der Übergang zwischen Kern- und Halbschatten bis zum beleuchteten Bereich wird fließender, wenn mehr Schattenfühler zur Flächenlichtquelle gesendet werden. Die folgenden Bilder zeigen die Abhängigkeit zwischen Anzahl der Schattenfühler bzw. Größe der Flächenlichtquelle und dem abgebildeten Schatten. Sie wurden mit dem Ray Tracer gerendert, daher sind die Ergebnisbilder nicht verrauscht.



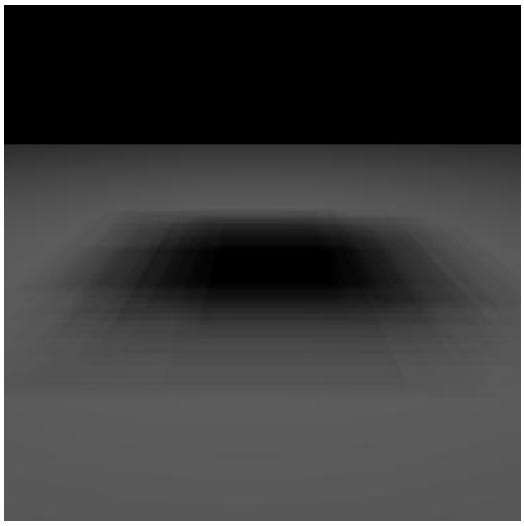
(Abbildung 3.2)
eckige Lichtquelle der Größe 2x2,
10 Schattenfühler pro Schnittpunkt
eines Objektes



(Abbildung 3.3)
eckige Lichtquelle der Größe 2x2,
50 Schattenfühler pro Schnittpunkt
eines Objektes



(Abbildung 3.4)
eckige Lichtquelle der Größe 4x4,
10 Schattenfühler pro Schnittpunkt
auf den Objekten



(Abbildung 3.5)
eckige Lichtquelle der Größe 4x4,
50 Schattenfühler pro Schnittpunkt
auf den Objekten

Aus den obigen Abbildungen von Schattenbereichen wird deutlich, dass bei gleicher Größe der Lichtquelle der Übergang vom Schattenbereich in den beleuchteten Bereich auf der grauen Basisfläche fließender wird. Dies wird beim Vergleich der obigen beiden Bilder deutlich sowie bei Vergleich der unteren beiden Bilder. Außerdem ist bei Vergleich der unteren beiden Bilder mit den oberen beiden Bildern sichtbar, dass eine größere Flächenlichtquelle mit wesentlich mehr Schattenfühlern abgetastet werden muss als eine kleinere, um annähernd den gleichen Übergang vom Schattenbereich zum beleuchteten Bereich zu erhalten.

3.2 Realisierung des bidirektionalen Path Tracing Algorithmus' durch Änderungen im Ray Tracer

Erweiterung der Klassen für die Lichtquellen

Die Klassen Light, Spotlight und die neue Klasse AreaLight mussten für den bidirektionalen Path Tracer erweitert werden um Methoden, die die primären Lichtstrahlen generieren. Im folgenden Abschnitt werden zuerst Auszüge aus dem Quellcode gezeigt und dann gerenderte Bilder gegenübergestellt, die nur eine rechteckige Bodenfläche zeigen, über die jeweils einmal eine Punktlichtquelle, ein Spotlight und eine Flächenlichtquelle platziert wurde, um die Unterschiede in der Beleuchtung zu erkennen.

3.2.1 Primärstrahlen bei der Punktlichtquelle

Wie bereits im 2. Kapitel erklärt wurde, versendet eine Punktlichtquelle Strahlen in alle beliebigen Raumrichtungen. Daher werden zufällig Richtungsvektoren bestimmt. Die Werte der Komponenten liegen im Bereich [-1…1], so dass ein Vektor auch in Richtung negativer Achse zeigen kann. Der folgende Quelltext stammt aus der Methode „getPrimaryLightRaysPoint()“ der Klasse Light:

(aus light.cpp):

```
vector<Ray*> Light::getPrimaryLightRaysPoint(int anzahlPrimaryRays)
{
	// Ablauf beim Bestimmen der Zufallspunkte:
	/*
	- drei Koordinaten eines Vektors zufällig wählen;
	- dieser Vektor ist Richtungsvektor für den neuen Strahl;
	- neuen Strahl erzeugen mit dieser neuen Richtung;
	- Ausgangspunkt des Strahles ist m_Position;
	- dem Primärstrahl die Intensität der Lichtquelle mit übergeben;
	- neuen Strahl als primären Lichtstrahl an die Liste lightRaylist übergeben;
	- wiederhole, bis anzahlPrimaryRays Strahlen generiert wurden; (=for-Schleife)
	*/

	// Sonderfall abfragen bzw. vector erstmal löschen;
	if (m_lightRaylist.size() > 1)
		m_lightRaylist.clear();

	// für den Zufallspunkt auf der Sphäre werden drei Punkte zufällig bestimmt:
	// drei Koordinaten zufällig bestimmen, die im Bereich [-1 .. 1] liegen;

	for (int lauf= 0; lauf < anzahlPrimaryRays; lauf++)
	{
		Real zahl1 = 2.0*(Real)rand()/RAND_MAX - 1;
		Real zahl2 = 2.0*(Real)rand()/RAND_MAX - 1;
		Real zahl3 = 2.0*(Real)rand()/RAND_MAX - 1;

		// Richtungsvektor für den Lichtstrahl erzeugen:
		Vector3D* neueRichtung = new Vector3D(zahl1, zahl2, zahl3);
		// neuen Strahl erzeugen:
		// Parameter für den Konstruktor aus der Klasse Ray sind:
		// - m_Position als Startpunkt
		// - neueRichtung als Richtungsvektor
		// - m_Intensity als Intensitätswert;
		Ray* strahl = new Ray(m_Position, (*neueRichtung), m_Intensity);

		// Strahl zur Liste hinzufügen:
		m_lightRaylist.push_back(strahl);
	}	// ende der for-Schleife;

	// Rückgabe des Vektors mit primären Lichtstrahlen;
	return m_lightRaylist;
}	// ende der Methode getPrimaryRaysPoint
```

Der Quotient (rand()/RAND_MAX) liefert Werte im Bereich [0..1]. Nach der Multiplikation mit 2 liegen die Werte im Bereich [0..2]. Nach der Subtraktion von 1 liegen sie im Bereich [-1..1]. Der Rückgabewert der Methode „getPrimaryRayPoint()“ ist ein Vektor von Rays, der die primären Lichtstrahlen der Punktlichtquelle enthält und zum ersten Mal verwendet wird in der Klasse Raytracer, wenn die Lichtstrahlen an die Trace-Routine übergeben werden.

3.2.2 Primärstrahlen beim Spotlight

Die Klasse Spotlight wurde ebenfalls um eine Methode zur Strahlengenerierung erweitert. Hier musste nur sichergestellt werden, dass die Strahlen innerhalb des Lichtkegels die Lichtquelle verlassen. Der Ansatz dazu wurde bereits im zweiten Teil von Kapitel 2 erklärt. Jetzt wird das wichtigste aus der Methode „getPrimaryLightRays()“ gezeigt.

(aus spotlight.cpp):

```
// zur Bestimmung von primären Lichtstrahlen vom Spotlight aus:
vector<Ray*> Spotlight::getPrimaryLightRays(int anzahlPrimaryRays)
{
	(…)

	// falls x-Komponente von m_direction != 0, dann kann durch diese geteilt werden:
	// und es wird nach der x-Komponente des Vektors v aufgelöst:
	// x so berechnen, dass m_direction * (Zufallsvektor v) = 0 (Skalarprodukt)

	(…)
	// Schleife, die über die gewünschte Anzahl der Primärstrahlen läuft:
	for (int i1 = 0; i1 < anzahlPrimaryRays; i1++)
	{
			// den zu m_direction senkrecht stehenden Vektor bestimmen;
			v2 = 2.0*(Real)rand()/RAND_MAX - 1; // ergibt Zufallswert zwischen [-1..1]
			v3 = 2.0*(Real)rand()/RAND_MAX - 1;
			v1 = (-m_direction.getY()* v2 - m_direction.getZ()* v3) / m_direction.getX();

			/// Koordinaten des Zufallsvektors (x,y,z) sind bekannt;
			v = new Vector3D(v1,v2,v3);

			/// zufallsvektor v in m_Position angesetzen (=Vektor mp);
			// führt zum zufallspunkt P;
			// wichtig ist aber hier nur der Vektor, nicht der Punkt;

			/// Zufallsvektor v normalisieren:
			(*v).normalize();

			// zufällig den Winkel theta bestimmen, der von m_angle begrenzt wird:
			// theta liegt in [0..m_angle]
			Real theta = (Real)rand()/RAND_MAX * getAngle();

			// die Längen x und y berechnen mit theta über sin und cos:
			// Hypothenuse ist der Radius der Hemisphäre und gleich 1;
			Real x = sin(theta);
			Real y = cos(theta);
			// v wird jetzt auf die Länge von x begrenzt:
			(*v) *= x;
			// dadurch zeigt v vom Mittelpunkt aus auf die Stelle, an die
			// der Punkt C von der Hemisphäre auf die Kreisfläche projiziert würde;

			/// die Normale m_direction wird mit y skaliert, neuen Vektor tmp anlegen;
			Vector3D tmp = getDirection();
			tmp.normalize();
			tmp *= y;

			/// neuer Zufallspunkt auf der Hemisphäre und innerhalb des Kegels vom Spotlight:
			// erhält man durch Vektoraddition von m_Position aus mit
			// tmp und v;

			/// wichtig ist aber nur die neue Richtung für den Strahl: tmp + (*v)
			/// und die Intensität m_Intensity für den Primärstrahl;
			Vector3D neueRicht = tmp + (*v);
			Ray* neuerPrimar = new Ray(m_Position, neueRicht, m_Intensity);

			m_lightRaylist.push_back(neuerPrimar);

	}	// ende der for-Schleife für die x-Komponente != 0 von m_direction;
```

