Brandenburgische Technische Universität Cottbus
Institut für Informatik
Lehrstuhl Praktische Informatik / Grafische Systeme

Masterarbeit



Spektrales Raytracing auf der GPU

Stephan Rogge

Datum der Abgabe
29.05.2009

1. Gutachter: Prof. Dr.rer.nat. Winfried Kurth
2. Gutachter: Prof. Dr.rer.nat. Claus Lewerentz
Betreuer: Dipl.-Inf. Reinhard Hemmerling

Aufgabenstellung

Für die Bildsynthese wurde das Raytracing-Verfahren mehrfach für die Grafikkarte umgesetzt. Jedoch wurden zur Berechnung lediglich die drei Farbkomponenten rot, grün und blau (RGB) berücksichtigt. Das ist ausreichend um ansprechende Ergebnisse durch die Bildsynthese zu erreichen, jedoch sind Phänomene wie Wellenüberlagerungen (Interferenzen), Lichtbrechung (Dispersion) und Lichtbeugung (Diffraktion) mit nur drei Komponenten nicht umsetzbar.
Der Grund dafür ist, dass konventionelle Raytracer Strahlen erzeugen und verfolgen und dabei keine Rücksicht auf Farbinformationen nehmen, die sie transportieren.
Die genannten Phänomene beruhen darauf, dass Lichtstrahlen in Abhängigkeit ihrer Wellenlänge an Objekten gebrochen oder reflektiert werden. Mit einem spektralen Raytracer sollen nun diese Effekte umgesetzt werden, so dass physikalisch korrekte Ergebnisse bei der Bildsynthese entstehen.
Die Arbeit soll die notwendigen theoretischen Grundlagen darstellen, aktuelle Forschungsarbeiten anderer Arbeitsgruppen auf verwandtem Gebiet berücksichtigen und ein Konzept zur software- und hardwaregestüzten Umsetzung (auf gegenwärtig gebräuchlichen und möglichen zukünftigen Grafikkarten) entwickeln. Als praxisorientierter Teil der Arbeit soll dieses Konzept teilweise in Form eines Softwarewerkzeuges für aktuelle, gängige Grafikkarten umgesetzt werden.
Dazu ist ein bereits existierender GPU-Raytracer zu erweitern und entsprechende Shader für die Berechnung der lokalen Beleuchtung zu programmieren. In Abhängigkeit von Parametern werden dann Strahlen mit unterschiedlichen Wellenlängen erzeugt und durch die Szene verfolgt. Die Verteilung der Strahlen über das Spektrum soll dabei nach dem Prinzip des so genannten Distributed Ray Tracing erfolgen.
Die zu entwickelnde Software soll als Plugin für die existierende 3D-Plattform GroIMP und optional für weitere 3D-Modeller verwendbar sein.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst habe. Weiterhin versichere ich, dass ich keine anderen als die angegebenen Quellen benutzt beziehungsweise die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Cottbus, den 29. Mai 2009

_____

Stephan Rogge

Danksagung

Diese Masterarbeit entstand am Lehrstuhl Praktische Informatik / Grafische Systeme der Brandenburgischen Technischen Universität in Cottbus. Bei der Erstellung dieser Arbeit wurde ich von einigen Menschen unterstützt, denen ich an dieser Stelle Danken möchte.
Vielen Dank an Herrn Professor Dr. Winfried Kurth für eine wirklich interessante und herausfordernde Aufgabenstellung sowie seinen Anmerkungen, die mir sehr behilflich bei der Bearbeitung dieser Arbeit waren. Auch ein Dankeschön dafür, dass er mir trotz besonderer Umstände die Bearbeitung meines Wunschthemas erst ermöglicht hat.
Einen weiteren Dank möchte ich auch Herrn Professor Dr. Claus Lewerentz für die Erstellung des Zweitgutachtens aussprechen. Seine Anmerkungen bei den Diskussionen während der Zwischenstandspräsentation halfen mir die Dinge vielleicht doch etwas besser erklären zu können und Ungereimtheiten zu beseitigen.
Reinhard Hemmerling möchte ich meinen besonderen Dank aussprechen. Jede noch so schwierige Fragen konnte ich ihm jederzeit stellen. Er stand mir an diesen Punkten äußerst Kompetent mit Rat zur Seite. Ohne Ihn wäre die Arbeit nicht das, was sie jetzt ist. Danke dafür.
Vielen Dank auch an Thomas Huwe. Während der Berarbeitungszeit haben wir uns ein Büro geteilt, wodurch wir fachliche Diskussionen bei Problem sehr einfach führen konnten. Die Atmosphäre im Büro war wirklich sehr angenehm.
Uwe Mannl möchte ich ebenfalls einen großen Dank aussprechen. Er half mir an vielen Stellen weiter, ging es um die Anbindungen zu GroIMP. Nicht selten stellte er an den richtigen Punkten die falschen Fragen, die mir dann aber zeigten, wo es noch hapert.
Meiner Freundin Dajana Strecker möchte ich ein ganz großes Dankeschön aussprechen für den Rückhalt in der arbeitsintensiven Zeit sowie das Verständnis, wenn es mal etwas später wurde.
Einen besonderen Dank möchte ich meinen Eltern, Christa und Thomas Rogge, für die Unterstützung während der Bearbeitungszeit und des gesamten Studiums, aussprechen. Ebenso unterstützen mich Ilona und Michael Dahley sowie Benji in der Zeit meines Studiums. Daher möchte ich auch Ihnen ganz herzlich danken.
Abschließend möchte ich Solvig Noack und Friedrich Gensicke für ihre zahlreichen Besuche danken. Die Besuche waren oft eine willkommene Gelegenheit kurz abzuschalten.

Inhaltsverzeichnis

1 Einleitung und Motivation **7**

2 Grundlagen **11**

2.1 Farbenlehre und Licht . . . 11
2.1.1 Tristimulus-Theorie – Das menschliche Farbsehen . . . 13
2.1.1.1 Der RGB-Farbraum . . . 14
2.1.1.2 Der XYZ-Farbraum . . . 15
2.1.2 Spektrum zu XYZ Konvertierung . . . 16
2.1.3 XYZ zu RGB Konvertierung . . . 19
2.1.4 RGB zu Spektrum Konvertierung . . . 20
2.1.5 Natürliche Phänomene des Lichts . . . 20
2.1.5.1 Reflexion . . . 21
2.1.5.2 Transmission . . . 21
2.1.5.3 Fresnel . . . 22
2.1.5.4 Dispersion . . . 24
2.2 Grundlagen des Raytracing . . . 26
2.2.1 Rekursives Raytracing . . . 27
2.2.2 Path Tracing und Monte Carlo . . . 29
2.2.2.1 Rendergleichung . . . 29
2.2.2.2 Monte Carlo Integration . . . 30
2.2.2.3 Zufallszahlen . . . 31
2.2.2.4 Path Tracing . . . 32
2.2.3 Reflexionsmodelle . . . 33
2.2.3.1 Shading-Koordinatensystem . . . 34
2.2.3.2 Lambert'sche Reflexion . . . 35
2.2.3.3 Spekulare Reflexion . . . 36
2.2.3.4 Spekulare Transmission . . . 37
2.2.3.5 Microfacetten-Modelle . . . 38
2.2.3.6 Zusammenfassung . . . 41
2.3 Sunshine – Ein GPU Raytracer . . . 41
2.3.1 Zugrundeliegende Prinzipien für das Arbeiten auf der GPU . . . 42
2.3.1.1 Texturen . . . 43
2.3.1.2 Frame-Buffer-Objekt . . . 44
2.3.1.3 GLSL Programm - Einschränkungen . . . 46
2.3.1.4 GLSL Programm - Erzeugung und Ausführung . . . 46
2.3.2 Bestandteile . . . 50

2.3.2.1 Grundlegendes Konzept der Pipeline von Sunshine 50
2.3.2.2 Erweiterung der Pipeline 52
2.3.3 Datenstruktur in FBO-Textur 53
2.3.4 Ablaufsteuerung 54
2.3.5 Repräsentation der Szene-Daten 56

3 Konzept zur Umsetzung 58
3.1 Konzept der Ablaufsteuerung 58
3.1.1 Modifizierung 60
3.1.2 Vorbereitung 61
3.1.3 Renderprozess 62
3.1.4 Finalisierung 63
3.2 Organisation von Materialdaten 63
3.2.1 Das Grundprinzip 64
3.2.1.1 Feldgrößen der Materialtypen 64
3.2.1.2 Adressierung von Einträgen 65
3.2.1.3 Blockzerlegung 66
3.2.2 Behandlung von RGB-Farben 68
3.2.3 Behandlung von Bild-Texturen 69

4 Implementierung 71
4.1 Ablaufsteuerung 71
4.2 Darstellung der Materialien in der GroIMP 3D-Ansicht 78
4.3 Organisation von Materialdaten 80
4.4 Shader-Programme 85
4.4.1 Auslesen der Materialdaten 88
4.4.2 BSDF-Modelle 90
4.4.2.1 Lambertian 92
4.4.2.2 Glas 93
4.4.2.3 Microfacetten 95

5 Ergebnis und Bewertung 98
5.1 Renderer-Auswahl für Vergleich 98
5.2 Qualitäts- und Geschwindigkeitsvergleich 98
5.2.1 Referenzbilder 99
5.2.2 Vergleich mit spektralen CPU Raytracern 100
5.2.3 Vergleich mit anderen GPU 103
5.3 Dispersive Effekte 107

6 Zusammenfassung und Ausblick 109
6.1 Zusammenfassung 109
6.2 Ausblick 110

1 Einleitung und Motivation

Die Bildsynthese hat als oberstes Ziel die Erzeugung von computergenerierten Bildern, die von realen Bildern kaum bis gar nicht zu unterscheiden sind. Um dieses Ziel zu erreichen, wurden in der Vergangenheit viele Verfahren und Theorien vorgestellt. Das Raytracing ist eines dieser Verfahren, welches auch am häufigsten zum Einsatz kommt. Es wird dabei der Weg des Lichtes durch die Szene zur Kamera verfolgt. Je nach Ansatz kann dies in beide Richtungen geschehen.
Bei vielen Ansätzen wird das Verhalten der Lichtstrahlen nachempfunden, was die Abbildung vieler Effekte, beobachtbar in der Natur, ermöglicht und somit zu realistischen Ergebnissen führen kann. Da sich Raytracing bereits in der Vergangenheit als äußerst robust erwiesen hat, wurden viele Verfahren, die im Kern auf diesem Ansatz basieren, entwickelt. Diese zeichneten sich dadurch aus, ob beziehungsweise wie gut sie natürliche Effekte nachempfinden konnten. Die wichtigsten Effekte sind Schatten, Reflexion sowie Transmission von Lichtstrahlen und die gegenseitige Verdeckung von Objekten im Raum.
Eines der ersten Verfahren, welches die genannten Effekte umsetzen konnte, war Whitted's Rekursiver Raytracer [32]. Dieses Verfahren ist nach wie vor Grundlage vieler Weiterentwicklungen. Es berücksichtigte bereits die globale Illumination, da Objekte mit spiegelnder oder transparenter Oberfläche dargestellt werden konnten. Eine Schwachstelle ist jedoch, dass hier nur harte Schatten sowie ideal reflektierende oder ideal transmittierende Oberflächen unterstützt wurden.
Eine Erweiterung zu Whitted's Ansatz ist Cook's et al. *Distributed Raytracer*[4]. Bei jedem gefundenen Auftreffpunkt wurden mehrere Strahlen erzeugt. Schatten konnten nun weicher und somit natürlicher dargestellt werden. Ebenso war es nun möglich, diffuse Reflexionen an Oberflächen zu erzeugen. Jedoch mussten mit diesem Verfahren wesentlich mehr Strahlen verfolgt werden, um die gewünschten Effekte zu erzielen. Kajiya's *Pathtracer* [9] reduzierte den Berechnungsaufwand, in dem immer nur ein Strahl weiterverfolgt wurde. Kajiya formulierte die sogenannte Rendergleichung[1], die bei der Bildsynthese einer Szene zu lösen ist. Er machte die Beobachtung, dass viele der erzeugten Lichtpfade mit zunehmender Länge immer weniger zur Beleuchtung des Bildpunktes beitragen und somit überflüssig sind. Stattdessen sei es besser, Lichtpfade ausgehend von der Kamera neu zu erstellen. Die aufgestellte Rendergleichung wird aufgrund dieser Beobachtung nicht analytisch, sondern mit Hilfe von zufällig ausgewählten Stichproben numerisch gelöst. Dieses Verfahren nennt sich *Monte-Carlo*[9].
Allen Verfahren gemein ist, dass Schnittpunkte zwischen Strahlen und Objekten der Szenerie gefunden werden müssen. Speziell beim Pathtracing wird ein Strahl pro Bildpunkt verfolgt. Entlang der Schnittpunkte entsteht somit ein Pfad. Im Gegensatz zu Whitted

[1] Siehe Abschnitt 2.2

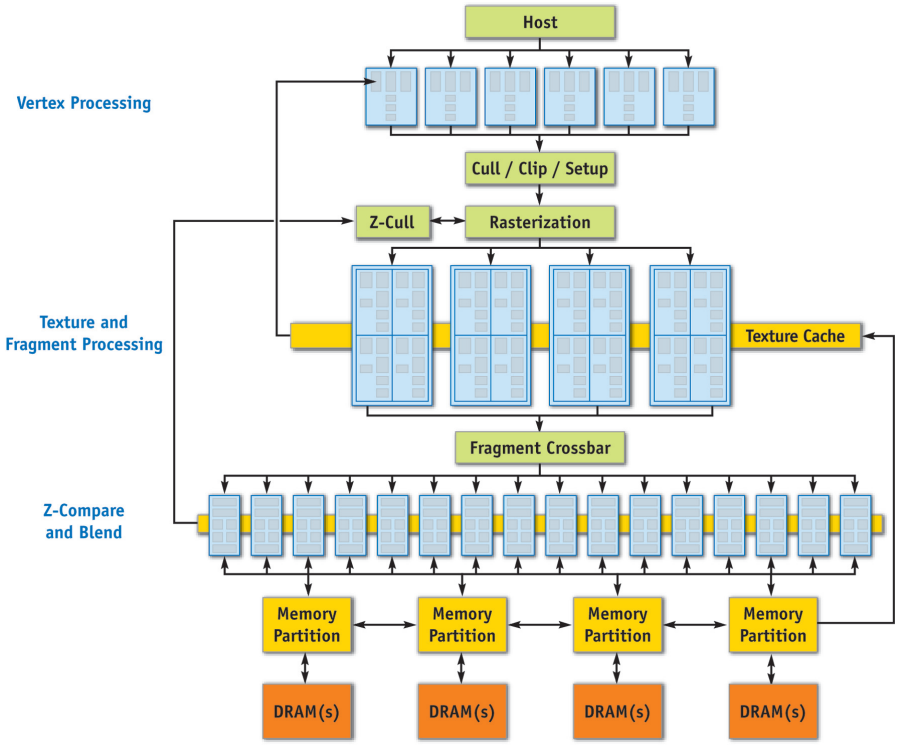


Abbildung 1.1: Aufbau der Pipeline einer GPU der GeForce 6er Serie des Grafikkartenherstellers NVIDIA. Entnommen aus Kilgariff et al. [10].

wird an den Schnittpunkten nur ein Strahl – transmittierend oder reflektierend – weiterverfolgt. Da für das Lösen der Rendergleichung die Monte-Carlo-Methode genutzt wird, kommt es zu Rauschen im Bild, was durch Berechnen mehrerer Pfade pro Bildpunkt reduziert werden kann[2]. Dies bedeutet jedoch auch einen erheblichen Mehraufwand bei der Berechnung.
Alle zu verfolgenden Strahlen können unabhängig voneinander betrachtet werden. Dies ist ein wichtiger Punkt, da somit das Verfahren effizient parallelisierbar wird. Der Berechnungsaufwand für die Strahlenverfolgung ist enorm: Jeder Strahl muss mit allen Objekten[3] getestet und lokale Beleuchtungsmodelle berechnet werden. Können diese Aufgaben parallel abgearbeitet werden, ist das Ergebnis schneller fertiggestellt.
Der Grund für das Verlagern des Raytracings auf die Grafikkarte (GPU) ist begründet in ihrer Architektur. Abbildung 1.1 zeigt die parallel angelegten Elemente der GPU-Pipeline. Die hochgradig parallele Architektur ermöglicht es, wesentlich mehr Strahlen gleichzeitig zu verfolgen, verglichen mit einer konventionellen CPU-basierten Implementierung. Deshalb wurden in der Vergangenheit einige Konzepte zum Raytracing auf der GPU[20, 24, 2] vorgestellt, die diesen Geschwindigkeitsvorteil ausnutzten. Hinzu kommt,

[2]Siehe Abschnitt 2.2.2, Supersampling.
[3]Dies gilt nur, sofern keine Beschleunigungsstruktur eingesetzt wird.

dass die Rechenleistung der GPU in den letzten Jahren weitaus stärker zugenommen hat als die der CPU, wie in den Abbildungen 1.2 und 1.3 zu sehen ist.