Der obige Auszug ist aus einer if-Bedingung entnommen, die prüft, ob die x-Komponente des Vektors „m_direction“ ungleich null ist. Der Vektor „m_direction“ war der Vektor, der die Richtung des Spotlights angibt. Der Zufallsvektor (v1, v2, v3) soll senkrecht zu m_direction stehen. Wenn wie im obigen Quellcodeauszug nach v1 aufgelöst werden soll, dann geht das nur, wenn die x-Komponente von m_direction ungleich Null ist. Daher gibt es drei if-Bedingungen in der Methode „getPrimaryRays()“, die einzeln abfragen, welche Komponente von m_direction ungleich Null ist, damit entsprechend nach v1, v2 oder v3 aufgelöst werden kann, ohne versehentlich durch null zu teilen.
Die Koordinaten der Richtungsvektoren wurden testweise als Zahlen ausgegeben. Die Ausgaben ließen Rückschlüsse darauf zu, dass die Lichtstrahlen nur innerhalb des Lichtkegels des Spotlights in die Szene wanderten.

3.2.3 Primäre Lichtstrahlen bei der Flächenlichtquelle

Die Klasse AreaLight wurde nicht nur um eine Methode zur Generierung von Primärstrahlen, sondern zuerst um zwei Methoden zur Punktegenerierung auf der Fläche selbst erweitert. Die Methode „squareShapePoints()“ sampelt Zufallspunkte auf einer rechteckigen Fläche und die Methode „roundShapePoints()“ sampelt Zufallspunkte auf einer runden Fläche. Beide Methoden geben einen gefüllten Punktevektor zurück, der dann in der Methode „getPrimaryRays()“ die Startpunkte zur Bestimmung von primären Lichtstrahlen enthält. Die Arbeitsweise der Methoden zur Punktegenerierung wurde bereits im zweiten Teil von Kapitel 2 angesprochen. Wichtig ist hier jetzt nur die Methode „getPrimaryLightRays()“:

(aus arealight.cpp):

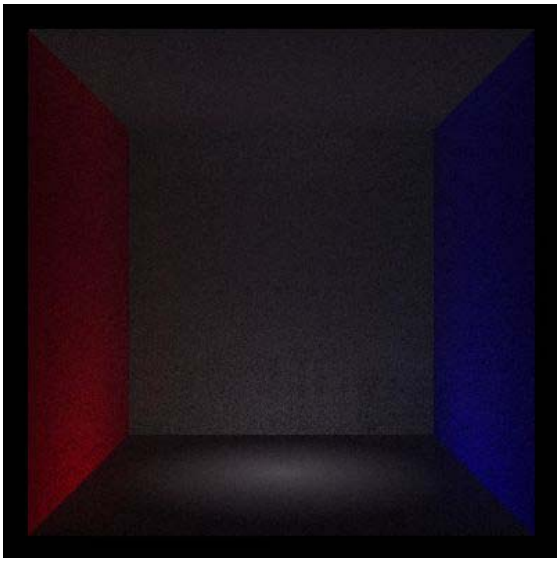
```
vector<Ray*> AreaLight::getPrimaryLightRays(int anzahlPrimaryRays)
{

        (…) // Sonderfälle abfangen;

        // zum Zwischenspeichern der Zufallpunkte der eckigen bzw.
        // runden Flächenlichtquelle:
        vector<Point*> basispunkte;
        /// Radius der Hemisphäre;
        Real radiusHemisphere;

        /// Abfrage, ob es eine eckige oder runde Fläche ist;
        // falls es eine runde Lichtquelle ist, tue:
        // (Länge und Breite wurden im Konstruktor auf null gesetzt für runde Lichtqu.)
        if ((m_width == 0)&&(m_height == 0))
        {
                // dann wird die Methode zum Generieren von Zufallspunkten auf
                // der runden Fläche aufgerufen mit Parameter anzahlPrimaryRays
                // Ergebnis wird den basispunkten zugewiesen;
                basispunkte = roundShapePoints(anzahlPrimaryRays);
                radiusHemisphere = m_radius;
        }
        else if (m_radius == 0)     // dann ist es eine eckige Lichtquelle:
                                                        // weil im Konstruktor für eine eckige Lichtqu.
                                                        // der Radius auf null gesetzt wurde;
        {
                // Aufrufen der Methode zur Punktegenerierung auf einer eckigen Fläche;
                basispunkte = squareShapePoints(anzahlPrimaryRays);
                // Berechnung des Radius' der Hemisphäre bei eckiger Flächenlichtquelle:
                // es gilt: (Breite/2)^2 + (Länge/2)^2 = (radiusHemisphere)^2
                // Auflösen nach heimsphere durch Berechnen der Wurzel;
```

```
        radiusHemisphere = sqrt((m_width/2.0)*(m_width/2.0) + (m_height/2.0)*(m_height/2.0));
}


/// Schleife über alle Punkte, die zufällig auf der
/// Flächenlichtquelle abgetastet wurden
for (int f = 0; f < anzahlPrimaryRays; f++)
{
        /// Zufällig zwei Punkte P1 und P2 aus der Liste der Flächenpunkte aussuchen:
        /// einen Index zufällig bestimmen, der zum Punkt P1 führt:
        /// index liegt im Bereich [0..(anzahlPrimaryRays - 1)]
        int randomIndex = (Real)rand()/RAND_MAX *(basispunkte.size() - 1);
        Point* p1 = basispunkte[randomIndex];
        /// zweiten Index zufällig bestimmen:
        randomIndex = (Real)rand()/RAND_MAX *(basispunkte.size() - 1);
        Point* p2 = basispunkte[randomIndex];

        /// Projektion des Punktes P1 auf die Hemisphäre; ergibt den Punkt H;
        // dazu: zunächst MP1 bestimmen und seine Länge berechnen
        // (M = Mittelpunkt m_Position aus der Oberklasse)
        Vector3D mp1 = (*p1) - m_Position;
        Real streckeMP1 = mp1.getLength();

        // über Pythagoras die Strecke P1H berechnen (vgl. Skizzen in StudiArbeit dazu)
        Real streckeP1H = sqrt((streckeMP1*streckeMP1) + (radiusHemisphere*radiusHemisphere));

        // die Normale m_lightNormal der Fläche hat Anfangs die Länge eins;
        // wird jetzt skaliert mit streckeP1H, der Länge der Strecke von P1 nach H;
        // dadurch wird die Normale direkt auf die richtige Länge skaliert;
        m_lightNormal.normalize();
        Vector3D projektion = m_lightNormal*streckeP1H;

        /// man gelangt zum proijzierten Punkt H durch Vektoraddition
        // von "projektion" auf P1:
        Point h = (*p1) + projektion;
        // Richtungsvektor für neuen Primärstrahl ergibt sich dann aus: P2 H;
        Vector3D neueRichtung = h - (*p2);

        // falls der Winkel zwischen der Normalen und dem Richtungsvektor zu groß wird,
        // wird neueRichtung "nach oben" korrigiert durch Addition des Normalenvektors auf
        // neueRichtung
        // dann läuft der Vektor neueRichtung weniger parallel von der Flächenlichtquelle weg;

        // Winkel "angle" zwischen Normale und neueRichtung bestimmen über Skalarprodukt:
        Real angle = acos(m_lightNormal.dotprod(neueRichtung) /
                                        (m_lightNormal.getLength() * neueRichtung.getLength()));

        if (angle > (PI/4.0))          // falls Winkel größer als 45 Grad (im Bogenmap pi/4)
        {
                // Korrektur der neuenRichtung durch Vektoraddition mit der Normalen;
                neueRichtung = neueRichtung + m_lightNormal;
        }

        // m_Intensity steht in der Oberklasse Light, wird hier cosinus-gewichtet zur Normalen:
        // Winkel zwischen Normale und neueRichtung bestimmen udn als Faktor für die
        // m_Intensity nehmen:
        Real cosAlpha = m_lightNormal.dotprod(neueRichtung)/
                                        (neueRichtung.getLength() * m_lightNormal.getLength());
        Color neueFarbe = getIntensity() * cosAlpha;

        Ray* lightRay = new Ray((*p2), neueRichtung, neueFarbe);
```