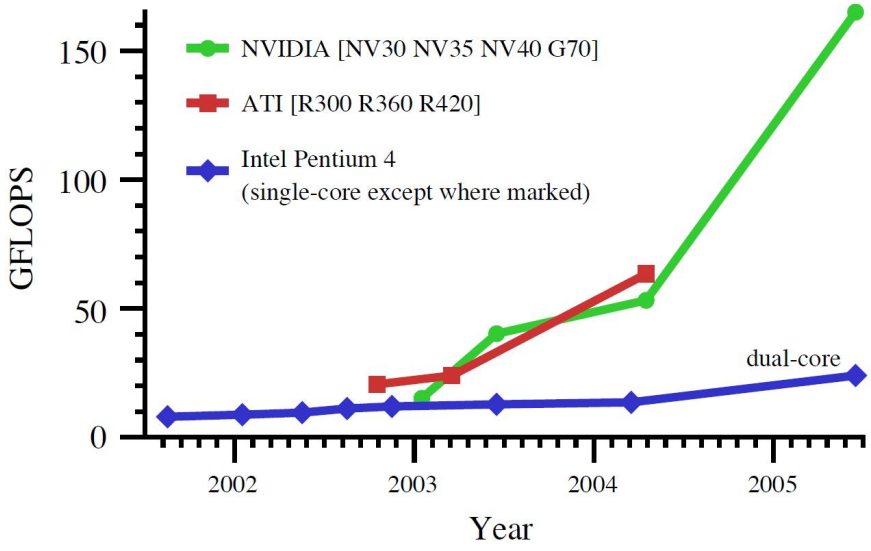


Abbildung 1.2: Vergleich der Entwicklung der Leistungsfähigkeit zwischen Intels CPUs und ATIs beziehungsweise NVIDIAs GPUs. Entnommen aus Owens et al. [15].

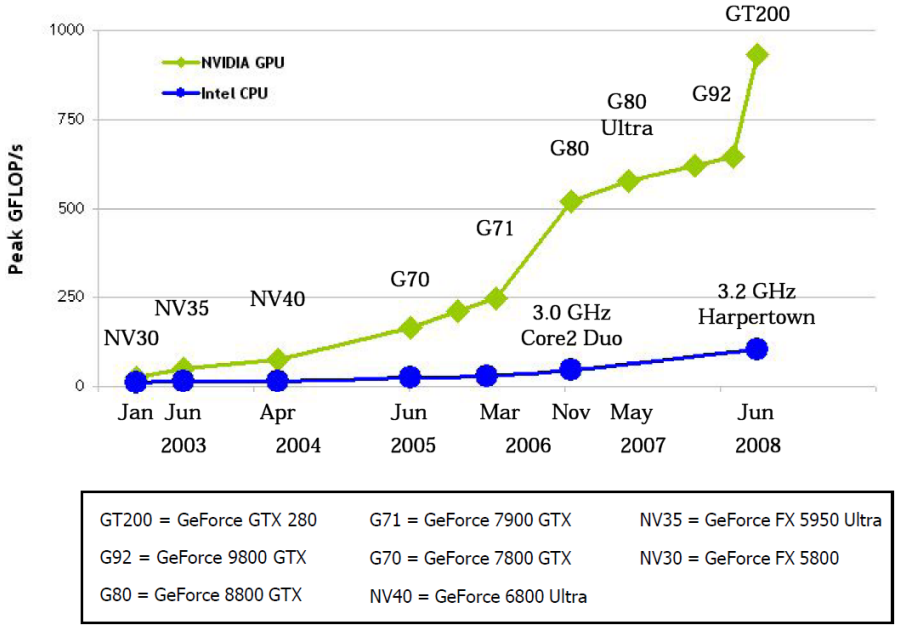


Abbildung 1.3: Vergleich der Entwicklung aktueller Grafikkarten. Entnommen aus NVIDIA CUDA Programming Guide 2.0[13].

Die Rendergleichung kann auf zwei Arten gelöst werden: *farb-* und *spektralbasiert*. Bei der farbbasierten Variante werden zumeist die drei Grundfarben rot, grün und blau genutzt, um die lokalen Beleuchtungsmodelle zu berechnen. Dies reicht aus, um einen Großteil von Effekten zu erzielen, hat jedoch Einschränkungen gegenüber der spektralbasierten Variante. Hier wird das Licht beziehungsweise die Farbinformationen nicht auf drei Grundfarben reduziert, sondern durch ein Spektrum repräsentiert. Spektren sind in Wellenlängen-Bereiche eingeteilt: zu jeder Wellenlänge gibt es einen Wert, die Bestrahlungsstärke. Diese wird in der Literatur auch als *Irradiance* bezeichnet. Sie beschreibt die Leistung der elektromagnetischen Energie, die auf eine Oberfläche einfällt, und setzt diese ins Verhältnis zur Größe der bestrahlten Fläche. Mit Spektren ist es möglich, wellenlängenabhängige Effekte zu erzeugen. Der bekannteste Effekt ist die Spektral-Zerlegung von weißem Licht durch einen Prisma, wodurch Regenbogenfarben zu sehen sind. Das farbbasierte Rendern kann als Sonderfall des spektralen Renderns betrachtet werden: Alle Berechnungen beschränken sich in diesem Fall auf die Wellenlängenbereiche für rot, grün und blau.
Das Rendern mit Spektren bedeutet jedoch auch einen erheblichen Mehraufwand bei der Berechnung. Für die in der Industrie verwendete Bildsynthese ist eben jener Mehraufwand nicht wünschenswert. Dies und die Tatsache, dass ein RGB-basierter Renderer bereits einen Großteil von Effekten darstellen kann und zufriedenstellende Ergebnisse liefert, lässt spektralbasierte Verfahren eher in den Hintergrund rücken.
Der bereits mehrfach nachgewiesene Geschwindigkeitsvorteil des GPU-basierten Raytracings gegenüber der CPU-basierten Variante macht den spektralen Ansatz wiederum interessant.
Arbeiten zu spektral-basiertem Rendern auf der GPU wurden vorgestellt[5, 23]. Jedoch verfolgten diese Ansätze, im Gegensatz zu dieser Arbeit, das Ziel, dass Bildsynthese möglichst in Echtzeit geschieht. Außerdem wurde auf einen Raytracing-Ansatz verzichtet. Das Ziel dieser Arbeit ist, basierend auf dem Raytracing-Ansatz möglichst realistische Bilder zu synthetisieren, wobei es nicht auf eine Erstellung in Echtzeit ankommt.
Aufbauend auf den Ergebnissen von Huwe [8] sollen Ideen und Konzepte zum spektralen Raytracing auf der GPU gegeben und in der interaktiven Modellierplattform GroIMP[11] implementiert werden.

2 Grundlagen

2.1 Farbenlehre und Licht

Für das spektrale Raytracing sind im Vorfeld einige Betrachtungen zur Farbenlehre beziehungsweise dem Farbsehen zu machen. Im Wesentlichen werden hier nun einige für den Verlauf dieser Arbeit wichtige Aspekte zusammengefasst. Für genauere Informationen sei hier auf das Buch *Computer Graphics: Principles and Practice in C*[6, Seite 563 ff.] verwiesen[1].
Im Bereich der Computergrafik ist die *Farbe* ein wichtiger Bestandteil. In den Gebieten der Physik, Physiologie, Psychologie und Kunst wurden im vergangenen Jahrzehnt viele Erfahrungen gesammelt[6, Seite 563]. In diesem Abschnitt soll kurz auf den Stand der Dinge eingegangen werden. Dabei beziehen wir uns vor allem auf die beiden erstgenannten Gebiete der Physik und Physiologie. Nach Foley et al. [6, Seite 563] wird die Farbe eines Objektes nicht nur durch das Objekt selbst bestimmt, sondern durch seine Oberfläche, die Lichtquelle, die es beleuchtet, die Umgebung sowie durch das menschliche Sehsystem.
Das Licht ist Teil des elektromagnetischen Spektrums: Energie wird auf periodisch schwingenden Wellen transportiert, wobei sich diese durch ihre *Frequenz* unterscheiden können. Diese Unterschiede können zu verschiedenen Farbeindrücken führen. Licht lässt sich als eine Funktion der Frequenz f in Abhängigkeit der Wellenlänge λ darstellen,

$$f = \frac{c}{\lambda}, \tag{2.1}$$

wobei die Konstante c der Lichtgeschwindigkeit im Vakuum von ca. 300.000 $km \cdot s^{-1}$ entspricht. Im Zusammenhang von *Licht* und *Farbe* spricht man jedoch seltener von der *Frequenz*, als viel mehr von der *Wellenlänge*. Der für Menschen sichtbare Bereich des Spektrums liegt laut Foley et al. zwischen 400nm (blau) und 700nm (rot) [6, Seite 575]. In der Literatur schwanken die Angaben der Grenzen jedoch. Lichtstrahlen mit nur einer klar dominierenden Wellenlänge werden auch als *monochromatisches Licht* bezeichnet.
Die für den Menschen wahrnehmbaren Farbeindrücke des ankommenden Lichtes entstehen nicht nur durch monochromatisches Licht, sondern viel mehr durch eine Mischung von Lichtstrahlen verschiedener Wellenlänge. Die einzelnen Lichtstrahlen werden durch eine spektrale Energieverteilung repräsentiert – des Weiteren als *Spectral Power Distribution*, kurz SPD, bezeichnet. Zu jeder Wellenlänge ist ein Wert für die Bestrahlungsstärke

[1] Komprimierte Form von Shoaff:
http://www.cs.fit.edu/~wds/classes/graphics/Illuminate/illuminate.pdf (Letzter Zugriff: 26.02.2009)

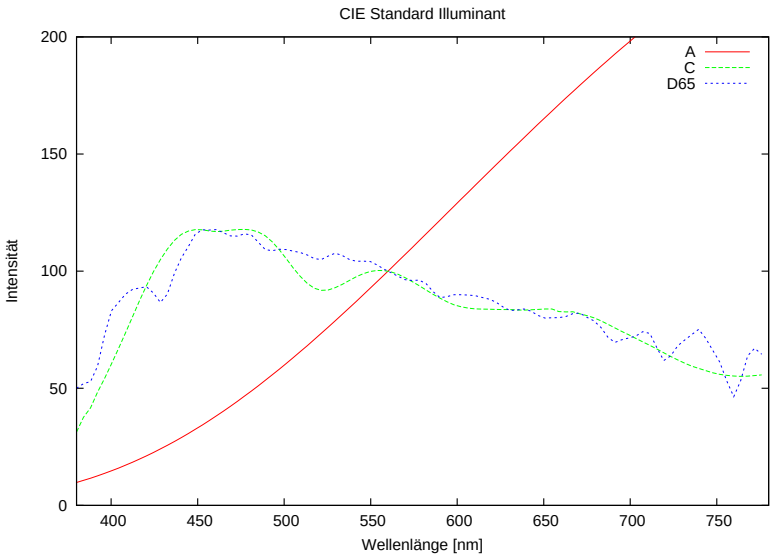


Abbildung 2.1: Beispiel einiger SPDs von standardisierten Lichtquellen.

mit der Einheit W/m^2 angegeben. Einige SPDs bezüglich standardisierter Lichtquellen sind in Abbildung 2.1 zu sehen.
Zwar kann ein gegebenes Spektrum eindeutig einem Farbeindruck zugeordnet werden, jedoch kann nicht von einem gegebenen Farbeindruck auf das zugrundliegende Spektrum zurückgeschlossen werden. Wenn verschiedene Spektren einen gleichen Farbeindruck erzeugen, nennt man diese *Metamere*. Dieser Umstand, dass nicht eindeutig von Farben auf die zu grundeliegende spektrale Verteilung zurückgeschlossen werden kann, wird in Abschnitt 2.1.4 gesondert behandelt.
Für die Beschreibung der Wahrnehmung von *Licht* werden drei Größen genutzt: Farbton (*Hue*), Sättigung (*Saturation*) und Helligkeit (*Lightness*)[6, Seite 574]. Der Farbton lässt uns Farben wie rot, grün oder blau auseinanderhalten. Die Sättigung gibt an, wie wenig Grau in der Farbe, bei gleicher Intensität, enthalten ist. Das heißt, einer gesättigten Farbe – zum Beispiel Rot – wird Weiß hinzugemischt. Bei gleichbleibender Intensität nimmt somit die Sättigung der Farbe ab. Die Helligkeit bringt die Intensität des ankommenden Lichtstrahls zum Ausdruck, wobei die Helligkeit unterschieden wird in:

- reflektierte Helligkeit (*Lightness*): an Oberflächen reflektiertes Licht
- selbst-leuchtende Helligkeit (*Brightness*): Objekte, die selbst Licht ausstrahlen.