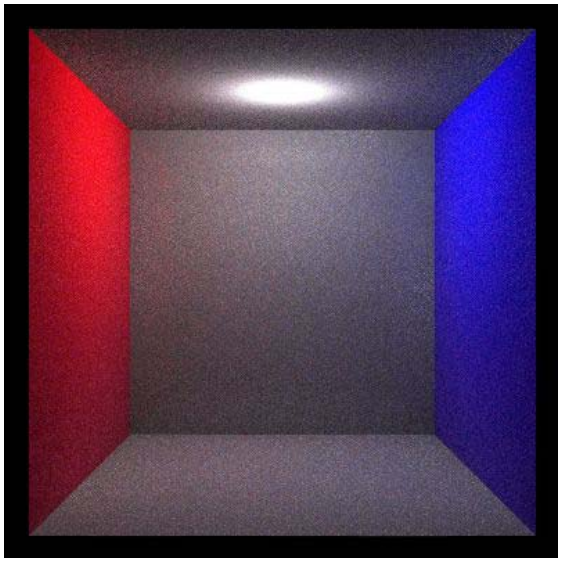
```
                // neue Strahl zur Liste der Primärstrahlen hinzufügen;
                m_lightRaylist.push_back(lightRay);

        }       // ende der for-Schleife über alle Primärstrahlen;
        // jetzt ist die Liste der Primärstrahlen gefüllt und kann zurückgegeben werden;
        return m_lightRaylist;
}       // ende der Methode getPrimaryLightRays()
```

Der Ablauf der Methode „getPrimaryLightRays()“:
Zuerst wird abgefragt, ob es eine runde oder eine eckige Lichtquelle ist. Falls es eine runde Fläche ist, dann wird die Methode „roundShapePoints()“ zur Generierung von Punkten auf der runden Fläche aufgerufen. Andernfalls ist es eine eckige Fläche und die Punkte werden über die Methode „squareShapePoints()“ bestimmt. Beide Methoden liefern einen gefüllten Punktevektor zurück. Aus diesem Vektor wird mit einem zufällig bestimmten Index ein Punkt P1 ausgesucht. Er wird auf eine Hemisphäre projiziert, die um die Flächenlichtquelle aufgebaut wird. Es wird ein zweiter Punkt P2 aus dem Punktevektor zufällig gewählt und als Startpunkt für den Lichtstrahl verwendet. Dann ergibt sich die Richtung aus dem Vektor von P2 zum projizierten Punkt P1. Mit diesen Angaben wird ein neuer Strahl erzeugt und in die Liste der Primärstrahlen aufgenommen, die die Methode „getPrimaryLightRays()“ zurückliefert.

Jetzt wird ein einfacher Szenenaufbau nacheinander mit den verschiedenen Lichtquellen beleuchtet und das Ergebnisbild verglichen.



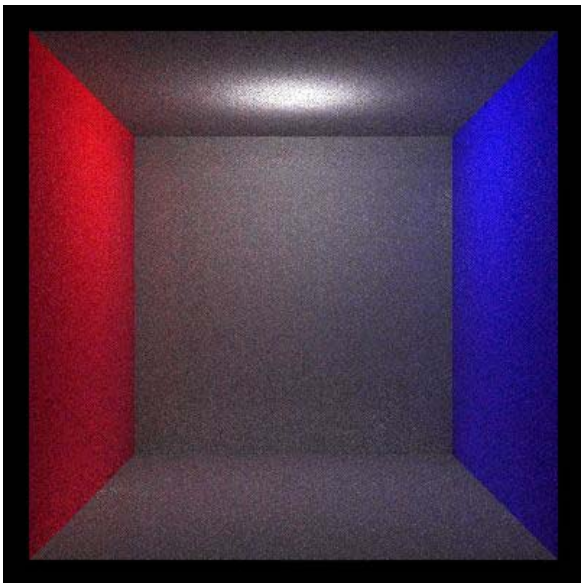
(Abbildung 3.6)
eine Cornell-Box, mit einem Spotlight ausgeleuchtet



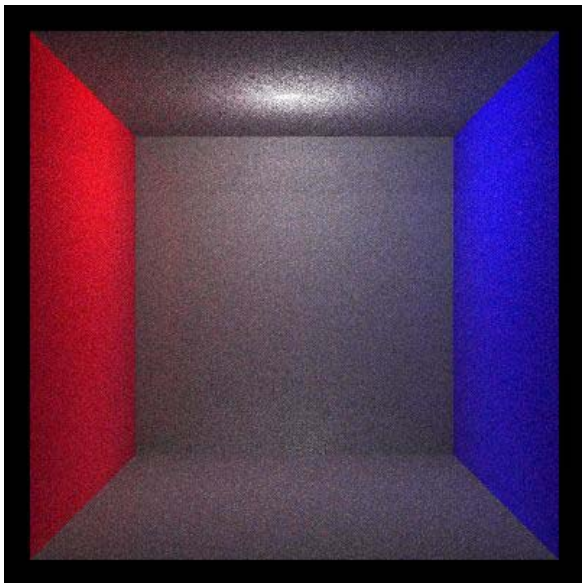
(Abbildung 3.7)
Cornell-Box, mit einer Punktlichtquelle augeleuchtet

Das linke Bild zeigt die Cornell-Box, wie sie mit einem Spotlight ausgeleuchtet wurde. Im Gegensatz zur Beleuchtung mit einer Punktlichtquelle wie im rechten Bild ist das linke wesentlich dunkler, obwohl beide Lichtquellen dieselbe Intensität haben. Auffällig ist auch, dass die Punktlichtquelle die Decke mit beleuchtet und das Spotlight nicht. Der Grund dafür ist, dass für jeden Schattenfühler von einem Punkt auf einem Objekt zurück zur Lichtquelle überprüft wird, ob der Schattenfühler in den Lichtkegel des Spotlights eintritt. Wenn nicht, dann wird für diesen Objektpunkt keine direkte Beleuchtung durch das Spotlight berechnet. Hinzu kommt, dass beim Spotlight die Intensität der Lichtstrahlen zum Lichtkegelrand hin abnimmt. Daher ist hier der Übergang vom beleuchteten in den dunklen Bereich fließend.

Bei der Punktlichtquelle braucht nicht überprüft werden, ob ein Schattenfühler in einen Lichtkegel fällt, weil für eine Punktlichtquelle kein Lichtkegel bestimmt wird, sondern sie von jedem Standpunkt aus sichtbar ist und daher auch jeder Objektpunkt direkt beleuchtet wird. Die Decke ist deshalb so stark beleuchtet, weil die Intensität mit dem Kehrwert der Entfernung zwischen Objektpunkt und Lichtquelle gewichtet wird. Die Intensität für die Objektpunkte, die nahe bei der Punktlichtquelle liegen, wird daher mit einem hohen Faktor multipliziert und es wird ein hoher Beleuchtungswert berechnet.



(Abbildung 3.8)
Cornell-Box, mit einer runden Flächenlichtquelle ausgeleuchtet (Radius: 0.2)



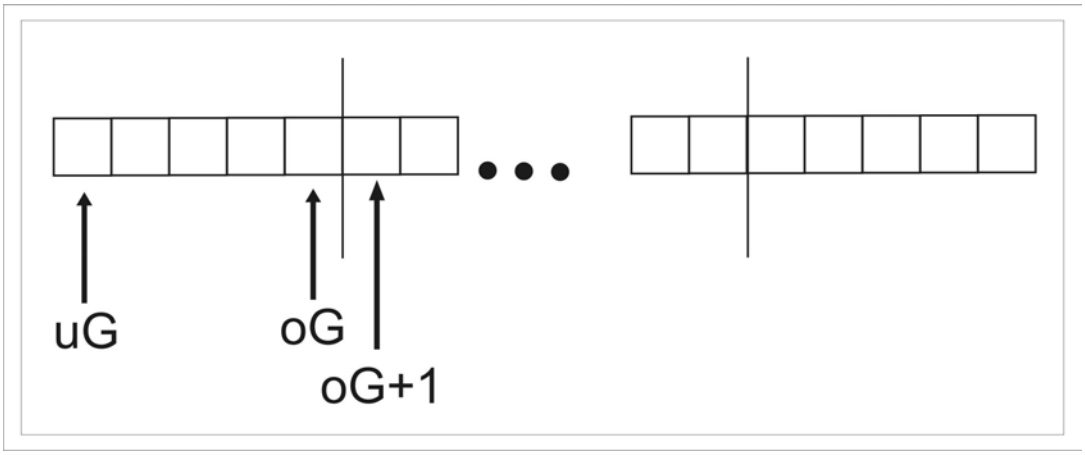
(Abbildung 3.9)
Cornell-Box, mit einer eckigen Flächenlichtquelle ausgeleuchtet (Breite = Länge = 0.2)

Diese beiden Bilder wurden mit Flächenlichtquellen ausgeleuchtet. Das linke Bild mit einer runden Flächenlichtquelle mit Radius 0.2 und das rechte mit einer rechteckigen Flächenlichtquelle der Größe 0.2 x 0.2. Weil die eckige Fläche einen kleineren Flächeninhalt hat als die runde, werden die Primärstrahlen hier stärker um den Mittelpunkt der Fläche konzentriert. Daher ist im rechten Bild an der Decke der Cornellbox ein kleinerer Lichtfleck zu sehen als im linken.
Sowohl die Bilder der Flächenlichtquellen als auch die der Punktlichtquelle und des Spotlights wurden mit 10 Strahlen pro Pixel abgetastet.

3.2.4 Aufrufen der Lichtstrahlen in der Klasse Raytracer

Die Methoden zur Generierung von Lichtstrahlen werden zum ersten Mal in der Methode „getPartOfImage()“ in der Klasse Raytracer aufgerufen. In dieser Methode wird ein Teilbild der Szene mit Augenstrahlen abgetastet. Dazu wird über einen Zeiger „m_camera“ auf die Kameraklasse die Methode „getPrimaryRays(m_sSampling, x0, y0, x1, y1);“ aufgerufen. Diese Methode gibt die primären Augenstrahlen der Kamera zurück, die im Bildbereich (x0, y0) bis (x1, y1) liegen und speichert sie in „m_raylist“. Weil beim bidirektionalen Path Tracing auch primäre Lichtstrahlen gebraucht werden, wird jetzt das gleiche für die Lichtquellen getan. In einer Schleife über alle Lichtquellen werden nacheinander über die bereits beschriebenen get-Methoden „getPrimaryLightRaysPoint()“ bzw. „getPrimaryLightRays()“ die Lichtstrahlen der jeweiligen Lichtquelle aufgerufen. Weil es

mehrere Lichtquellen in einer Szene geben kann, werden die einzelnen Vektoren mit primären Lichtstrahlen zu einem großen zusammengesetzt. Das heißt, wenn beim bidirektionalen Path Tracing immer nur ein Augenstrahl mit einem Lichtstrahl einer Lichtquelle verknüpft wird, wird bei N Lichtquellen in der Szene ein Augenstrahl mit N Lichtstrahlen verknüpft, wobei keine zwei von N Lichtstrahlen zu einer Lichtquelle gehören dürfen. Das heißt, der Index, der zufällig aus dem großen Vektor mit allen primären Lichtstrahlen die Lichtstrahlen auswählt, muss in jedem Schleifendurchlauf neu gewählt werden, und zwar so, dass er immer einen Strahl von einer einzigen Lichtquelle auswählt, und das für alle N Lichtquellen der Szene. Die folgende Abbildung 3.10 verdeutlicht dieses Problem.



(Abbildung 3.10)
Die Kästen repräsentieren Vector-Elemente und enthalten Zeiger auf Instanzen der Klasse Ray. Die Indices „uG“ und „oG“ Grenzen die Strahlen ein, die zur ersten einer Lichtquelle gehören. Der Index (oG + 1) ist die untere Grenze für den Index, der einen Strahl aus der zweiten Lichtquelle auswählt.

Die Vektorelemente des Vektors mit allen Lichtstrahlen enthalten Zeiger auf Instanzen der Klasse „Ray“. Die senkrechten Striche in der Abbildung markieren die Zusammengehörigkeit von Lichtstrahlen zu einer Lichtquelle. Die Elemente vom Index 0 bis zum Index i gehören zu einer Lichtquelle, die Elemente vom Index (i+1) bis zum Index j gehören zu einer anderen Lichtquelle usw. Ein Zufallsindex wird jetzt so bestimmt, dass er immer zufällig in einem Indexbereich einer Lichtquelle einen Strahl auswählt. Dieser Strahl wird dann an die Trace-Methode „raytraceLight()“ in der Klasse Raytracer übergeben, die den Lichtstrahl traversiert und in der Materialklasse die Methode „getBiPathColor()“ aufruft, die den rekursiven Aufruf enthält. Im nächsten Schleifendurchlauf über die Lichtquellen wird dann ein Strahl einer anderen Lichtquelle ausgewählt. Diese Schleife ist unabhängig von der, in der die primären Lichtstrahlen aufgesammelt werden. Denn der Aufruf eines primären Lichtstrahles über den Zufallsindex und der Aufruf der Methode „raytraceLight()“ stehen in der Schleife über alle primären Augenstrahlen in der Methode „getPartOfImage()“.
Der Indexbereich für eine Lichtquelle wird durch eine untere (uG) und eine obere Grenze (oG) festgelegt. Im ersten Schleifendurchlauf über alle Lichtquellen ist die uG = 0 und oG entspricht der Anzahl der Primärstrahlen, die für die erste Lichtquelle generiert wurden. Im nächsten Schleifendurchlauf, ist die untere Grenze uG = (alte oG) + 1. Die obere Grenze wird weiter geschoben und zeigt jetzt auf das letzte Element im Indexbereich der zweiten Lichtquelle. Der Zufallsindex muss dann nur im Bereich [uG…oG] liegen. Das wird in der Schleife über alle Lichtquellen so lange wiederholt, bis von jeder Lichtquelle ein Lichtstrahl ausgewählt wurde.
Die Lichtstrahlen werden einzeln an die raytraceLight-Methode übergeben.

(aus raytracer.cpp, Methode void getPartOfImage(…)):

```
// vorher:
uG = oG = 0;
for (int laufIndex = 0; laufIndex < m_allLights.size(); laufIndex++)
{
	// im ersten Schleifendurchlauf gilt: neue obere Grenze = Anzahl
	// der Strahlen der ersten Lichtquelle-1;
	oG = (m_numberRaysPerLight[laufIndex]-1) + uG;

	// Zufallsindex bestimmen zwischen 0 und
	// (Anzahl Lichtstrahlen in der aktuellen Lichtquelle ):
	int zufallsindex = (Real)rand()/RAND_MAX
				* (m_numberRaysPerLight[laufIndex]-1);

	// diese Zufallszahl soll aber zwischen uG und oG liegen, daher:
	zufallsindex += uG;

	// mit diesem Index einen Strahl aussuchen, aus dem großen Vector
	// m_lightRaylistAll (wegen der Grenzen wird die Zugehörigkeit zur Lichtqu.
	// sichergestellt):
	// ein Lichtstrahl für einen Augenstrahl;
	Ray* einLichtstrahl = m_lightRaylistAll[zufallsindex];

	// anhand dieses Lichtstrahles lassen sich gut die Informationen bestimmen, die
	// für das LocalGeoemtry der Lichtquelle selbst wichtig sind:

	// für jeden Lichtquellentyp ist gleich:
	// Stützstelle des LocalGeom. = Startpunkt des Lichtstrahles in/auf der Lichtquelle:
	Point* m_startpkt = new Point(0, 0, 0);
	(*m_startpkt) = einLichtstrahl->getOrigin();
	// Farbe der Lichtquelle ist auch immer gleich:
	Color* m_farbeLicht = new Color(0, 0, 0);
	(*m_farbeLicht) = m_allLights[laufIndex]->getIntensity();
	// Strahl, der den Schnittpunkt verursacht hat = Strahl vom Auge zur Lichtquelle;
	// genauer: vom Auge zum Startpkt des primären Lichtstrahles = startpkt - eyePoint;
	Vector3D eyeLightVector = (*m_startpkt) - eyePoint;
	Ray* m_rayEyetoLight = new Ray(eyePoint, eyeLightVector);
	// Abstand dist = Abstand Startpunkt - EyePoint:
	Real abstand = eyeLightVector.getLength();

	// für die Normalenberechnung (normaleLight) muss in die Lichttypen AreaLight,
	// Spotlight und Punktlichtquelle unterschieden werden:
	Vector3D* m_normaleLight = new Vector3D(0, 0, 0);
	// falls es eine Lichtquelle ist: Normale = getNormal();
	if (m_allLights[laufIndex]->isArealight())
		(*m_normaleLight) = ((AreaLight*)m_allLights[laufIndex])->getNormal();

	else if (m_allLights[laufIndex]->isSpotlight())
		(*m_normaleLight) = ((Spotlight*)m_allLights[laufIndex])->getDirection();

	else // dann ist es eine Punktlichtquelle und die Normale ist der Vektor vom Licht zum Auge:
	{
		(*m_normaleLight) = eyeLightVector * (-1);
	}

	// mit diesen Daten wird jetzt das LocalGeometry für die Lichtquelle selbst angelegt:
	// danach wird über die Set-Methode die Intensität für dieses LocalGeom gesetzt:
	// als SceneItem wird NULL übergeben und distance ist Abstand Lichtquelle-Auge;
	LocalGeometry* lightVertexFirst =
		new LocalGeometry(m_farbeLicht, NULL, abstand, m_startpkt,
					m_normaleLight, m_rayEyetoLight);
```

```
            // über die set-Methode wird in diesem LocalGeometry m_LightRayIntensity:
            // gesetzt wird dafür die Intensität des Lichtstrahles, der zumindest bei der AreaLight
            // von der Intensität der Lichtquelle selbst verschieden sein kann, wegen Cosinusgewichtung:
            Color intensRay = einLichtstrahl->getIntensityPart();
            lightVertexFirst->setLightRayIntensity(intensRay);

            // so: dieses LocalGeometry der Lichtquelle wird jetzt (in der Schleife) zu
            // m_LightRayVertices (=Membervariable in dieser Klasse) hinzugefügt:
            m_lightRayVertices.push_back(lightVertexFirst);

            // den neuen Lichtstrahl an die raytraceLight-Methode in dieser Klasse
            // übergeben, die den LocalGeometry-Vector auffüllt, der global in dieser
            // Klasse definiert wurde: (m_lightRayVertices)

            raytraceLight(einLichtstrahl, m_recDepthLightray);

            delete m_rayEyetoLight;

             uG = oG + 1;    // für den nächsten Schleifendurchlauf: neuer Indexbereich;
}           // ende for-Schleife über alle Lichtquellen zum Aussuchen von prim. Lichtstrahlen
```