2.1.1 Tristimulus-Theorie – Das menschliche Farbsehen

Das menschliche Auge besitzt in der Retina drei Typen von Farbsensoren, genannt Zäpfchen [6, Seite 576]. Die drei Zäpfchentypen sind auf rote, grüne oder blaue Lichtanteile sensibilisiert. In Abbildung 2.2 sind die Verläufe der Ansprechverhalten jeweils für die drei Zäpfchentypen dargestellt. So ist zu erkennen, dass das Auge auf blaues Licht wesentlich weniger stark anspricht als auf rotes oder grünes Licht. Daher können unterschiedliche Farbeindrücke mit konstanter Intensität unterschiedlich hell wahrgenommen werden.
In den dargestellten Verläufen sind die wellenlängenabhängigen Maxima der Ansprechverhalten zu entnehmen: Blaues Licht hat sein Maximum bei 440nm, grünes Licht bei circa 545nm und rotes Licht bei 580nm, was eigentlich eher einem gelben Farbeindruck entspricht. Das Maximum des blauen Lichtes ist im Vergleich zu grün und rot wesentlich geringer. Bei konstanter Intensität, aber sich ändernder Wellenlänge variiert die gesamte Sensibilität des Auges. Dies kann in einer Kurve dargestellt werden, welche auch als *relative Helligkeitsfunktion* bezeichnet wird. Sie erstreckt sich über das sichtbare Spektrum und hat bei 550nm ihr Maximum erreicht. Das bedeutet, dass das Auge am empfindlichsten bei grünerem Licht reagiert.

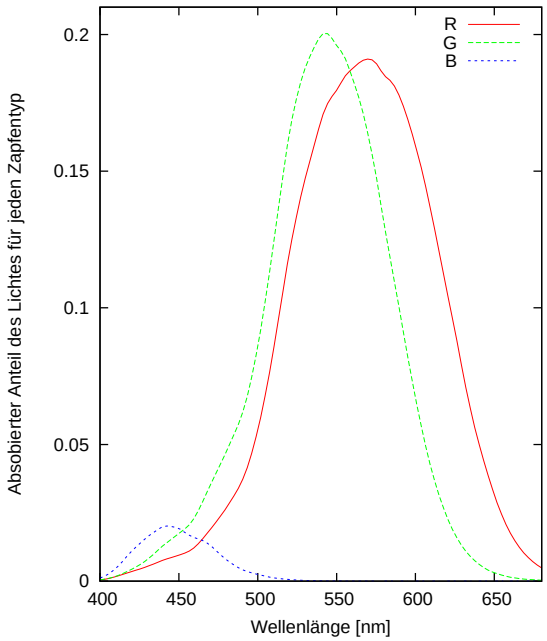


Abbildung 2.2: Verlauf des spektralen Ansprechverhaltens für jeden Zapfentyp. Adaptiert von Foley et al. [6].

Experimentell erwiesen ist, dass ein Test-Farbeindruck $C^t(\lambda)$ durch Mischen von vorher festgelegten Primär-Farbreizen komplett nachgebildet werden kann[17, Seite 388]. Mit den Primär-Farbreizen $C^r(\lambda)$, $C^g(\lambda)$ und $C^b(\lambda)$ lautet die Gleichung

$$C^t(\lambda) = R\,C^r(\lambda) + G\,C^g(\lambda) + B\,C^b(\lambda). \tag{2.2}$$

Die relativen Größen R, G und B sind die jeweiligen Intensitäten, mit denen die drei Primärfarben mit einander vermischt werden, um den Farbeindruck $C^t(\lambda)$ zu erzeugen.

2.1.1.1 Der RGB-Farbraum

Die *Commission Internationale de l'Éclairage* (CIE) entwickelte 1931 auf Grundlage von gemittelten Messdaten mehrerer Probanden das *CIE-Normalvalenzensystem*. Die gemittelten Messdaten wurden zusammengefasst und beschreiben den sogenannten *Normalbeobachter*.
In einem Experiment wurden drei monochromatische Lichtquellen definiert, die die drei Primärvalenzen in Gleichung 2.2 repräsentieren. Die drei Primär-Farbreize wurden demnach auf $C^r(\lambda)$ ($\lambda = 700nm$), $C^g(\lambda)$ ($\lambda = 546,1nm$) und $C^b(\lambda)$ ($\lambda = 435,8nm$) festgelegt[17, Seite 388].

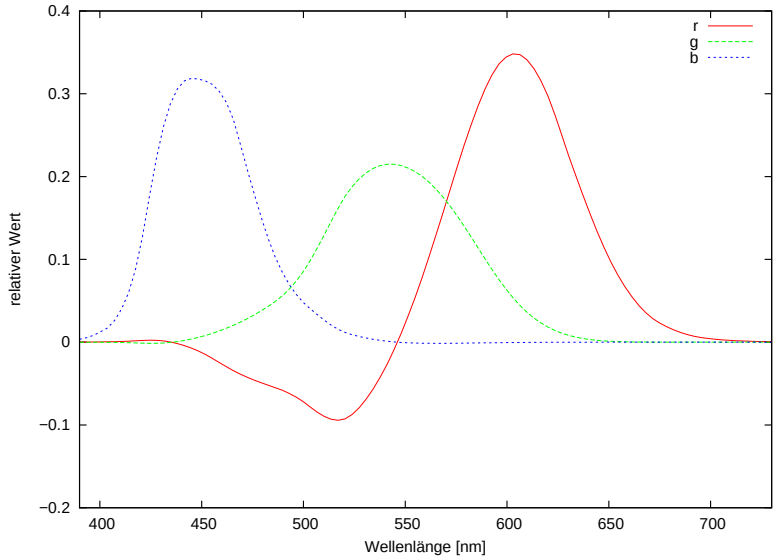


Abbildung 2.3: Darstellung der Farbanpassungsfunktionen nach CIE genormt für die drei Primärvalenzen R, G und B.

Ein Proband stellte mittels Regler die Intensitäten der Lichtquellen ein. Dieser sollte einen vorgegebenen Farbreiz durch Einstellen der Lichtquellen nachempfinden. Dabei

entstanden zu den drei Primär-Farbreizen Kurvenverläufe, die die relativen Größen R, G und B in Gleichung 2.2 beschreiben. Die gemittelten Kurven des Normalbeobachters sind in Abbildung 2.3 als die Funktionen r, g und b dargestellt. Diese werden des Weiteren als CIE-RGB Farbanpassungsfunktionen mit der Notation $\overline{r}$, $\overline{g}$ und $\overline{b}$ bezeichnet.
Die negativen Werte in den Kurven-Verläufen der Funktionen resultieren daher, dass nicht alle Farbwerte des RGB-Farbraums durch adaptives Mischen der drei Primärvalenzen erzeugt werden können. In diesem Fall wird eine der Primärvalenzen zu dem gegebenen Farbreiz hinzuaddiert. Dies entspricht quasi einer Subtraktion dieser Primärvalenz. Dieses Hinzumischen zur Primärvalenz drückt sich in Form negativer Werte in Abbildung 2.3 aus.
Werden die Primärvalenzen normiert, so dass $R + G + B = 1$ gilt, und projiziert man diese auf eine Fläche, welche durch die Achsen für rot und grün festgelegt wird, kann der Farbraum auch als ein Dreieck in der Ebene dargestellt werden. Mit der genannten Berechnungsvorschrift lassen sich nun alle Farben des Farbraumes mischen.

2.1.1.2 Der XYZ-Farbraum

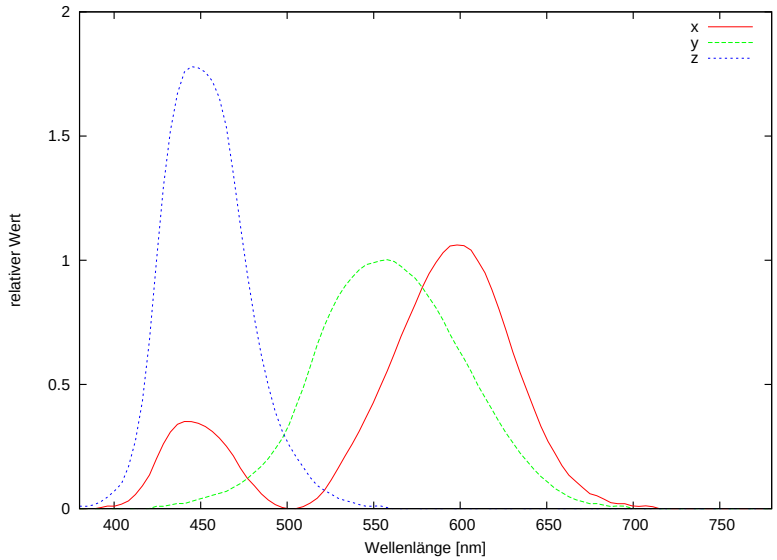


Abbildung 2.4: Darstellung der Farbanpassungsfunktionen nach CIE genormt für die drei Primärvalenzen X, Y und Z.

Die in Abbildung 2.3 dargestellten Funktionen nehmen stellenweise negative Werte an. Da dies ist in der Computergrafik nicht erwünscht ist[6, Seite 578 ff.], wurde der CIE XYZ-Farbraum eingeführt. Die drei Primärfarben rot, grün und blau wurden durch X, Y und Z ersetzt. Im Experiment wurden die monochromatischen Lichtquellen durch

Lichtquellen mit Mischfarben ersetzt. Diese wurden so ausgewählt, dass in den Funktionsverläufen keine negativen Werte mehr auftraten. Die daraus resultierenden Funktionsverläufe sind in Abbildung 2.4 dargestellt.
Die in Abbildung 2.4 dargestellten Farbanpassungsfunktionen sind lineare Kombinationen aus den in Abbildung 2.3 dargestellten Farbanpassungsfunktionen. Somit kann eine Farbe aus dem XYZ-Farbraum mittels linearer Transformation in den RGB-Farbraum überführt werden und umgekehrt. Die Gleichung einer Transformation von RGB nach XYZ lautet:

$$\begin{bmatrix} \overline{x}(\lambda) \\ \overline{y}(\lambda) \\ \overline{z}(\lambda) \end{bmatrix} = \begin{bmatrix} 0,49000 & 0,31000 & 0,20000 \\ 0,17697 & 0,81240 & 0,01063 \\ 0,00000 & 0,01000 & 0,99000 \end{bmatrix} \begin{bmatrix} \overline{r}(\lambda) \\ \overline{g}(\lambda) \\ \overline{b}(\lambda) \end{bmatrix}. \tag{2.3}$$

Werden X,Y und Z normiert, so dass $x + y + z = 1$ gilt, können somit alle sichtbaren Farben durch das sogenannte „Schuhsohlen-Diagramm“ dargestellt werden. Abbildung 2.5 zeigt die Darstellung der CIE-Normfarbtafel in der Ebene.
In der Abbildung ist der RGB-Farbraum als Dreieck zu erkennen. Nun wird auch deutlich, dass der RGB-Farbraum wesentlich kleiner als der XYZ-Farbraum ist. Aus diesem Grund wird das XYZ-Farbsystem für die Endverarbeitung dieses Raytracers genutzt. Neben der größeren Abdeckung des Farbbereichs, ohne negative Werte zu nutzen, ist ein weiterer Grund die Möglichkeit der leichteren Umwandlung in andere Farbsysteme.
In dieser Arbeit werden SPDs zunächst in den XYZ- und anschließend in den RGB-Farbraum überführt. Das Umrechnen in RGB ist für die Darstellung auf herkömmlichen Anzeigegeräten notwendig.

2.1.2 Spektrum zu XYZ Konvertierung

Im vorangegangen Abschnitt wurden die Farbangleichungsfunktionen $\overline{x}(\lambda)$, $\overline{y}(\lambda)$ und $\overline{z}(\lambda)$ bereits erwähnt. Diese bilden die Grundlage für eine Konvertierung von spektralen Energieverteilungen in für Monitore darstellbare Farben. Dabei wird zunächst ein vorhandenes SPD in den XYZ-Farbraum überführt. Anschließend erfolgt durch eine lineare Transformation die Überführung vom XYZ- in den RGB-Farbraum.
Die Farbangleichungsfunktionen liegen tabellarisch vor[2] und sind in 1nm Schritten definiert. Die Berechnung des XYZ-Tristimulus erfolgt durch die Integration[6, Seite 580]

[2] http://cvision.ucsd.edu/database/data/cmfs/ciexyz31_1.txt (Letzter Zugriff: 04.03.2009)

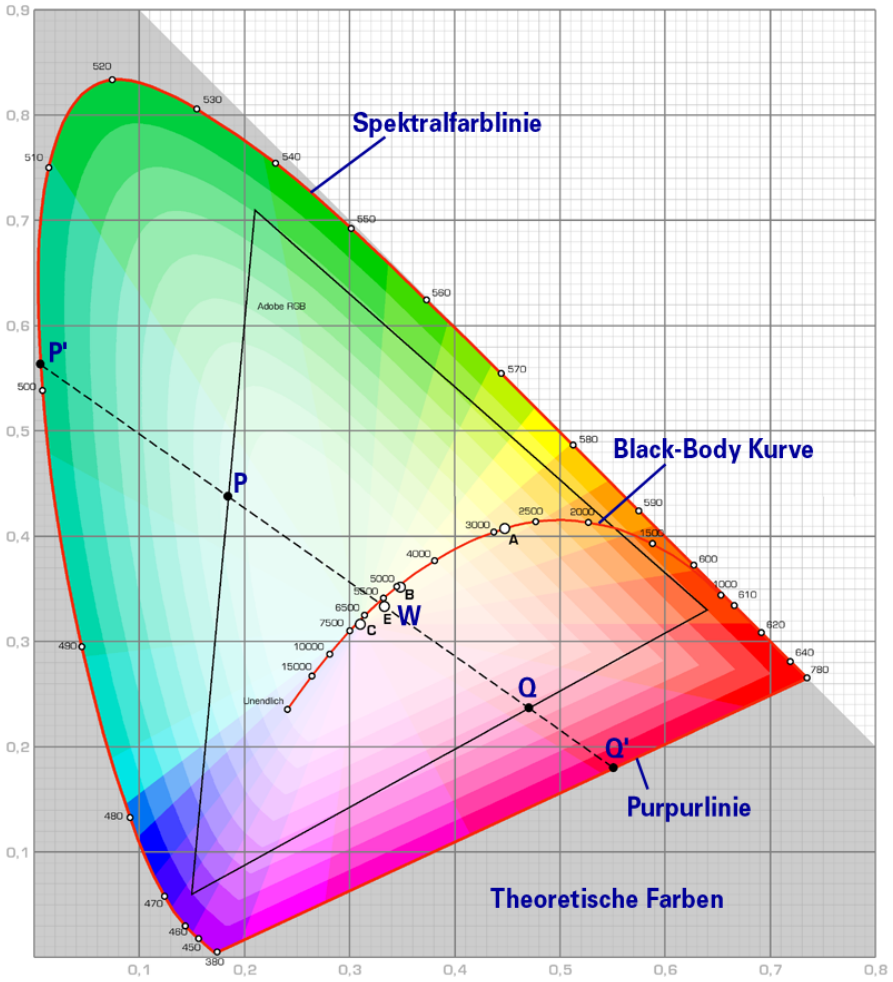


Abbildung 2.5: Darstellung der CIE Normfarbtafel. Entnommen aus wikipedia `http://de.wikipedia.org/w/index.php?title=Datei:CIE-Normfarbtafel.png&filetimestamp=20050328115020` (Letzter Zugriff: 03.03.2009)

des gegebenen Spektrums P:

$$XYZ = \begin{cases} X = k \int_{\lambda_{\min}}^{\lambda_{\max}} P(\lambda)\, \overline{x}(\lambda)\, d\lambda, \\ Y = k \int_{\lambda_{\min}}^{\lambda_{\max}} P(\lambda)\, \overline{y}(\lambda)\, d\lambda, \\ Z = k \int_{\lambda_{\min}}^{\lambda_{\max}} P(\lambda)\, \overline{z}(\lambda)\, d\lambda. \end{cases} \tag{2.4}$$