In der raytraceLight()-Methode der Klasse Raytracer wird der Vektor „m_lightRayVertices()“ aus LocalGeometries mit den Schnittpunkten aufgefüllt, die beim Auftreffen des traversierten Lichtstrahles mit Objekten der Szene entstanden. Dieser Vektor mit Schnittpunkten des Lichtstrahles wird dann an die „raytraceBiPath()“-Methode übergeben, die den Augenstrahl traversiert und dabei von jedem Schnittpunkt des Augenstrahles mit den Objekten Schattenfühler zu allen Schnittpunkten des Lichtstrahles versendet.

Die Methode „raytraceLight()“ sieht wie folgt aus:
(aus Raytracer.cpp:)

```
void Raytracer::raytraceLight(Ray* lightray, int recursionDepthLight)
{
        // Abfrage nach Rekursionsabbruch:
        if (recursionDepthLight == 0)
                return;

        // Lichtstrahl mit Szenenobjekten schneiden:
        LocalGeometry *lg_light=m_scene->intersect(lightray);

        // zurückgegeben wird von der Intersect-Routine ein Zeiger auf ein LocalGeometry:
        // falls ungleich NULL, dann existiert der Schnittpunkt:
        if (lg_light != NULL)
        {
                GeomObject* go= (GeomObject*)lg_light->getSceneItem();

                // über den Materialzeiger wird gleich die Methode "changeLightVertex()"
                // aufgerufen, die die Intensität verrechnet:
                Material* mat = go->getMatPointer();

                // Intensität aus dem Lichtstrahl holen und für den Stützpunkt
                // des Lichtstrahles setzen; (ist in LocalGeom. vom Typ "Color", also kein Zeiger!
                Color strahlenIntens = lightray->getIntensityPart();

                // Lichtstützpunkt lg_light erhält die ankommende Intensität;
                // in LocalGeom. ist jetzt die Set-Methode auf "Color" und nicht
                // mehr auf "Color*" eingestellt:
                lg_light->setLightRayIntensity(strahlenIntens);
```

```
                // dieses lg_light wird jetzt an die Methode changeLightVertex übergeben
                // und bzgl. seiner Intensität verändert:
                // ist die Intensität, die nach der Reflexion übrigbleibt;

                lg_light = mat->changeLightVertex(lg_light);

                // dieses geänderte wird zur Liste der LocalGeometries hinzugefügt:
                m_lightRayVertices.push_back(lg_light);

                // neue Methode im Material, die den reflektierten Strahl erzeugt:
                mat->getReflectedRay(lightray, lg_light, recursionDepthLight);

                return;
        }       // end of if;

} // Ende der neuen Methode raytraceLight() zum Verfolgen eines Lichtstrahles;
```

Der Ablauf in raytraceLight:
Zunächst wird der Lichtstrahl an die Intersect-Routine übergeben. Die Intersect-Methode liefert einen Zeiger auf ein LocalGeometry-Objekt zurück. Falls dieser NULL ist, dann hatte der Lichtstrahl kein Objekt getroffen. Das heißt, es wird nur weitergerechnet, wenn ein Objekt getroffen wurde. In diesem LocalGeometry-Objekt wird jetzt die Intensität des aktuellen Lichtstrahles zwischengespeichert. Weil es jetzt davon abhängt, zu wie viel Prozent das Material des Objektes, auf dem der Schnittpunkt liegt, diese Intensität reflektiert, wird über einen Zeiger auf das Material die Methode „changeLightVertex“ aufgerufen. An diese Methode wird das LocalGeometry-Objekt übergeben, dass jetzt die Intensität des Lichtstrahles „lightray“ enthält. In der Methode „changeLightVertex“ kann in der Materialklasse anhand von Membervariablen die Intensität aus dem LocalGeometry-Objekt entsprechend den Anteilen der Reflexion des Materials berechnet werden. Wenn ein Material zum Beispiel zu 50% spekular reflektiert, dann wird die Intensität auch um 50% verringert. Die neu berechnete Intensität wird wieder in das aktuelle LocalGeometry-Objekt geschrieben und geändert von der Methode zurückgegeben.
Das geänderte LocalGeometry-Objekt wird zu der Liste hinzugefügt, die die Stützstellen des Lichtstrahles aufsammelt.
Anschließend wird der Lichtstrahl an die „getReflectedRay()“-Methode übergeben, die den reflektierten Strahl nach dem Gesetz „Einfallswinkel gleich Ausfallswinkel“ berechnet. Hier kann eigentlich noch genauer differenziert werden nach Oberflächenbeschaffenheit bzw. nach lokalen Beleuchtungsmodellen. Der Einfachheit wegen wurde darauf jedoch bei der Implementierung verzichtet.
In der Methode getReflectedRay() steht dann für den reflektierten Lichtstrahl der Aufruf „raytraceLight“, so dass hier eine indirekte Rekursion implementiert wird.

Analog zu „raytraceLight“ gibt es die „raytraceBiPath()“-Methode in der Klasse Raytracer, die einen Augenstrahl als Parameter aufnimmt, an die Intersect-Methode übergibt und über einen Materialzeiger die Methode „getBiPathColor()“ aufruft, die ihrerseits den rekursiven Aufruf für einen sekundären Augenstrahl enthält. Weil „raytraceBiPath()“ fast identisch ist mit „raytraceLight()“, wird sie hier nicht näher dokumentiert.

3.3 Die Implementierung der Beleuchtungsberechnung

Jetzt wird die neue Materialklasse „BiPathMaterial“ angesprochen. In dieser Klasse stehen Methoden zum Setzen von Variablen und zum Aufruf der Werte von Variablen. Von größter Bedeutung in Bezug auf den bidirektionalen Path Tracer sind jedoch die Methoden zum Ändern der Intensität an einem LightVertex (Methode „changeLightVertex“), die Methode, die den reflektierten Lichtstrahl berechnet und den rekursiven Trace-Aufruf für den reflektierten Lichtstrahl enthält (Methode „getReflectedRay“) und die Methode „getBiPathColor()“, die die direkte Beleuchtung anhand der lokalen Beleuchtungsmodell von Phong, Schlick oder Blinn berechnet, den reflektierten Augenstrahl berechnet und die Interaktion zwischen den Stützstellen des Lichtstrahles und des Augenstrahles berechnet.

Die Formel von Henrik Wann Jensen, die im zweiten Kapitel angesprochen wurde, konnte leider nicht zur Beleuchtungssimulation verwendet werden. Die Berechnung der Beleuchtungsstärke E nach der Formel

$$E = \frac{I * \cos\alpha}{r^2}$$

war gut realisierbar, weil die Intensität I bekannt war und die Größen alpha und r aus ankommendem Strahl und Abstand zum Augenstützpunkt berechnet werden konnten. Probleme traten allerdings bei der Berechnung der BRDF als Quotient zwischen Beleuchtungsstärke und Leuchtdichte auf. Laut Definition hat die BRDF einen Wertebereich von [0…4 [. Der Wert für die BRDF kann also beliebig kleine oder beliebig große Werte annehmen. Testausgaben des BRDF-Wertes am Bildschirm bestätigten diese Annahme. Das größte Problem war, dass der BRDF-Wert sehr klein wurde für die Beleuchtungswerte, die durch die direkte Beleuchtung einer Oberfläche durch die Primärstrahlen entstand. Da die BRDF mit als Faktor bei der Berechnung der Ergebnisfarbe verwendet wurde, entstanden sehr dunkle Bildpunkte an den Stellen im Ergebnisbild, die eigentlich direkt beleuchtet werden sollten. Daher wurde der Ansatz von Jensen verworfen und nur die Klasse HallMaterial ein wenig verändert. Darin wird das Endergebnis für die Farbe des Bildes nach einer anderen Formel berechnet. Der Hauptgrund für die falschen Ergebniswerte bei Verwendung der Formel nach Jensen ist sicher die fehlende physikalische Grundlage des Ray Tracers, der erweitert wurde. Zum Beispiel wurde die Intensität einer Lichtquelle nicht als skalare Größe, sondern als Farbe behandelt und von ihr wurden immer Anteile in die Ergebnisfarbe eingerechnet.

Die wichtigste Änderung von HallMaterial für BiPathMaterial war der Ablauf bei der Beleuchtungsberechnung. In der Methode „getBiPathColor()“, die fast analog zu HallMaterial arbeitet, steht die Schleife, die alle Stützstellen der Lichtstrahlen einbezieht und durch Schattenfühler mit dem aktuellen Schnittpunkt des Augenstrahles verknüpft. Das ist die eigentliche Änderung von HallMaterial. Die Berechnung des reflektierten Strahles und der rekursive Aufruf der Methode „raytraceBiPath()“, die den Augenstrahl traversiert, sind nahezu identisch zu HallMaterial. Der folgende Ausschnitt zeigt daher nur die Schleife über alle Stützstellen des Lichtstrahles.

(aus bipathmaterial.cpp):

```
for (int index = 0; index < LGlight.size(); index++)
{
        // die zwei betrachteten Punkte sind IPointEye und der aktuelle
        // Lichtstützpunkt IPointLight;
        Point IPointLight = *(LGlight[index]->getIPoint());
        // die Intensität, die vom Punkt IPointLight ausgeht und bei IPointEye ankommt;
```

```
intens_Light_Eye = LGlight[index]->getLightRayIntensity();

// kann noch mit Abstand der beiden Punkte verrechnet werden;
// Abstandsberechnung zwischen IPointLight und IPointEye;
// !! Vektor von Augenstützpunkt zum Lichtstützpunkt zurück (xi -> yj)
einfallendInvert = IPointLight - IPointEye;

// Abstand ist dann die Länge dieses Vektors (= r für E(in) )
Real abstandLight_EyePoint = einfallendInvert.getLength();

// Optional die Entfernung von zwei Punkten bzgl. der Intensität mit einrechnen:
//intens_Light_Eye *= 1.0/abstandLight_EyePoint;

/// hier ist bereits der Einsatz eines Schattenfühlers wichtig:
/// wenn der ein Objekt schneidet, braucht gar nicht weitergerechnet werden
/// sondern es wird der nächste Lichtstrahlstützpunkt gewählt;
shadowRay.setDirection(einfallendInvert);


// WICHTIG : wenn der Schattenstrahl kein Objekt trifft,
// wird der Beitrag zum result berechnet;
if (scene->shadowIntersect(&shadowRay) == false)
{
        // Normalisieren des Vektors einfallendInvert, weil er gleich bei der Berechnung gebraucht
        // wird:
        einfallendInvert.normalize();

        rLightVec = (normalEye * (einfallendInvert.dotprod(normalEye) * 2)) - einfallendInvert;

        angle = (einfallendInvert.dotprod(normalEye));
        if (angle > 0.0)
        {
                // diffuser Anteil für den bidir. Path Tracer:
                *diffusPart += m_diffuseColor->getColor()
                                * intens_Light_Eye
                                * angle;
                // falls die Glanzzahl größer als null ist, tue:
                // (ist gebunden an die spekularen Anteile):
                if (m_glossiness > 1.0)
                {
                        // Unterscheidung in die lokalen Beleuchtungsmodelle:
                        // Unterschied besteht nur in der Berechnung des spekularen Anteils:
                        if (lightModel==0)
                        {
                                //Berechnung nach Phong:
                                rAngle = rLightVec.dotprod(viewVecEye);
                                if (rAngle > 0.0)
                                {
                                        // spekularer Term
                                        *specularPart += m_specularColor->getColor()
                                                * intens_Light_Eye
                                                * pow(rAngle, m_glossiness);
                                }
                        }       // end of if (light == 0);
```

(Jetzt folgt die Berechnung für specularPart nach den anderen lokalen Beleuchtungsmodellen.)

```
                        (…)
                }       // end of if (glossiness > 1.0)

        }       // end of if angle > 0.0;
}       // end of if (shadowIntersect == false);
}       // Ende der for-Schleife über alle LGlights;
```

Wichtig ist hier, dass durch die if-Abfrage

„if (scene->shadowIntersect(&shadowRay) == false)“

sichergestellt wird, dass nur dann der Beleuchtungsaustausch zwischen zwei Punkten stattfindet, wenn der zugehörige Schattenfühler kein Objekt der Szene trifft.

Nach dieser Schleife über alle Stützstellen der Lichtstrahlen ist die lokale Beleuchtung eines Schnittpunktes des Augenstrahles erfasst. Dann wird mit russischem Roulette entschieden, ob nur der diffus reflektierte, nur der spekular reflektierte oder nur der gebrochene Strahl weiterverfolgt wird. Je nachdem, welcher Strahl weiterverfolgt wird, wird die Ergebnisfarbe anders berechnet. Der folgende Auszug zeigt die Berechnung der Farbe, nach dem Berechnen des Sekundärstrahles durch Aussuchen mit russischem Roulette.

(aus BiPathMaterial, getBiPathColor()):

```
(…)
// die Farbe, die gleich mit der Ergebnisfarbe von raytraceBiPath:
Color* generalColor = new Color(0,0,0);

// hier kommt erst der rekursive Aufruf …
if (raytoTrace != NULL)
{
        delete generalColor;
        // …allerdings mit der Methode "raytraceBiPath(...)" aus dem Raytracer;
        generalColor = raytracer->raytraceBiPath(raytoTrace, --recDepthEye, LGlight);
}
// die Anteile des lokalen Beleuchtungsmodells „diffusPart“ und „specukarPart“ müssen beide zum result
// hinzugerechnet werden, weil sie beide zur lokalen Beleuchtung gehören:
*result = (*diffusPart) + (*specularPart);

// was an globaler Beleuchtung hinzukommt, wird erst jetzt berechnet:
// "wahl" entscheidet über die richtige Farbverrechnung; sie wurde in Abhängigkeit zur Wahl beim russischen
// Roulette gesetzt:
if (wahl == 3)      // dann muß der diffuse Anteil eingerechnet werden;
{
        *result = (*result)
                + ((*generalColor)*gewichtung)*m_diffusionColor->getColor();
}
else if (wahl == 1) // dann muß der speculare Anteil eingerechnet werden;
{
        *result = (*result)
                + (*generalColor)*m_reflectionColor->getColor();
}
else if (wahl == 2) // dann muß der transmittierende Anteil eingerechnet werden;
{
        *result = (*result)
                + m_refractionColor->getColor() * (*generalColor);
}

// Zeiger löschen:
delete diffusPart;
delete specularPart;

delete generalColor;
delete raytoTrace; // dann aber nicht in raytraceBiPath freigeben!

   // die Farbe auf [0,1] beschränken
   Real red = result->getR();
   Real gre = result->getG();
   Real blu = result->getB();
```

```
if (red < 0.0) result->setR(0.0);
if (gre < 0.0) result->setG(0.0);
if (blu < 0.0) result->setB(0.0);
if (red > 1.0) result->setR(1.0);
if (gre > 1.0) result->setG(1.0);
if (blu > 1.0) result->setB(1.0);

return result;                    ( Ende der Methode „getBiPathColor()“)
```

Weil bei der Berechnung von „result“ jetzt die Farbe aus „generalColor“, die die raytraceBiPath-Methode zurückgibt eingerechnet wird, wird hier die globale Beleuchtung berücksichtigt. Dann werden die Werte aus den RGB-Kanälen der Ergebnisfarbe auf den Bereich [0..1] beschränkt und durch return zurückgegeben.

Damit ist die Ergebnisfarbe bestimmt und wird in der Klasse Raytracer über einen Zeiger auf ein PPMImage als Pixelfarbe gesesetzt. Für jedes einzelne Pixel des Abtastgitters wird eine Pixelfarbe berechnet und das gesamte Bild im ppm-Format abgespeichert.

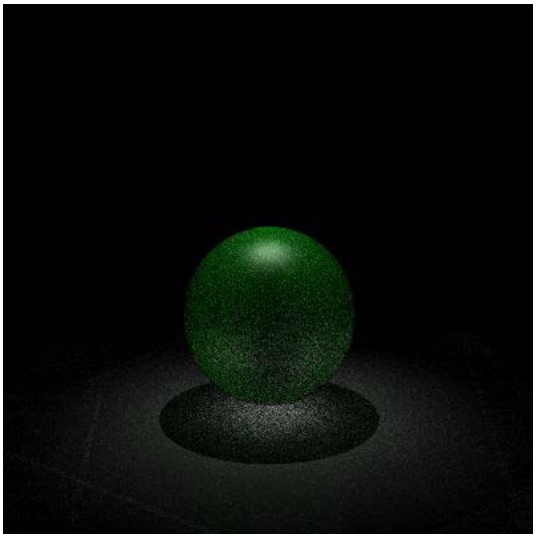
Abschließend zu diesem Kapitel möchte ich noch ein paar Ergebnisbilder zeigen.