Der Faktor k dient zur Normierung von an Oberflächen *reflektiertem* Licht (Körperfarben) auf einen Wertebereich von 0 bis 100. Dabei wird für den hellsten Punkt $Y = 100$ gesetzt. Die Gleichung[6, Seite 580] zur Normierung lautet:

$$k = \frac{100}{\int_{\lambda_{\min}}^{\lambda_{\max}} P_W(\lambda)\ \overline{y}(\lambda)\ d\lambda}. \tag{2.5}$$

In der Berechnung von k taucht die SPD P_W auf. Der Grund hierfür liegt darin, dass Körperfarben nicht von selbst leuchten, sondern erst durch eine Lichtquelle, beziehungsweise eine *Lichtquellenart* beleuchtet werden muss. Ändert man die Lichtquelle, ändert sich auch der Farbeindruck. Vertreter von Lichtquellenarten sind die sogenannten *Standardlichtquellen*. Durch die Normierung mit diesen Standardlichtquellen wird definiert, welche Koordinaten das Weiß im Farbraum hat. In diesem Zusammenhang spricht man auch vom Weißpunkt. Je nach Lichtquellenart hat der Weißpunkt unterschiedliche Positionen im Farbraum.
Mit Hilfe des Planck'schen Strahlungsgesetzes für Schwarzkörper lässt sich die *Unbuntkurve* angeben, mit deren Hilfe Weißpunkte beschrieben werden können. In Abbildung 2.5 ist diese Kurve als *Black Body Kurve* gekennzeichnet. Durch Angabe einer Temperatur in K (Kelvin) kann die Energieverteilung der entsprechenden Lichtquelle nachempfunden werden. Die Temperatur ist auch gleichzeitig der Namensgeber für die Standardlichtquellen. So entspricht D65 einem idealen schwarzen Strahler mit einer Temperatur von ungefähr 6500° K. In Abbildung 2.1 sind 4 Vertreter solcher Lichtquellen als SPD grafisch dargestellt.
Für den Fall von *selbstleuchtenden* Objekten ist $k = 680\ lm/W$ (Lumen pro Watt) festgelegt. Dies entspricht einer Normierung unter Berücksichtigung der wellenlängenabhängigen Helligkeitsempfindung des menschlichen Auges. Da in unserem Fall die Farben am Ende auf dem Monitor dargestellt werden sollen, ist diese Normierung unter physiologischen Gesichtspunkten nicht erforderlich und kann weggelassen werden.
Somit lautet die Gleichung für *selbstleuchtende* Objekte

$$XYZ = \begin{cases} X = \int_{\lambda_{\min}}^{\lambda_{\max}} P(\lambda)\ \overline{x}(\lambda)\ d\lambda, \\ Y = \int_{\lambda_{\min}}^{\lambda_{\max}} P(\lambda)\ \overline{y}(\lambda)\ d\lambda, \\ Z = \int_{\lambda_{\min}}^{\lambda_{\max}} P(\lambda)\ \overline{z}(\lambda)\ d\lambda. \end{cases} \tag{2.6}$$

Die Integrale in Gleichung 2.6 beziehen sich auf kontinuierliche Funktionen P, $\overline{x}$, $\overline{y}$ und $\overline{z}$. Für die Zwecke dieser Arbeit reichen jedoch diskrete Funktionen mit einer endlichen

Anzahl an Werten. Dazu werden Stichproben, sogenannte *Samples* mit konstantem Intervall aus den Funktionen ausgewählt. Im Fall der Gleichungen 2.6 sind dies Stichproben bezüglich λ. Es wird nun festgelegt, dass

$$\lambda_{min} \leq \lambda \leq \lambda_{max}, \lambda \in \mathbb{N} \tag{2.7}$$

gilt. Trifft man nun die Annahme, dass die Funktionen diskretisiert und stückweise konstant sind, kann das Integral als Gleichung einer Summe über $\lambda_{max} - \lambda_{max} + 1$ Stichproben dargestellt werden:

$$XYZ = \begin{cases} X = \sum_{i}^{N} P_i \, \overline{x}_i \, , \\ Y = \sum_{i}^{N} P_i \, \overline{y}_i \, , \\ Z = \sum_{i}^{N} P_i \, \overline{z}_i \, . \end{cases} \tag{2.8}$$

2.1.3 XYZ zu RGB Konvertierung

Wie in den vorherigen Abschnitten bereits erwähnt, erfolgt die Konvertierung vom XYZ-Farbraum in den RGB-Farbraum durch eine lineare Transformation. Gemäß der durch die CIE definierte Gleichung 2.3 entsteht eine Transformationsmatrix M

$$M = \begin{bmatrix} 0.49, & 0.31, & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.0 & 0.01 & 0.99 \end{bmatrix}, \tag{2.9}$$

mit welcher die Primärvalenzen für den XYZ-Farbraum so festgelegt wurden, dass keine negativen Werte mehr auftreten konnten. Die Gleichung für diese Transformation lautet:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = M \begin{bmatrix} R \\ G \\ B \end{bmatrix} . \tag{2.10}$$

Damit für eine gegebene XYZ-Farbe den RGB-Farbwert ermittelt werden kann, wird die Gleichung umgestellt. Dies geschieht durch Invertieren von M und Auflösen nach RGB^T:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 2.36461 & -0.89654 & -0.46807 \\ -0.51517 & 1.42641 & 0.08876 \\ 0.00520 & -0.01441 & 1.009201 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{2.11}$$

Die in Gleichung 2.11 genutzte Transformationsmatrix entspricht der Definition nach CIE und erzeugt eine RGB-Farbe gemäß des CIE RGB-Farbraumes. Je nach Anforderungen an den gewünschten Farbraum kann diese Matrix durch andere Matrizen ersetzt werden[3].

2.1.4 RGB zu Spektrum Konvertierung

Nachdem für die Umwandlung von Spektren in darstellbare Farben die nötigen Formeln hergeleitet wurden, soll nun besprochen werden, wie aus einer gegebenen RGB-Farbe ein Spektrum ableitbar ist. Dies ist notwendig, da ein Großteil der Daten nicht als SPD vorliegen wird, sondern nur als RGB-Farbwert. Der Renderer benötigt jedoch eine spektrale Repräsentation aller Farben, damit er die Farbinformationen verarbeiten kann.
Die Umrechnung vom Spektrum in eine Farbe ist eindeutig. Das heißt, mit einem Spektrum erzeugt man genau eine Farbe. Von einer gegeben Farbe auf ein Spektrum zurück zu schließen ist auf Grund der bereits erwähnten Metamere nur sehr schwer möglich, da es eine unendliche Anzahl an möglichen Kandidaten, also Spektren, gäbe[26, Seite 1]. Das Ziel bei der Umwandlung einer RGB-Farbe in ein Spektrum ist es, den Kandidaten zu finden, der am besten zur vorgegebenen Farbe passt.
Dazu hat Smits[26] einen Algorithmus vorgestellt, der versucht, eben einen „guten“ Kandidaten zu finden. Es werden vordefinierte Basis-Spektra genutzt, um eine Konvertierung vorzunehmen. Smits hat dazu als Basis-Spektra weiß, cyan, magenta, gelb, rot, grün und blau gewählt. Nach Smits kann jede Farbe konstruiert werden, wenn diese zum weißen Spektrum addiert wird. Zusätzlich wird cyan, magenta oder gelb, sowie rot, grün oder blau hinzuaddiert. Dieser Ansatz kann als Pseudo-Code geschrieben werden und ist der Abbildung 2.6 zu entnehmen.

2.1.5 Natürliche Phänomene des Lichts

Um beobachtbare Effekte wie in diesem Abschnitt erklären zu können, sollte zunächst die *Reflexion* und *Transmission* näher betrachtet worden sein. Diese lassen sich in einfachen mathematischen Formeln ausdrücken und helfen dabei, einige der Reflexionsmodelle in Abschnitt 2.2.3 besser zu verstehen.
Die Notation für eine beliebige Richtung lautet ω. Sollen konkrete Richtungen zum Ausdruck gebracht werden, kommen Indices zum Einsatz. So steht i in ω_i für den englischen Begriff *incident* und deutet in diesem Zusammenhang auf die *einfallende Richtung* von Lichtstrahlen hin. Die Richtung ω_o bezieht sich auf das englische Wort *out* und beschreibt einen reflektierten Strahl. Transmittierende Richtungen werden mit ω_t gekennzeichnet. Dies wird in den Abbildungen 2.7(a) und 2.7(b) verdeutlicht.

[3] Eine Auflistung von verschiedenen Farbräumen und den dazugehörigen Transformationsmatrizen: `http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html` (Letzter Zugriff: 09.03.2009)

```
Spectrum RGBToSpectrum(float red, float green, float blue)
{
        Spectrum result = 0;

        if (red <= green && red <= blue)
        {
                result = red * whiteSpec;

                if (green <= blue)
                {
                        result += (green - red) * cyanSpec;
                        result += (blue - green) * blueSpec;

                } else {

                        result += (blue - red) * cyanSpec;
                        result += (green - blue) * greenSpec;
                }

        } else if (green <= red && green <= blue) {
                :
        } else { // blue <= red && blue <= green
                :
        }

        return result;
}
```

Abbildung 2.6: Pseudo-Code zur Erzeugung eines Spektrums von einer gegebenen RGB-Farbe.

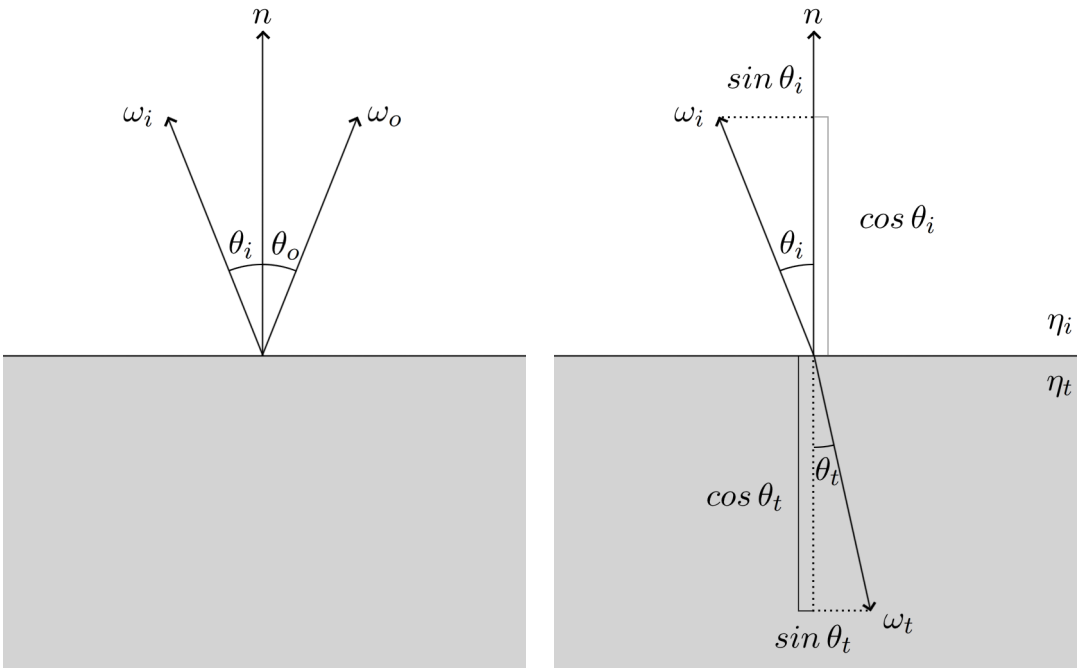
2.1.5.1 Reflexion

Die Beschreibung der idealen Spiegelung eines Lichtstrahls ist relativ einfach: Ein Lichtstrahl aus Richtung ω_i trifft die Oberfläche und wird um den dortigen Normalenvektor n gespiegelt. Der Winkel zwischen ω_i und n heißt θ_i. Er hat den gleichen Betrag wie der Winkel zwischen n und dem reflektierten Strahl ω_o[18, Seite 418]:

$$\theta_i = \theta_o. \tag{2.12}$$

2.1.5.2 Transmission

Mit dem Gesetz von *Snell* lässt sich die Transmission beschreiben[18, Seite 418]: Wieder ist ω_i die ankommende Richtung des Lichtes mit dem dazugehörigen Winkel θ_i zwischen n und ω_i. Der Winkel θ_t des transmittierenden Strahls ω_t wird durch sogenannte *Brechungs-* oder auch *Refraktionsindices* der betreffenden Medien bestimmt. Der Refraktionsindex



(a) Spekulare Reflexion

(b) Spekulare Transmission

Abbildung 2.7: Schematische Abbildung idealer spekularer Reflexion und Transmission.

für ein Medium wird durch das griechische Zeichen η (sprich „eta“) angegeben. Die Gleichung zur Berechnung einer idealen Transmission nach Snell's Gesetz lautet:

$$\eta_i \, sin\,\theta_i = \eta_t \, sin\,\theta_t. \tag{2.13}$$

Es kommt nur dann zu einer Brechung von Lichtstrahlen, wenn die Grenzschicht zweier Medien mit unterschiedlichen Indices erreicht wird. Vergleicht man beide Medien miteinander, hat das Medium mit dem größeren Brechungsindex eine größere *optische Dichte* als das Medium mit dem geringeren Brechungsindex. In Tabelle 2.1 sind exemplarisch einige Brechungsindices aufgeführt[12, Seite 258]. In der Praxis wird der Wert für das Medium Luft auf $1,0$ gerundet.
Der Brechungsindex η ist wellenlängenabhängig und kann als Funktion der Wellenlänge $\eta(\lambda)$ angegeben werden. Oft wird jedoch nur ein Wert für η angegeben. In diesen Fällen handelt es sich dann meistens um gemittelte Wert, die als Annäherung für viele betrachtete Sachverhalte ausreichend sind. Für einen Dispersionseffekt ist jedoch diese Abhängigkeit entscheidend.