3.4 Ergebnisbilder

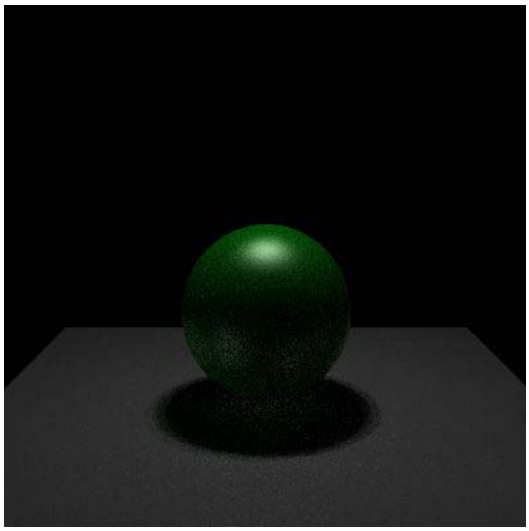
3.4.1 Die Kaustik

Am interessantesten ist natürlich die Frage: wie sieht die Kaustik aus, denn das war das eigentliche Ziel dieser Studienarbeit. Die folgenden Bilder zeigen eine grüne Glaskugel mit Brechnungsindex 1,6. Der Unterschied zwischen den Bildern besteht nur in der Wahl der Lichtquelle.

Die folgenden beiden Bilder wurden mit 20 Strahlen pro Pixel abgetastet.



(Abbildung 3.11)
eine Kaustik auf diffus reflektierendem Boden gerendert mit einem Spotlight

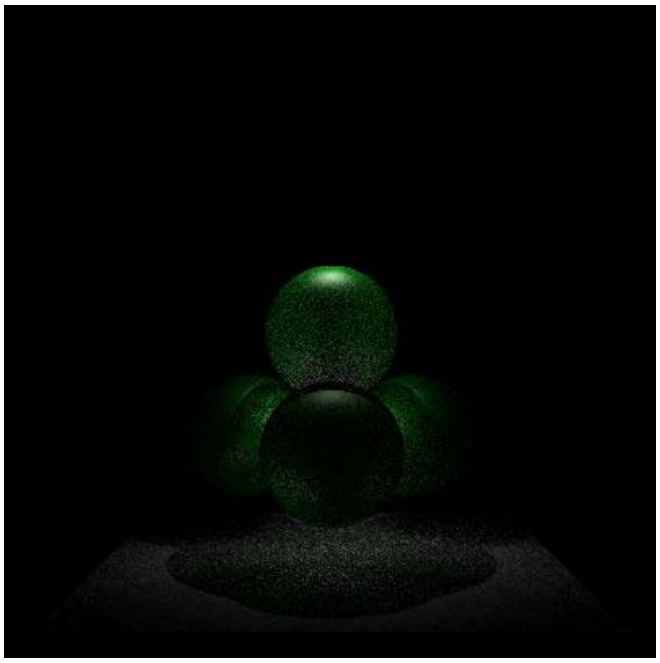


(Abbildung 3.12)
bei der Flächenlichtquelle ist die Kaustik fast gar nicht zu sehen; dazu muss die Zahl der Strahlen pro Pixel wesentlich erhöht werden

Auf den Einsatz einer Punktlichtquelle zum Rendern einer Kasutik wurde hier verzichtet, weil dabei noch weniger Strahlen durch die Kugel wandern und gebrochen werden.
Beim rechten Bild ist der Schattenrand der Kugel etwas verschwommen. Das liegt an der Flächenlichtquelle, die in der Lage ist, Halbschatten darzustellen. Im linken Bild sieht man die Ecken der Bodenfläche nicht mehr, weil sie nicht vom Lichtkegel des Spotlights erfasst werden.

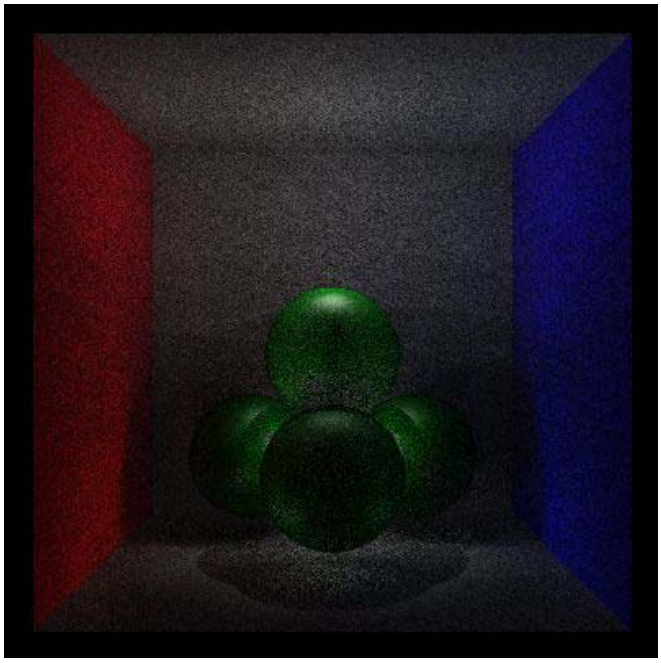
3.4.2 Glaskugeln

Interessant ist auch der Einsatz mehrerer Glaskugeln, um das „Zusammenspiel“ der Refraktionen zu untersuchen, wie das folgende Bild zeigt. (20 Strahlen pro Pixel)



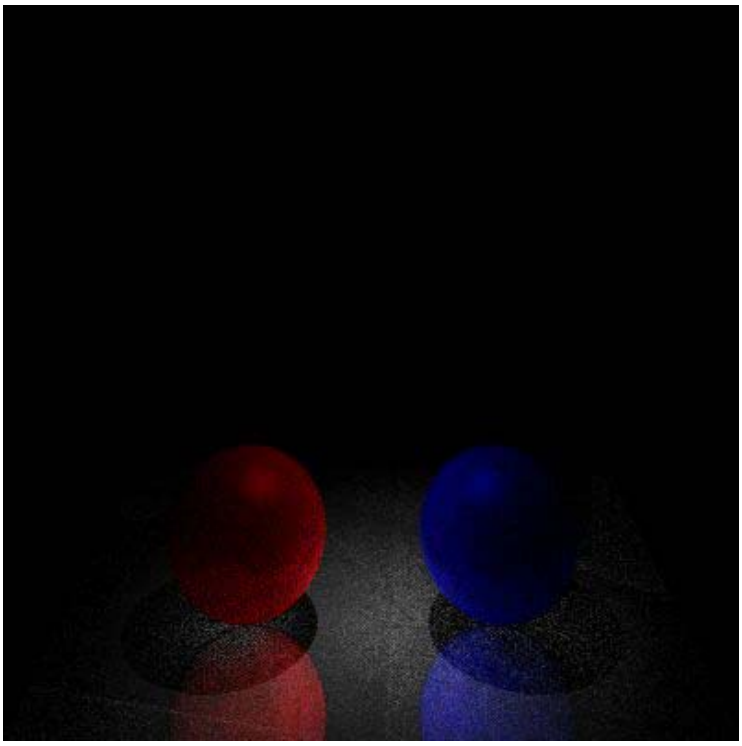
(Abbildung 3.13)

Der gleiche Szenenaufbau mit allen Wänden der Cornell-Box: (20 Strahlen pro Pixel)



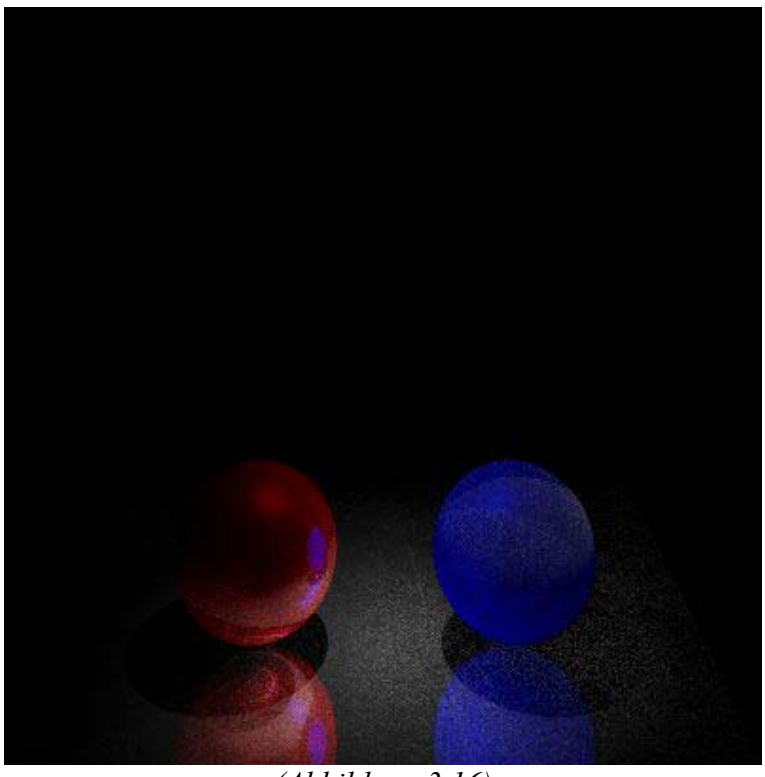
(Abbildung 3.14)

Ein spiegelnder Boden, darüber schweben eine rote und eine blaue Kugel, beide teilweise refraktierend. Beleuchtet wird die Szene mit einem Spotlight. (20 Strahlen pro Pixel)



(Abbildung 3.15)

Noch mal ein spiegelnder Boden, allerdings ist die rote Kugel jetzt zu 100% spiegelnd und die blaue Kugel zu 100% refraktierend. Die Kaustik, die am Boden bei der blauen Kugel eigentlich sichtbar sein sollte, ist nur sehr schwach sichtbar, weil die blaue Kugel am Rande des Lichtkegels liegt und hier die Lichtstrahlen eine eher geringe Intensität haben.