2.1.5.3 Fresnel

Nach der Betrachtung von Reflexion und Transmission ist nun klar, wie die Richtungen des Lichtes, in die es gestreut wird, bestimmt werden. Jedoch ist auch die Betrach-

Medium	η bei $589,3nm$
Vakuum	1,0
Luft	1,000292
Quarzglas	1,4584
Kronglas	1,50988
Diamant	2,4173

Tabelle 2.1: Exemplarische Vertreter der Brechungsindices verschiedener Medien. Die Werte wurde unter Verwendung einer Natriumlichtquelle für die Wellenlänge $\lambda = 589,3nm$ gemessen.

tung notwendig, wie viel Licht aus einer Richtung in eine andere Richtung reflektiert oder transmittiert wird. Der Anteil der gestreuten Energie hängt von den Winkeln des einfallenden und gestreuten Lichtes ab. Dies lässt sich physikalisch genau mit den *Fresnel'schen* Formeln ausdrücken[18, Seite 419]. Die Gleichungen beziehen sich auf die Brechungsindices der beiden betroffenen Medien. Durch die Wellenlängenabhängigkeit der Brechungindices sind ebenso die Fresnel-Gleichungen wellenlängenabhängig.
Die Fresnel-Gleichungen berücksichtigen Polarisationen, also die Lage elektromagnetischer Wellen im Raum, von Lichtstrahlen. Aus Gründen der Vereinfachung werden lediglich die zwei Spezialfälle betrachtet, wie das Licht die Oberfläche erreicht. Mit $r_{\parallel}$ wird die parallele und mit $r_{\perp}$ die rechtwinklige Polarisation repräsentiert. Anschließend kann die gesamte Reflexion F_r bestimmt werden:

$$F_r = \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2). \tag{2.14}$$

Ein weiterer wichtiger Punkt ist, dass die Gleichungen mediumspezifisch sind. So wird unterschieden zwischen elektrisch leitenden und nicht leitenden Medien. Nicht leitende Medien werden auch als *dielektrische* Medien bezeichnet. Damit lässt sich beispielsweise das Reflexionsverhalten von Glas nachempfinden. Zur Modellierung von Metall müssen die Gleichungen für leitende Medien verwendet werden.
Je nach Art des Mediums sind anschließend $r_{\parallel}$ und $r_{\perp}$ bekannt und können zum Bestimmen von F_r gemäß der Gleichung 2.14 genutzt werden.

Evaluierung leitender Medien Bei der Evaluierung der Gleichung für leitende Medien bedarf es lediglich der Berechnung von $r_{\parallel}$ und $r_{\perp}$. In den dazugehörigen Gleichungen taucht eine weitere Größe neben η auf: k. Diese repräsentiert die Absorption des betrachteten Mediums, also das Umwandeln von Licht in Wärme, und muss wie η ebenfalls angegeben werden:

$$r_{\parallel}^2 = \frac{(\eta^2 + k^2)\ cos\,\theta_i^2 - 2\eta\ cos\,\theta_i + 1}{(\eta^2 + k^2)\ cos\,\theta_i^2 + 2\eta\ cos\,\theta_i + 1} \tag{2.15}$$

$$r_{\perp}^2 = \frac{(\eta^2 + k^2) - 2\eta\ cos\,\theta_i + \ cos\,\theta_i^2}{(\eta^2 + k^2) + 2\eta\ cos\,\theta_i + \ cos\,\theta_i^2}. \tag{2.16}$$

Evaluierung dielektrischer Medien Für dielektrische Medien ist die Evaluierung etwas komplexer. Zunächst muss festgestellt werden, welches Medium vom Strahl verlassen und in welches Medium eingetreten wird. Die Brechungsindices η_i beziehungsweise η_t werden dementsprechend der Medien der betroffenen Objekte zugeordnet. Es gilt die Festlegung, dass der Normalenvektor immer nach Außen zeigt. Ist der Kosinus vom Winkel zwischen einfallenden Strahl und Normalenvektor größer Null, also $cos\,\theta_i > 0$, so tritt der Strahl in das Objekt ein. Der Index η_t wird dem Medium des getroffenen Objektes zugeordnet. Der Index η_i repräsentiert das vom Strahl verlassene Medium. Ist jedoch $cos\,\theta_i < 0$, so verlässt der transmittierende Strahl das Objekt. In diesem Fall ist η_i der Index des Mediums und η_t gehört zu dem angrenzenden Medium.
Anschließend wird mittels Snell's Gesetz und der Gleichung $sin^2\,\theta + cos^2\,\theta = 1$ nach der Umstellung $sin\,\theta_t$ ermittelt[18, Seite 426]. Dies entspricht dem Sinus vom Winkel zwischen transmittierter Richtung und Normalenvektor. Die Rechte Seite der so umgestellten Gleichung wird zunächst $snellSinus\,\theta_t$ zugeordnet:

$$snellSinus\,\theta_t = \frac{\eta_i}{\eta_t}\sqrt{1 - cos^2\,\theta_i} \tag{2.17}$$

Dieser Betrag entspricht $sin\,\theta_t$, solange $snellSinus\,\theta_t \leq 1$ gilt. Mit dieser Bedingung kann nun mit der folgenden Gleichung $cos\,\theta_t$ berechnet werden:

$$cos\,\theta_t = \sqrt{1 - sin^2\,\theta_t} \tag{2.18}$$

Anschließend werden auch hier die Reflexionen für die parallele und rechtwinklige Polarisation bestimmt.

$$r_{\parallel} = \frac{\eta_t\,cos\,\theta_i - \eta_i\,cos\,\theta_t}{\eta_t\,cos\,\theta_i + \eta_i\,cos\,\theta_t} \tag{2.19}$$

$$r_{\perp} = \frac{\eta_i\,cos\,\theta_i - \eta_t\,cos\,\theta_t}{\eta_i\,cos\,\theta_i + \eta_t\,cos\,\theta_t}. \tag{2.20}$$

Wenn $snellSinus\,\theta_t$ aus Gleichung 2.17 größer 1, dann liegt eine *Totalreflexion* vor[18, Seite 426]. Das bedeutet, dass ein einfallender Strahl in einem Medium mit höherem Brechungsindex nicht in das Medium mit geringerem Index transmittiert, sondern intern reflektiert wird. Der Winkel, ab dem dieser Effekt auftritt, wird auch als *kritischer Winkel* bezeichnet. In diesem Fall ist $F_r = 1$, ansonsten erfolgt die Berechnung gemäß Gleichung 2.14:

$$F_r = \begin{cases} \frac{1}{2}(r_{\parallel}^2 + r_{\perp}^2), & \text{sonst} \\ 1, & sin\,\theta_t > 1. \end{cases} \tag{2.21}$$

2.1.5.4 Dispersion

Wie bereits bei der Transmission und den Fresnel-Gleichungen angesprochen, sind die Brechungsindizes zweier einander angrenzender Medien ausschlaggebend für die Richtung eines transmittierenden Strahls. Ändert sich der Brechungsindex in Abhängigkeit

Linie	Wellenlänge $[nm]$
B	687,0
C	656,3
D	589,3
F	486,1
G	430,8

Tabelle 2.2: Fraunhofer-Linien aus Sun et al.[27]

von der Wellenlänge, so spricht man von einer *Dispersion*. Lichtstrahlen mit kürzerer Wellenlänge – bläuliches Licht – werden stärker abgelenkt als rotes, also langwelliges Licht.
Abbildung 2.8 zeigt beispielhaft den Effekt der Dispersion mit einem dielektrischen Medium. Durch das Prisma wird weißes Licht, was aus verschiedenen Lichtstrahlen verschiedener Wellenlänge besteht, so abgelenkt, dass ein Regenbogeneffekt zu erkennen ist.

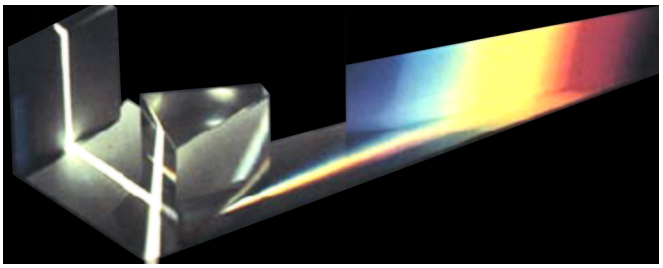


Abbildung 2.8: Dispersion: Weißes Licht wird im Prisma in seine spektralen Bestandteile zerlegt. Es entsteht ein Regenbogeneffekt. Entnommen aus Sun et al.[27].

In der Optik kann die Dispersion auf zwei Arten beschrieben werden. Bei einer der beiden Varianten wird die dispersive Stärke des Materials mit der *Abbschen Zahl V* angegeben[27]. Die dazugehörige Gleichung lautet:

$$V = \frac{\eta_D - 1}{\eta_F - \eta_C}. \tag{2.22}$$

Die Brechungsindizes η_D, η_F und η_C des Materials sind für die *Fraunhofer Linien D*, *F* und *C* angegeben. Dies sind dunkle Bereiche im Spektrum des Sonnenlichts, welche oft als Referenzmarkierung für Brechungsindizes genutzt werden[21]. In Tabelle 2.2 sind einige Vertreter der Fraunhofer-Linien mit ihrer dazugehörigen Wellenlänge, die sie repräsentieren, aufgeführt.
Eine weitere Variante für die Angabe der dispersiven Stärke mit η für die Fraunhofer-Linien G und B ist folgende Gleichung:

$$DISP = \eta_G - \eta_B. \tag{2.23}$$

Material	η_D	V	DISP
Quarz-Kristall	1,54	39,89	0,016
Kronglas	1,50	56,48	0,010
Flintglas	1,60	38,24	0,018
Diamant	2,42	37,17	0,044

Tabelle 2.3: Auswahl einiger Brechungsindizes und Dispersionsangaben aus Sun et al.[27].

In Tabelle 2.3 findet man exemplarisch einige Brechungsindizes (η_D) von transparenten Medien mit den Angaben zur Dispersion.
Im sichtbaren Bereich des Spektrums verhält sich die Dispersion für *normale* Medien wie eine monoton fallende Funktion. Dieses Verhalten lässt sich mit der *Cauchy-Reihe* ausdrücken[27]. Die allgemeine Form der Reihe lautet:

$$\eta(\lambda) = 1 + \sum_{j=0}^{i} \frac{A_j}{\lambda^{2j}}. \tag{2.24}$$

Nach dieser Form setzt sich die Gleichung zur Bestimmung von η zu einer gegebenen Wellenlänge λ aus i Koeffizienten (A) zusammen. Oft reicht jedoch auch die Evaluierung der ersten beiden Terme aus. Somit bedarf es neben der Angabe des Brechungsindexes A eines weiteren Koeffizienten B. Die vereinfachte Gleichung hat dann die Form:

$$\eta(\lambda) = A + \frac{B}{\lambda^2}. \tag{2.25}$$

Die in Gleichung 2.25 dargestellte Form wird für die Umsetzung der Dispersion dienen. Somit bleibt festzuhalten, dass neben dem Brechungsindex auch ein Dispersionskoeffizient, des Weiteren *CauchyB* genannt, benötigt wird.

2.2 Grundlagen des Raytracing

In diesem Abschnitt soll ein kurzer Einblick in die Grundlagen des Raytracing gegeben werden. Mit diesen Kenntnissen können dann exemplarisch einige Reflexionsmodelle betrachtet und umgesetzt werden. Um die Richtungen eines Strahls auseinander halten zu können, soll die nun folgende Terminologie dienen: Strahlen, die in die Richtung des Beobachters zeigen, werden mit ω_o, Strahlen, die in Richtung Licht zeigen, mit ω_i bezeichnet. In Abbildung 2.9 ist diese Terminologie dargestellt, wie sie auch bei Veach[30] sowie Pharr und Humphreys[18] genutzt wird. Wichtig hierbei ist, dass beide Richtungen, ω_i sowie ω_o, immer vom Punkt p wegzeigen. Dies muss bei der Betrachtung vor allem bei den Reflexionsmodellen berücksichtigt werden.
Des Weiteren können Richtungen in sphärischen Koordinaten werden. Somit lassen sich Richtungsvektoren in Form von zwei Winkeln angeben. Die beiden Winkel werden als θ und ϕ bezeichnet und haben folgende Bedeutung:

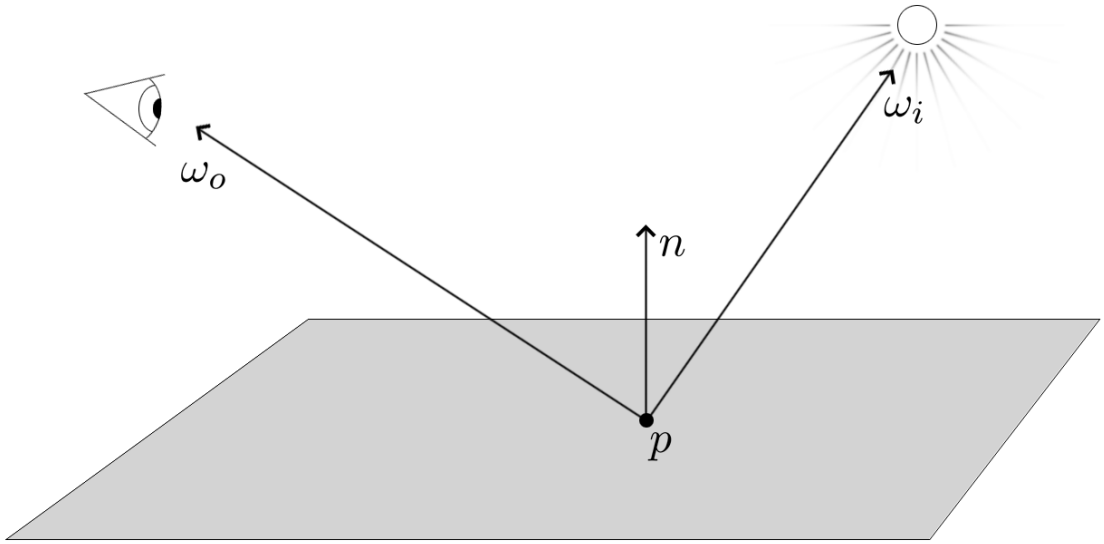


Abbildung 2.9: Das einfallende Licht ω_i wird am Punkt p in Richtung ω_o reflektiert.

- θ: Winkel zwischen der Richtung ω und der z-Achse,
- ϕ: Winkel zwischen der Richtung ω und der x-Achse.

Die Abbildung 2.10 verdeutlicht die Definition dieser Winkel.

2.2.1 Rekursives Raytracing

Dass Strahlenverfolgung als ein rekursives Problem aufgefasst werden kann, stellte Whitted[32] in seinem Paper fest. Ein Strahl, im Auge beginnend, wird durch eine virtuelle Szene verfolgt. Am nächstgelegenen Auftreffpunkt kann dann der Strahl um den Normalenvektor reflektiert und die Strahlenverfolgung rekursiv wiederholt werden. Analog wird dies auch mit einem transmittierenden Strahl durchgeführt.
Das Grundprinzip ist in Abbildung 2.11 angedeutet. Vor einer virtuellen Kamera beziehungsweise dem Beobachter der Szene wird ein Rastergitter aufgespannt. Die Breite und Höhe des Rasters entsprechen der Auflösung der zu erstellenden Szenen-Abbildung. Die dabei entstehenden Kästchen entsprechen einem Bildpunkt, auch als Pixel bezeichnet. Durch jeden dieser Bildpunkte wird nun ein Strahl geschickt. Alle Strahlen haben den selben Ursprung, welcher der Position der Kamera in der Szene entspricht.
Die Strahlen werden in Schnittpunkttests mit Objekten der Szene getestet. Ist der nächste Schnittpunkt gefunden, wird die farbliche Erscheinung des Objekts ermittelt und der Beitrag auf den Bildpunkt summiert, durch den der Strahl geschickt wurde. Im Fall von Whitteds Ansatz würde nun am Schnittpunkt der Strahl um den Normalenvektor reflektiert und/oder transmittiert und der Strahl rekursiv weiterverfolgt werden.

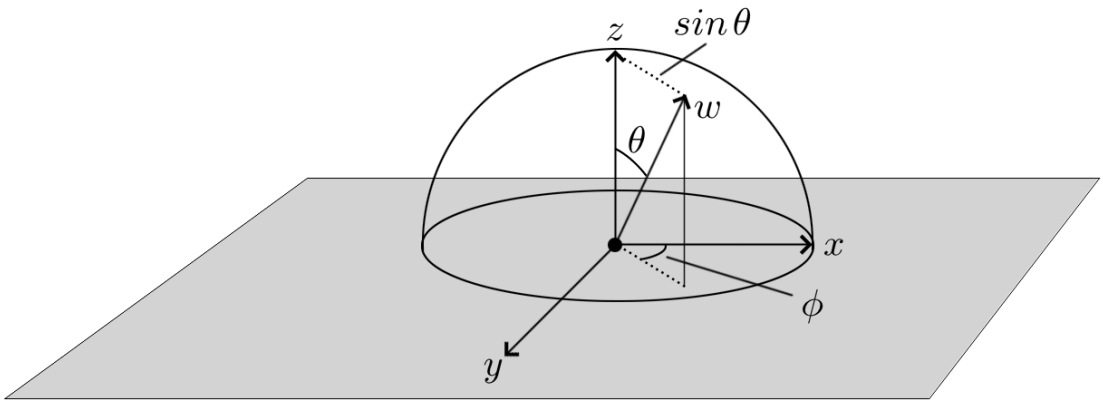


Abbildung 2.10: Verdeutlichung der sphärischen Koordinaten, die den Winkeln θ und ϕ zwischen ω und der z- beziehungsweise x-Achse entsprechen. Die Länge des Richtungsvektors ω ist eins.

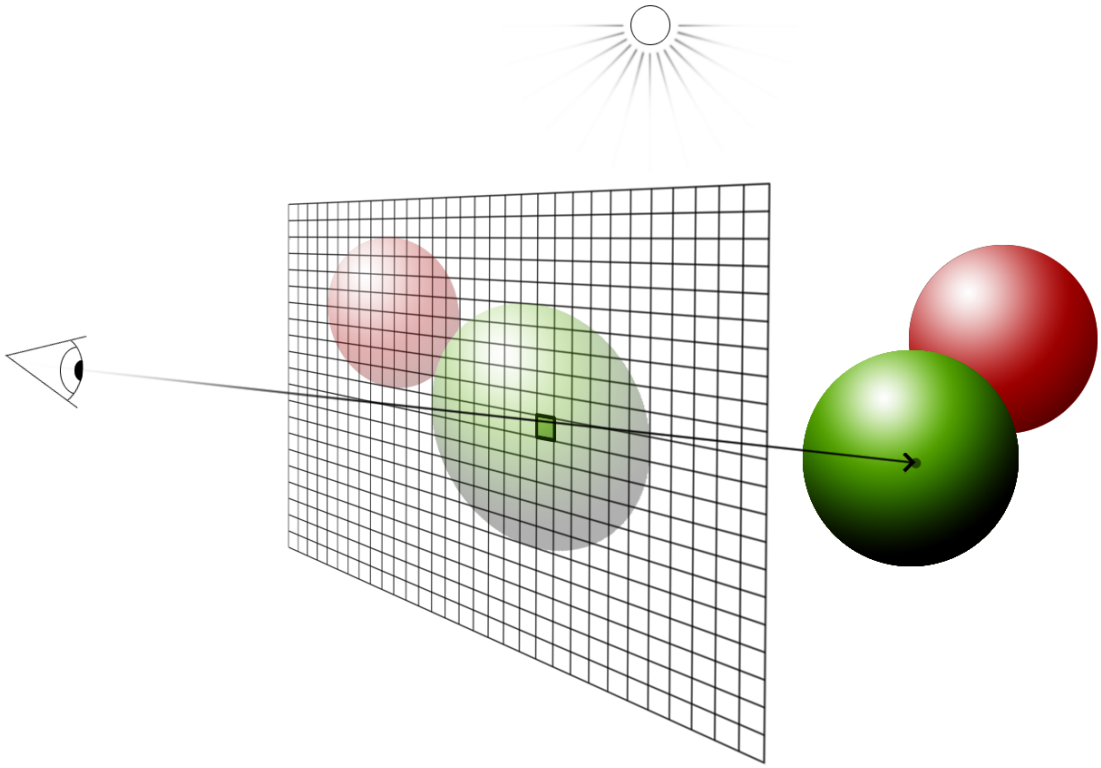


Abbildung 2.11: Grundprinzip: Vor dem Betrachter wird ein Raster aufgespannt. Durch jedes der entstehenden Kästchen wird ein Strahl geschickt und die Szene damit „abgetastet“.

Mit diesem Verfahren war es also möglich, spiegelnde und transparente Objekte darzustellen. Zwar konnten auch diffuse Flächen durch die Berechnung eines lokalen Beleuchtungsmodells dargestellt werden, jedoch war es nicht möglich diffuse Mehrfachreflexionen zu erzeugen. Denn die Richtung des reflektierten Strahls ω_i ist nur abhängig von der Richtung des Betrachters ω_o. Ändert sich die Richtung des Betrachters, erscheint der betrachtete Punkt in der Szene auch anders. Bei diffusen Oberflächen ist dies jedoch nicht der Fall. Egal aus welcher Richtung ein Punkt mit diffuser Oberfläche betrachtet wird, der Farbeindruck ändert sich nicht.

2.2.2 Path Tracing und Monte Carlo

Whitteds Ansatz war hervorragend geeignet, um spiegelnde oder transparente Objekte darzustellen. Jedoch gibt es in der Natur nicht nur ideal reflektierende beziehungsweise transmittierende Lichtstrahlen. Daher bedarf es einer allgemeineren Formulierung, um die Palette an darstellbaren Effekten zu erweitern zu können.

2.2.2.1 Rendergleichung

Eine solche allgemeine Formulierung drückt die sogenannte Lichttransport-Gleichung, auch als *Rendergleichung* bezeichnet, aus:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{S^2} f(p, \omega_o, \omega_i)\ L_i(p, \omega_i)\ |cos\theta_i|\ d\omega_i. \tag{2.26}$$

Sie wurde durch Kajiya[9] beschrieben und besagt, dass die Strahlung $L_o(p, \omega_o)$ von einem Punkt p in Richtung ω_o die Summe des emittierten Lichts dieses Punktes ($L_e(p, \omega_o)$) und die gesamte ankommende Strahlung aus allen Richtungen, verteilt über die Hemisphäre S^2, ist. Wie viel Strahlung am Punkt p reflektiert wird, legt die *Bidirectional Scattering Distribution Function*, *BSDF*, fest. Sie wird in Gleichung 2.26 durch $f(p, \omega_o, \omega_i)$ repräsentiert und drückt das Verhältnis zwischen eingehender und ausgehender Strahlung aus. Somit wird die eingehende Strahlung $L_i(p, \omega_i)$ skaliert, auf eben genau das beschriebene Verhältnis.
Die Herleitung von $|cos\theta_i|$, auch als *Kosinusterm* oder etwas allgemeiner *geometrischer Term* bezeichnet, bezieht sich auf das *Lambert'sche Gesetz* und wird hier übersprungen. Wichtig zu wissen ist jedoch, dass die Beleuchtungstärke auf einer Oberfläche direkt vom Winkel zwischen einfallendem Strahl und Normale abhängt.
Die Notation der BSDF dient als Generalisierung von zwei Sonderfällen. Denn Lichtstrahlen können an Objekten reflektiert werden oder transmittieren durch sie hindurch. Für diese beiden Fälle bedarf es jedoch besonderer Behandlungen. Daher wird für reflektierte Strahlen die *Bidirectional Reflectance Distribution Function*, *BRDF*, und für transmittierende Strahlen die *Bidirectional Transmittance Distribution Function*, *BTDF*, verwendet.

Die Notation für die BRDF lautet $f_r(p, \omega_o, \omega_i)$, für die BTDF $f_t(p, \omega_o, \omega_i)$, wobei im ersten Fall ω_o und ω_i auf der selben Seite der Hemisphäre liegen. Im letzteren Fall liegen beide Richtungen dagegen auf entgegengesetzten Hemisphären.
Bei der Skalierung der Strahlung durch eine BRDF gelten zwei wesentliche physikalische Eigenschaften[30, Seite 87], [18, Seite 250]. Die *Reziprozität* besagt, dass für alle Paare der Richtungen ω_i und ω_o gilt:

$$f_r(p, \omega_i, \omega_o) = f_r(p, \omega_o, \omega_i). \tag{2.27}$$

Eine weitere Eigenschaft ist die Energieerhaltung:

$$\int_{H^2(n)} f_r(p, \omega_o, \omega')\ cos\theta'\ d\omega' \leq 1. \tag{2.28}$$

Das bedeutet, dass für die Hemisphäre $H^2(n)$ um die Normale n weniger oder höchstens gleich so viel Energie reflektiert wird, wie eingefallen ist.
Die Energieerhaltung gilt auch für die BTDF, jedoch nicht die Reziprozität. Das bedeutet, dass bei der Betrachtung von BTDFs eine spezielle Behandlung erfolgen muss, damit der Pfad umkehrbar wird. Nur so können dann Strahlen von der Kamera durch die Szene hin zur Lichtquelle verfolgt werden.
Die BSDF hat also zur Aufgabe, eingehende Verteilungen von Strahlungen und Richtungen in eine ausgehende zu transformieren. Dazu werden in Abschnitt 2.2.3.2 einige Vertreter genauer betrachtet.

2.2.2.2 Monte Carlo Integration

Das Lösen des Integrals in Gleichung 2.26 ist auf analytischem Wege nicht möglich. Denn für jeden Punkt im Raum müssten unendlich viele Richtungen betrachtet werden, aus der Lichtstrahlen kommen könnten. Da generell solche Integralgleichungen analytisch nicht zu lösen sind, werden numerische Ansätze gewählt, wie beispielsweise die „Monte Carlo Integration“[18, Seite 631].
Monte Carlo nutzt dabei die Methode der Stochastik aus, um ein Integral der Form $\int f(x)dx$ zu evaluieren. Wird auf diesem Wege ein Problem wiederholt evaluiert, erhält man unter Umständen unterschiedliche Ergebnisse, sofern sich die Zufallswerte unterscheiden. Mittelt man die Ergebnisse, so ist es äußerst wahrscheinlich, dass das tatsächliche Ergebnis angenähert wurde[18, Seite 632].
Mit Hilfe der Zufallszahlen werden Integrale evaluiert, indem an beliebigen Punkten des Integrationsbereiches „Stichproben“ entnommen werden. Mit den Stichproben kann nun das gesamte Integral abgeschätzt werden. Die Wahl der Stichproben – des Weiteren als *Samples* bezeichnet – spielt hierbei eine wichtige Rolle.
Wie bereits verdeutlicht, werden Integrale abgeschätzt, was bedeutet, dass kein exaktes Ergebnis erzielt wird. Bei dem oben erwähnten Beispiel kann durch Mittlung aller Werte der Erwartungswert $E_P[f(x)]$ angenähert werden. Mit anderen Worten: der erwartete

Wert ist der Mittelwert aller erzielten Werte. Damit die Berechnungen möglichst schnell zum erwarteten Wert konvergieren, werden unterschiedliche Strategien angewendet.
Vergleicht man nun tatsächlich erhaltene Werte mit dem erwarteten Wert, lässt sich feststellen, wie groß der Fehler, sprich die Varianz, ist. Im Falle des Raytracers drückt sich Varianz als Rauschen im Bild aus. Wird ein Integral evaluiert, nimmt der Fehler mit einer Rate von $O(\sqrt{n})$ ab. Mit anderen Worten: Um das Rauschen zu halbieren, müssen vier mal so viele Samples genommen werden, dass heißt vier mal mehr Aufwand.
Damit nicht zwangsläufig mehr Samples genommen werden müssen, ist die Wahl der Samples ein wesentlicher Punkt beim Evaluieren der Integrale, um Varianz zu vermindern[4]. *Importance Sampling* ist eine solche Strategie. Die Idee dahinter soll nun kurz verdeutlicht werden.
Nach Pharr und Humphreys entspricht der erwartete Wert E eines Integrals dem sogenannten *Monte Carlo Estimator*[18, Seite 636]:

$$E[\int f(x)\, dx] = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}. \tag{2.29}$$

Mit einer endlichen Anzahl von N Samples werden Werte des Integrals entnommen und mit der Anzahl der Samples normiert. So entsteht ein Mittelwert aller erhaltenen Werte.
Beim Importance Sampling werden die Samples von der Wahrscheinlickeitsdichtefunktion (*Probability Density Function*, *PDF*) $p(x)$ gewonnen, mit deren Hilfe die Funktion $f(x)$ evaluiert wird. Gleicht die Form der PDF der Funktion $f(x)$, wird die Varianz reduziert[18, Seite 673].
Das bedeutet, dass zu jeder Funktion $f(x)$ eine passende PDF, dass heißt eine Funktion $p(x)$, angegeben werden muss, die der Form von f ähnelt. Dies ist wichtig bei der Betrachtung der Reflexionsmodelle.