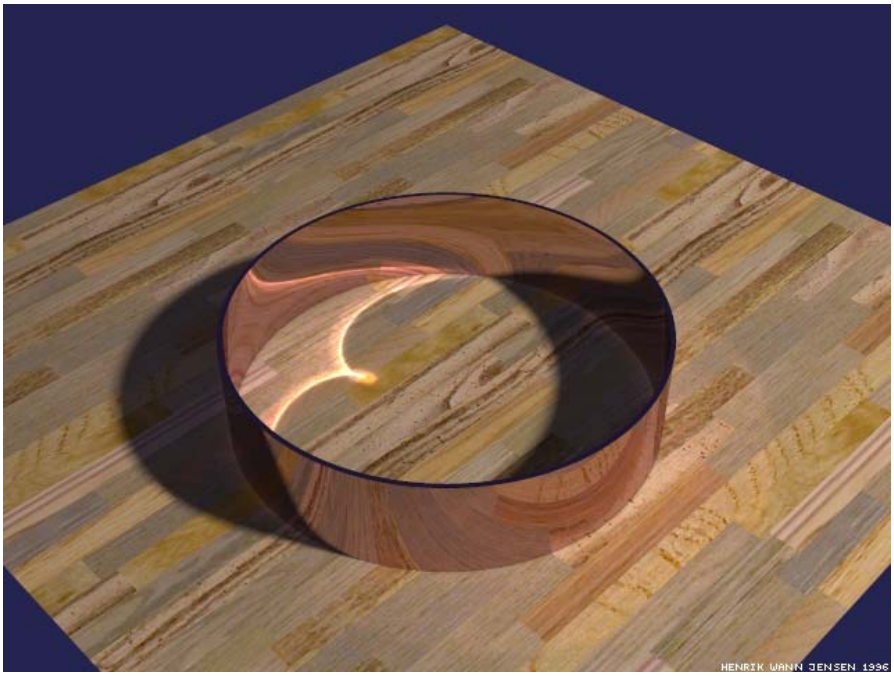


(Abbildung 3.16)

Kapitel 4
Fazit

Abschließend soll hier noch mal resümiert werden, ob das Ziel für die Studienarbeit erreicht wurde, was noch verbesserungswürdig ist und welche Möglichkeiten sich jetzt ergeben, weiter an diesem Thema „Bidirektionales Path Tracing“ zu arbeiten.

In dieser Studienarbeit wurde bidirektionales Path Tracing mathematisch betrachtet, es wurden eigene Ideen zur Implementierung eingebracht und am Schluss war es möglich, eine Kaustik zu rendern. Damit habe ich mein Ziel erreicht.
Ich möchte allerdings noch mal betonen, dass diese Implementierung keine physikalische Grundlage besitzt, sondern dass aufbauend auf einem vorhandenen Ray Tracer Ergänzungen und Berechnungen hinzugefügt wurden, die Ergebnisbilder berechnen, von denen ich sagen würde, dass sie photorealistisch gerenderte Szenen zeigen.
Was ich leider nicht mehr geschafft habe war die Erweiterung des Ray Tracers in Bezug auf den VRML-Loader, der dann Szenen in Form von VRML-Dateien für den bidirektionalen Path Tracer lädt und erfolgreich rendert. Das wäre aber wünschenswert gewesen, weil dann anhand eines Hohlspiegels festgestellt werden könnte, ob die Herzkurve, die dann entsteht, richtig berechnet wird (vgl. Abbildung 4.1). Der Einsatz einer Glaskugel aus dem Ray Tracer liefert zwar einen hellen Lichtfleck auf einer Fläche darunter, wegen der verrauschten Bilder lässt sich aber nicht genau bestimmen, ob die Kaustik im Detail mathematisch korrekt berechnet wurde.



(Abbildung 4.1)
Quelle: http://graphics.stanford.edu/~henrik/images/metalring.jpg
Das Bild zeigt einen Metallring, der von der von rechts oben beleuchtet wird.
An der Innenseite des Metallrings werden die Strahlen zu einer Herzkurve gebündelt.

Eine ebenso anspruchsvolle Aufgabe ist die Erweiterung der Berechnung von Refraktionsstrahlen für zwei Materialien, die beide einen von Luft verschiedenen Brechungsindex haben. Wenn zum Beispiel eine Glaskugel in einer Box liegt, die mit Wasser

gefüllt ist, dann wandert ein Lichtstrahl zuerst von Luft ins Wasser. Dort wird er zum ersten Mal gebrochen. Dann trifft der Lichtstrahl auf die Glaskugel und wird zum zweiten Mal gebrochen. Wenn er jetzt durch die Kugel hindurchwandert, trifft er wieder an die Grenze zwischen Glas und Wasser und wird wieder anders gebrochen. Das Problem ist, dass unter Umständen nicht immer bekannt ist, welche Brechungszahl das nächste Medium hat, in das der Lichtstrahl eintaucht. Eine Möglichkeit wäre, mit einem Stack zu arbeiten, um immer zu wissen, aus welchem Material der Lichtstrahl gerade kommt und in welches er wieder eintaucht.

Ein anderer interessanter Aspekt ist die Verbesserung des C++-Quellcodes in Bezug auf Schnelligkeit. Es wurde viel mit Zeiger-Strukturen gearbeitet, anstatt mehr Wert auf den Einsatz von Referenzen zu legen, die schnellere Verarbeitung garantieren als Zeigerstrukturen. Des Weiteren wurde noch nicht gestestet, ob die Anwendung der Beschleuniger BSP, BVH oder Uniform-Grid eine Verkürzung der Berechnungszeit ergeben.

Quellenverzeichnis

[1] The Rendering Equation, James Kajiya, Proceedings SIGGRAPH 1986, S. 143 – 150, 1986

[2] Bidirectional Path Tracing, Eric Lafortune und Yves Willems, in Proceedings 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics), pages 145--153, 1993.

[3] Bidirectional Estimators for Light Transport, Eric Veach and Leonidas Guibas, in Proceedings 5th Eurographics Worshop on Rendering, pages 147 -- 161, Darmstadt, 1994

[4] Reducing the number of shadow rays in bidirectional path tracing, Eric Lafortune and Yves Willems, in Proceedings of WSCG 95, (Pilsen, Czech Republic), pp. 384--392, Feb. 1995.

[5] Robust Monte Carlo Methods For Light Transport Simulation, Dissertation von Eric Veach, Stanford University, Dezember 1997, Kapitel 3, 8 und 10;
http://graphics.stanford.edu/courses/cs348b-03/papers/ (einzelne Kapitel)
http://graphics.stanford.edu/papers/veach_thesis/ (Link zur Dissertation selbst)
(Stand: 24.11.2003)

[6] Adaptive radiosity textures for bidirectional ray tracing, Paul S. Heckbert. Computer Graphics, 24(3):145--154, August 1990. ACM Siggraph 1990 Conference Proceedings.

[7] 3D-Computergrafik, von Alan Watt, 3. Auflage, Pearson Studium, 2002

[8] Graphische Datenverarbeitung 1, von José Encarnacão, Wolfgang Strasser, Reinhard Klein, 4.Auflage, R.Oldenbourg Verlag München Wien, 1996

[9] Realistic Image Synthesis Using Photon Mapping, Henrik Wann Jensen, AK Peters LTD., Massachusetts, 2001

[10] Taschenbuch der Physik, von Horst Kuchling, 14. Auflage, Fachbuchverlag Leipzig-Köln, 1994

[11] Lehrbuch der Experimentalphysik; Band 3; Optik, Bergmann-Schaefer, 5. Auflage, Verlag: Walter de Gruyter, 1972

[12] Vorlesung Photorealistische Computergraphik, Prof. Dr. Stefan Müller, WS 2002/2003, an der Universität Koblenz-Landau

[13] Optimally Combining Sampling Techniques for Monte Carlo Rendering, von Eric Veach und Leonidas Guibas, Proceedings of SIGGRAPH 1995, Seiten 419-428

Abbildungsverzeichnis