2.2.2.3 Zufallszahlen

Die Monte Carlo Integration nutzt Zufallswerte, um Integrale abschätzen zu können. Für die Erzeugung dieser Zahlen wird ein Mechanismus benötigt, der gleichverteilte Zahlenfolgen generiert, in einem Bereich von $[0, 1]$. Für den Raytracer kommt der Pseudo-Zufallszahlengenerator (*Pseudo Random Number Generator, PRNG*) von Park und Miller[16] zum Einsatz.
Der PRNG ist einfach umzusetzen und benötigt als Eingabe einen sogenannten „Seed“, der auch als interner Zustand aufgefasst werden kann. Nach jedem „Ziehen“ einer Zufallszahl ändert sich dieser. Würde der gleiche Seed wieder als interner Zustand dienen, käme das gleiche Ergebnis beim Ziehen der Zufallszahl heraus. Um Muster zu vermeiden, muss der Zustand des PRNG immer mitgeführt werden.
Die Berechnungen basieren auf einer Primzahl m und Z_n als Seed und haben die Form:

$$Z_{n+1} = Z_n \cdot a \; mod \; m. \tag{2.30}$$

[4]Diverse Sampling-Strategien wurden dazu vorgeschlagen. Gute Zusammenfassungen lassen sich in Veach[30] sowie Pharr und Humphreys[18] finden

Genauere Details zur Wahl der Zahlen a und m sind aus Park und Millers Paper[16] zu entnehmen. Für den Einsatz auf der Grafikkarte wurden aus Gründen der Genauigkeit bei der Zahlendarstellung die Parameter $a = 75$ und $m = 2^16 + 1 = 65537$ gewählt.

2.2.2.4 Path Tracing

Nachdem die rekursive Strahlenverfolgung durch Whitted[32] eingeführt wurde, war man nun in der Lage, ideal spiegelnde Oberflächen oder transparente Objekte darzustellen. An jedem Schnittpunkt wurden reflektierende und transmittierende Strahlen weiterverfolgt, was das Darstellen von Spiegeln oder Glaskugeln ermöglichte. Die für globale Beleuchtungssysteme so wichtigen diffusen Oberflächen wurden dabei jedoch nur unzureichend berücksichtigt.
Dies ermöglichte Cook et al. *Distributed Raytracing*[4]: An jedem Schnittpunkt mit Objekten wurden mehrere Sekundärstrahlen erzeugt. Somit konnten nun neben den spiegelnden und transmittierenden Effekten eben auch diffuse Oberflächen behandelt werden. Der Nachteil dieser Methode ist jedoch, dass ein wesentlich größerer Aufwand im Vergleich zu Whitteds Ansatz betrieben werden musste. Denn die Anzahl der zu verfolgenden Strahlen stieg mit steigender Rekursionstiefe exponentiell schnell an.
Kajiya[9] ging hier einen anderen Weg. Seine Grundidee war es, an jedem Schnittpunkt nur einen Strahl weiter zu verfolgen. Wurde die maximale Rekursionstiefe erreicht, wird dann in der Kamera ein neuer Primärstrahl erzeugt. Durch das Verfolgen nur eines Strahls, an einem Schnittpunkt, entsteht somit ein Pfad, der in der Kamera beginnt und auf einer Lichtquelle der Szene endet. Die entstehenden Pfade sind auch gleichzeitig der Namensgeber dieser Technik: *Path Tracing*. Optimiert wurde diese Variante mit dem Prinzip der direkten Beleuchtung: Auf jedem Schnittpunkt wird das lokale Beleuchtungssystem evaluiert.
Um das Problem der globalen Beleuchtung zu verallgemeinern, formulierte Kajiya die Rendergleichung und schlug vor, diese nach dem Monte Carlo Prinzip basierend auf Zufallswerten zu evaluieren. Damit die gleichen Effekte wie in Whitteds Raytracer möglich sind, sind reflektierende und transmittierende Strahlen essentiell. Wie es dennoch möglich ist, mit der Weiterverfolgung nur eines Strahls diese Effekte zu erzielen, soll nun kurz veranschaulicht werden.
Alle Pfade werden unabhängig voneinander betrachtet. Angenommen, es werden zwei Pfade P_0 und P_1 durch einen Bildpunkt verfolgt. Der erste Auftreffpunkt ist für beide Pfade eine Glaskugel. Die Wahrscheinlichkeiten, ob nun ein Strahl reflektiert oder transmittiert werden soll, sind gleichverteilt und liegen jeweils bei 50%[5]. Durch die beliebige Auswahl einer konkreten BSDF kann es passieren, dass im Pfad P_0 an dieser Stelle reflektiert und im Pfad P_1 transmittiert wird. Am Ende werden die Beiträge beider Pfade gemittelt. Somit wird der Punkt in der Kamera, in dem beide Pfade ihren Ursprung besitzen, durch Lichtstrahlen beleuchtet, die von der Kugel reflektiert beziehungsweise durch sie transmittiert werden.

[5] Die Auswahl der Richtungen kann auch nach anderen Kriterien, wie zum Beispiel Grad der Transparenz erfolgen. Einfachheitshalber betrachten wir aber eine Glaskugel mit eben genau dieser Wahrscheinlichkeitsverteilung.

Neben den bereits erwähnten ideal reflektierenden und ideal transmittierenden Strahlen ist mit Path Tracing das Verfolgen von drei weiteren Ereignissen verbunden. Die während mit Path Tracing behandelten Ereignisse lauten:

- Reflexion: ideal und diffus
- Transmission: ideal und diffus
- Absorption

Die Pfade nehmen also unterschiedliche Verläufe durch die Szene. Dies kann zu einer starken Varianz im Ergebnis-Bild führen, was sich als Varianz-Rauschen bemerkbar macht. Wie im vorherigen Teil bereits erwähnt, hat die Varianz ihren Ursprung in der Evaluierung der Rendergleichung, da nur Stichproben aus dem gesamten Integral genommen werden, mit deren Hilfe dann versucht wird, das gesamte Ergebnis abzuschätzen. Varianz kann unter anderem durch eine Vielzahl von zusätzlichen Pfaden durch einen Bildpunkt reduziert werden. Dies wird auch als *Supersampling* bezeichnet. Würden unendlich viele Pfade in einem Bildpunkt erzeugt, entspräche der erwartete Helligkeitswert dieses Bildpunktes dem Durchschnitt aller erhaltenen Werte, die Varianz würde gegen Null konvergieren und es gäbe kein Bildrauschen.

2.2.3 Reflexionsmodelle

In diesem Abschnitt sollen nun einige Vertreter von Reflexionsmodellen, des Weiteren als BSDFs bezeichnet, vorgestellt werden. Die Umsetzung der BSDFs[6] ist einer der wesentlichen Punkte dieser Arbeit, da die benötigten Daten Struktur und Konzept der Ablaufsteuerung des spektralen Raytracers beeinflussen. Um zu wissen, welche Daten während des Renderns zur Verfügung stehen müssen, ist daher ein Blick in die Theorie dieser Vertreter notwendig.
Der Begriff *Bidirectional Scattering Distribution Function* wurde in Abschnitt 2.2.2 bereits eingeführt. Zwei konkrete Vertreter sind die BRDF und BTDF. Je nach Fähigkeit eines Modells, ob Strahlen nur reflektiert oder nur transmittiert werden, kann man dies der BRDF beziehungsweise BTDF zuordnen. Modelle, die beide Fähigkeiten besitzen, werden wieder als BSDF bezeichnet.
Neben dieser Unterteilung können Reflexionen an der Oberfläche in weitere Kategorien unterteilt werden: diffus, verschmierte Spiegelung und perfekte Spiegelung. Die Hauptaufgabe einer BSDF ist es, die spektrale Verteilung eingehender Strahlung und die Verteilung der Richtungen reflektierter Strahlen zu modellieren.
Bei der Betrachtung von Reflexionsmodellen wird zunächst der Effekt der Dispersion nicht berücksichtigt. Dies hat den Grund, dass sich bei der Berechnung von dispersiven Effekten lediglich der Brechungsindex in Abhängigkeit der Wellenlänge des betroffenen Mediums ändert. Die restlichen Gleichungen bleiben davon jedoch unberührt. Daher sollte der für diesen Effekt anzugebende Koeffizient CauchyB nicht vergessen werden.

[6] Die Evaluierung einer BSDF wird des Weiteren auch als *Shading* bezeichnet.

Ein weiterer Punkt ist, dass die Farbe des Objektes in den folgenden Abschnitten keine Berücksichtigung findet. Diese wird im spektralen Kontext, wie bereits in Abschnitt 2.1 erläutert, als spektrale Verteilung k_R definiert. Da alle BSDFs mit k_R skaliert werden, wird darauf in den jeweiligen Abschnitten nicht weiter eingegangen.

2.2.3.1 Shading-Koordinatensystem

Die Betrachtungen der BSDFs beziehen sich auf ein Koordinatensystem für Reflexionen. So ist das Ermitteln einiger Quantitäten einfacher zu bewerkstelligen. Das Prinzip wurde auch in Pharr und Humphreys[18, Seite 411] umgesetzt und wird daher hier auch genutzt. Das Shading-Koordinatensystem wird durch den Normalenvektor n an einem Punkt auf einer Oberfläche und die beiden zueinander rechtwinkligen Vektoren s und t definiert. Sie bilden eine Tangentialebene, auf der n senkrecht steht. Abbildung 2.12 verdeutlicht diese Definition.

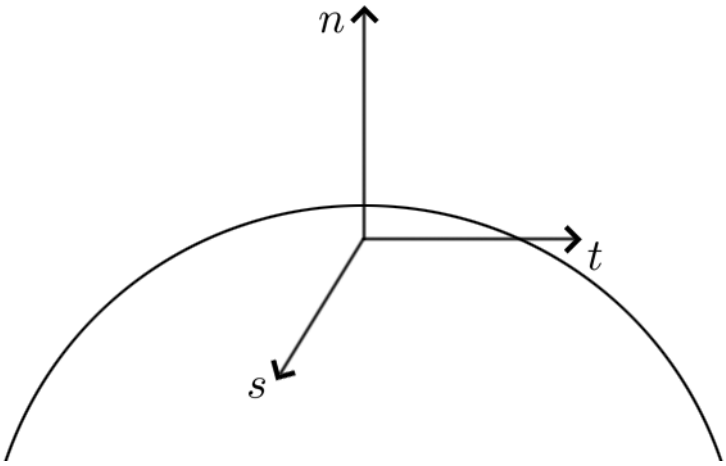


Abbildung 2.12: Koordinatensystem für die BSDFs: Zur Normalen n werden zwei Vektoren s und t bestimmt, die rechtwinklig zu einander und zu n sind. Somit spannen s und t eine Tangentialebene bezüglich des Punktes auf der Oberfläche auf.

Durch die Verwendung eines eigenen Koordinatensystems können Richtungen einfach in sphärischen Koordinaten angegeben werden. Im lokalen System hat n die Richtung $(0, 0, 1)$, zeigt also senkrecht nach oben. Der für viele Berechnungen wichtige Wert des Kosinus von θ ist als Skalarprodukt zwischen n und w definiert, und kann nun innerhalb dieses Koordinatensystems wie folgt ausgedrückt werden:

$$cos\,\theta = (n \cdot \omega) = ((0, 0, 1)^T \cdot \omega) = \omega_z \tag{2.31}$$

Somit entspricht also der Wert der z-Komponente von ω dem Kosinus von θ. Das heißt, dass dieser Wert in einem lokalen Shading-Koordinatensystem abgelesen werden kann ohne eine zusätzliche Berechnung. Ebenso einfach ist das Berechnen von $sin\,\theta$. Nach

einigen Umstellungen erhält man folgende Gleichung:

$$sin\,\theta = \sqrt{1 - \omega_z^2} \tag{2.32}$$

Damit in einem Shading-Koordinatensystem gearbeitet werden kann, muss zwangsläufig eine Transformation aus dem globalen Koordinatensystem, vor der eigentlichen Evaluierung der BSDF, durchgeführt werden. Nach der Evaluierung erfolgt anschließend wieder die Rücktransformation in den globalen Raum der neu erzeugten Richtungen.

2.2.3.2 Lambert'sche Reflexion

Die wohl einfachste Form aller BSDF ist das Lambert'sche Modell. Strahlen werden an der Oberfläche mit gleichem Energiebetrag in alle Richtungen reflektiert. Daher ist es der BRDF zuzuordnen mit der Eigenschaft, Strahlen diffus zu reflektieren. Ein Blatt Papier beispielsweise entspricht dem Verhalten eines diffusen Reflektors.
Die BRDF basiert auf dem *Gesetz von Lambert* und besagt, dass die Menge an Licht, die eine Fläche A erreicht, proportional zu $cos\,\theta$ ist[18, Seite 238]. Die Leistung, mit der die Lichtquelle strahlt, wird durch Φ, mit der Einheit W, ausgedrückt.
Die Bestrahlungsstärke E (W/m^2), auch als *Strahlungsstromdichte* bezeichnet, gibt an, wie viel Licht auf die Oberfläche, in Abhängigkeit der Fläche, einfällt. Dies kann mit folgender Gleichung ausgedrückt werden:

$$E = \frac{\Phi\, cos\,\theta}{A}. \tag{2.33}$$

Das heißt, E hat sein Maximum, wenn die Lichtquelle senkrecht ($\theta = 0°$) auf eine Fläche A scheint. Denn dann ist der $cos\,\theta = 1$. Mit zunehmenden Winkel θ nimmt nun E ab. Bei $\theta = 90°$ ist der Winkel so groß, dass die Lichtquelle die Fläche nicht mehr beleuchtet.
Die Skalierung in Abhängigkeit von $cos\,\theta$ ist auch Teil der Rendergleichung und taucht als *Kosinusterm* in Gleichung 2.26 auf. Das bedeutet, dass alle evaluierten BSDFs dieser Skalierung unterzogen werden. Da der Beitrag einer ideal diffusen Reflexion in alle Richtungen konstant ist, bleibt an dieser Stelle nichts weiter zu tun, als lediglich den Wert der diffusen Farbe des Objektes zurückzugeben.
Bleibt noch die Erzeugung des reflektierten Strahls, beziehungsweise die Richtung: Dies geschieht mit einer *kosinus-gewichteten Hemisphäre*. Zwar könnte man bei einer diffusen BRDF einfach eine zufällige Richtung aussuchen, jedoch würden hier eventuell Richtungen ausgesucht werden, die nicht viel oder gar nichts zu der Beleuchtung des betroffenen Punktes beitragen. Dies ist vor allem dann der Fall, wenn θ sehr groß ist.
Also bieten sich hier eher Richtungen mit einem kleinem θ an. In diesem Zusammenhang wird auch von einem *kosinus-gewichteten Hemisphären-Sampling*[18, Seite 656] gesprochen. Wie bei Pharr und Humphrey wird hierbei nach der Methode von *Malley* vorgegangen, wobei Punkte, beliebig über eine Scheibe verteilt, auf eine darüberliegende Hemisphäre projiziert werden. Daraus kann nun für eine gegebene Richtung ω_o die reflektierte Richtung ω_i erzeugt werden.
Im *Monte Carlo Estimator* in Gleichung 2.29 wird für das *Importance Sampling* die PDF, also die Wahrscheinlichkeitsverteilung $p(x)$, benötigt. Damit die Varianz-Reduktion effizient eingesetzt werden kann, sollte die Form der PDF der Form des zu evaluierenden

Integrals ähneln. Sie gibt für zwei gegebene sphärische Koordinaten einen Wert zurück, der der Wahrscheinlichkeit entspricht, mit der der Strahl eben in diese Richtung reflektiert wird. Da alle BSDFs nicht in Abhängigkeit von sphärischen Koordinaten, sondern in Abhängigkeit vom Raumwinkel angeben sind, vereinfacht sich die Gleichung 2.33 wie folgt:

$$p(\theta) = \frac{1}{\pi} cos\,\theta. \tag{2.34}$$

2.2.3.3 Spekulare Reflexion

Mit dem Anteil des Lichtes F_r, welches an der Oberfläche reflektiert wurde, kann nun die BRDF $f_r(p, \omega_o, \omega_i)$ der spekularen Reflexion umgesetzt werden. Wie in Gleichung 2.12 verdeutlicht, wird Licht aus einer gegebenen Richtung ω_i nur in eine Richtung ω_o reflektiert. Alle anderen Kombinationen von Richtungen, wobei $\theta_i \neq \theta_o$ gilt, reflektieren kein Licht.
Dieser Sachverhalt lässt sich mathematisch am besten durch eine Delta-, oder auch Dirac-Funktion darstellen. Die Herleitung folgender Gleichung ist zu entnehmen aus Pharr und Humphreys[18, Seite 428]:

$$f_r(p, \omega_o, \omega_i) = F_r(\omega_o)\frac{\delta(\omega_i - R(\omega_o, n))}{|cos\,\theta_i|}. \tag{2.35}$$

Die δ-Funktion dient zur Sicherstellung, dass nur dann Licht reflektiert wird, wenn die Bedingung $\theta_i = \theta_o$ erfüllt ist. $R(\omega_o, n)$ beschreibt hierbei die um die Normale n gespiegelte Richtung ω_o.
Die Erzeugung der reflektierten Richtung ω_i ist, auch wegen der Verwendung des Shading-Koordinatensystems, denkbar einfach: Da diese nichts anderes ist als eine 180° Rotation um die z-Achse, lässt sich dies durch eine Transformationsmatrix der Form

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

darstellen. Wird die Richtung ω_o mit dieser Matrix multipliziert, hat dies den Effekt, dass nur die x- und y-Komponente negiert werden:

$$\omega_{ix} = -\omega_{ox}$$
$$\omega_{iy} = -\omega_{oy}$$
$$\omega_{iz} = \omega_{oz}$$

Die PDF dieser BRDF ist eins: Der Grund hierfür liegt darin, dass es für eine gegebene Richtung ω_o keine Chance für eine andere Richtung gibt, als eben die der idealen Reflexion. Das bedeutet, dass keine andere Richtung per Zufall ausgewählt werden kann. Mathematisch verhält sich diese BRDF also wie eine Dirac-Funktion[18, Seite 427]:

$$p(\theta) = \begin{cases} \infty, & \text{wenn } \theta_i = \theta_o \\ 0, & \text{sonst.} \end{cases} \tag{2.36}$$

2.2.3.4 Spekulare Transmission

Analog zur Evaluierung der spekularen Reflexion in Gleichung 2.35 lässt sich die BTDF ebenfalls als Delta-Funktion angeben[18, Seite 431]:

$$f_t(p, \omega_i, \omega_t) = \frac{\eta_o^2}{\eta_i^2}(1 - F_r(\omega_o))\frac{\delta(\omega_i - T(\omega_i, n))}{|cos\,\theta_i|}. \tag{2.37}$$

Der Richtungsvektor der Transmission um die Normale n wird durch $T(\omega_i, n)$ repräsentiert. Bei der Reflexion nimmt der Anteil an gestreuter Strahlung mit steigendem θ zu. Das bedeutet, je flacher eine Oberfläche betrachtet wird, desto mehr Strahlung wird auf der Oberfläche reflektiert. Die Transmission nimmt jedoch nur mit fallendem θ zu: Je steiler auf eine transparente Oberfläche geschaut wird, desto mehr Strahlung wird transmittiert. Betrachtet man einen Fluss von oben, kann man unter Umständen den Grund des Flusses sehen. Je flacher der Beobachtungswinkel jedoch wird, desto mehr wird die Umgebung reflektiert. Diese Beobachtung wird mit dem Term $1 - F_r(\omega_o)$ ausgedrückt.
Die Bestimmung des Richtungsvektors basiert auf Snells Gesetz. Zunächst muss wieder festgestellt werden, welcher Brechungsindex auf der einfallenden beziehungsweise transmittierenden Seite der Schnittstelle liegt. Ähnlich wie bei der spekularen Reflexion werden die x- und y-Komponenten von ω_i um die Normale n gespiegelt und mit dem Verhältnis $sin\,\theta_t / sin\,\theta_i$ skaliert. Dies entspricht nach Snells Gesetz dem Verhältnis der Brechungsindices: $\eta = \eta_i / \eta_t$.
Für die Bestimmung der z-Komponente wird $cos\,\theta_t$ wie folgt berechnet:

$$\begin{aligned} sin\,\theta_i^2 &= 1 - \omega_{iz}^2 \\ sin\,\theta_t^2 &= \eta^2 sin\,\theta_i^2 \\ cos\,\theta_t &= \sqrt{1 - sin\,\theta_t^2}. \end{aligned}$$

An dieser Stelle ist auch der Fall der totalen internen Reflexion zu beachten, wobei es zu keiner Transmission kommt. Dies wurde bereits in Abschnitt 2.1.5.3 erläutert.
Somit lässt sich nun der Richtungsvektor für die Transmission bestimmen:

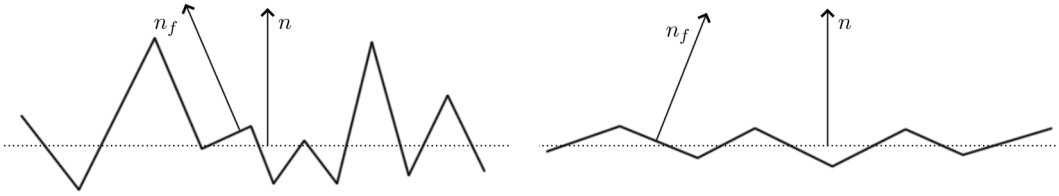
$$\begin{aligned} \omega_{tx} &= \eta\,(-\omega_{ix}) \\ \omega_{ty} &= \eta\,(-\omega_{iy}) \\ \omega_{tz} &= cos\,\theta_t. \end{aligned}$$

Die Bestimmung der Funktion $p(x)$ ist ebenso analog zur spekularen Reflexion. Da es nur eine mögliche transmittierende Richtung eines einfallen Lichtstrahls gibt, hat diese die Wahrscheinlichkeit Eins, ansonsten Null, und lässt sich mittels Snells Gesetz so ausdrücken:

$$p(\theta) = \begin{cases} \infty, & \text{wenn } \eta_i\ sin\,\theta_i = \eta_t\ sin\,\theta_t \\ 0, & \text{sonst.} \end{cases} \tag{2.38}$$

2.2.3.5 Microfacetten-Modelle

Die bisherigen Reflexionsmodelle vereinfachten die Oberflächen der Objekte auf perfekte diffuse oder spekulare Reflektoren, was jedoch nicht realen Oberflächen entspricht[3]. Torrance und Sparrow[29] modellierten mit Hilfe kleiner Mikrofacetten eine raue Oberfläche. Es wird die Annahme getroffen, dass eine beleuchtete Fläche A im Vergleich zu einer Mikrofacette relativ groß ist. Somit wird eine große Anzahl von Mikrofacetten beleuchtet, die das Verhalten der Lichtstreuung bestimmen.
Die Verteilung der Mikrofacetten und deren Anordnung wird statistisch beschrieben. Für Facetten kann in Abhängigkeit von ihrer Anordnung relativ zur Normalen der Oberfläche ermittelt werden, wie das Licht von einer einzelnen Facette reflektiert wird. Somit lässt sich dann das gesamte Reflexionsverhalten der Oberfläche bestimmen. Die Abbildung 2.13 zeigt zwei Flächen mit ihren jeweilig verschiedenen Anordnungen der Facetten.



(a) Raue Oberfläche (b) Glatte Oberfläche

Abbildung 2.13: Die Oberfläche wird durch eine Verteilungsfunktion der Normalenvektoren n_f der Mikrofacetten in Abhängigkeit von der Normale n beschrieben. Vergleich zweier Oberflächen mit unterschiedlicher Oberflächenrauheit: Abbildung a) starke Abweichung der Facetten-Normalen von der Oberflächen-Normalen, somit entsteht ein rauer Eindruck der Oberfläche. Abbildung b) erscheint mit einer geringeren Abweichung dagegen glatter.

Jede einzelne Facette wird als perfekter Spiegel betrachtet, das heißt, jede Facette reflektiert Licht spekular[29]. Ist die durchschnittliche Varianz der Facetten-Normalen n_f zur Oberflächen-Normale n relativ groß, wie auf der linken Seite in Abbildung 2.13(a), so erscheint die Oberfläche als sehr rau[7]. Der Grund ist, dass jede Facette das Licht in unterschiedliche Richtungen reflektiert. Dagegen erscheint mit sinkender Varianz zwischen

[7] Wie in Abbildung 2.13 zu erkennen ist, vergrößert sich die Oberfläche mit zunehmender Rauheit. Dies sollte bei der Evaluierung mit berücksichtigt werden. Genauere Angaben zu diesem Sachverhalt befinden sich in Veach[30] Abschnitt 5.3.

n_f und n die gesamte Fläche immer mehr als idealer Spiegel, da das Licht nicht mehr so breit gestreut wird.
Die Oberfläche wird durch die statistische Verteilungsfunktion $D(\omega_h)$ der Mikrofacetten beschrieben. Unter der Voraussetzung, dass die Facetten spekular reflektieren, entspricht der Halbwinkel-Vektor ω_h der Facetten-Normalen n_f. So kann ω_h wie folgt bestimmt werden:

$$\omega_h = \frac{\omega_i + \omega_o}{|\omega_i + \omega_o|}. \tag{2.39}$$

Die Funktion $D(\omega_h)$ gibt an, wie hoch die Wahrscheinlichkeit ist, dass die Facetten die Ausrichtung von ω_h besitzen. Anders ausgedrückt, welcher Anteil des ankommenden Lichtes wird spekular reflektiert. Bei einer geringen Oberflächenrauheit wären die Facetten sehr ähnlich denen der Oberflächen-Normale angeordnet und würden somit einen Großteil des ankommenden Lichtes spekular reflektieren.
Die eigentliche Verteilungsfunktion $D(\omega_h)$ der Facetten ist nicht festgeschrieben. Hier gibt es verschiedene Vertreter, die brauchbare Ergebnisse liefern. So verhält sich der Funktionsverlauf der Mikrofacetten-Verteilung nach Blinn exponentiell[18, Seite 446] und lautet:

$$D(\omega_h) = \frac{e+2}{2\pi} \, (\omega_h \ \cdot n)^e. \tag{2.40}$$

Der Parameter e, als Exponent bezeichnet, bestimmt hierbei, wie groß die Wahrscheinlichkeit ist, dass ω_h von der Normalen n stark abweicht. Je größer also e, desto mehr gleicht die Oberfläche dann einem Spiegel.
Eine weitere benötigte Funktion ist der geometrische Term $G(\omega_o, \omega_i)$. Unterschiedlich angeordnete Facetten können ankommendes Licht blockieren und verhindern somit die Beleuchtung eventuell benachbarter Facetten. Ebenso ist es möglich, dass sich Facetten gegenseitig verdecken und somit eventuell für den Beobachtbar nicht sichtbar sind. Aus Cook und Torrance ist folgende Gleichung zu entnehmen:

$$G(\omega_o, \omega_i) = min \left\{ 1, \ \frac{2(n \cdot \omega_h)(n \cdot \omega_o)}{\omega_o \cdot \omega_h}, \ \frac{2(n \cdot \omega_h)(n \cdot \omega_i)}{\omega_o \cdot \omega_h} \right\}. \tag{2.41}$$

Zusammen mit einem Fresnel-Term $F_r(\omega_o)$ für leitende Medien, wie er in 2.14 angegeben ist, lässt sich nun eine allgemeine Form des Mikrofacetten-Modells zur Evaluierung der dazugehörigen BRDF aufstellen[18, Seite 443]:

$$f_r(p, \omega_o, \omega_i) = \frac{D(\omega_h) \ G(\omega_o, \omega_i) \ F_r(\omega_o)}{4 \ cos\,\theta_o \ cos\,\theta_i}. \tag{2.42}$$

Ursprünglich tauchte im Nenner an der Stelle des Faktors 4 ein π auf. Jedoch wurde dies in der späteren Literatur korrigiert[31].
Die Erzeugung des reflektierten Strahls übernimmt die Mikrofacetten-Verteilung, da sie den größten Einfluss auf die Erscheinung der Oberfläche hat[18, Seite 681].
Im Fall der Blinn-Verteilung wird dabei wie folgt vorgegangen: Zunächst muss in Abhängigkeit von der Rauheit ein Vektor ω_h erzeugt werden. Hierzu wird mit Hilfe zweier

Zufallszahlen u_1 und u_2 auf der Hemisphäre ein Punkt gewählt und somit die Richtung von ω_h bestimmt. Je nach Wahl des Exponenten e entspricht dies mehr oder weniger der Richtung der Oberfläche-Normalen n. Die Gleichungen dazu lauten wie folgt:

$$cos\,\theta = \sqrt[e+1]{u_1} \tag{2.43}$$

$$sin\,\theta = \sqrt{1 - cos^2\,\theta} \tag{2.44}$$

$$\phi = 2\,u_2\,\pi. \tag{2.45}$$

$$\omega_h x = sin\,\theta\,cos\,\phi \tag{2.46}$$

$$\omega_h y = sin\,\theta\,sin\,\phi \tag{2.47}$$

$$\omega_h z = cos\,\theta. \tag{2.48}$$

Anschließend wird ω_o um ω_h gespiegelt, um ω_i zu erzeugen. Je nachdem, wie weit n_f von der Normalen n abweicht, wird der Lichtstrahl bezüglich der Normalen n ideal oder eher diffus gespiegelt.
Für die Bestimmung der PDF wird ebenfalls die genutzte Mikrofacetten-Verteilung genutzt. In diesem Fall ist die Verteilung nach Blinn:

$$p(x) = \frac{(e+1)cos^e\,\theta}{2\pi\ 4(\omega_o \cdot \omega_h)}. \tag{2.49}$$

Das hier vorgestellte Mikrofacetten-Modell trifft die Annahme, dass die Mikrofacetten kleine perfekte Spiegel sind. Torrance und Sparrow[29], die als erste eine Oberflächenbeschreibung basierend auf Mikrofacetten vorstellten, beschrieben die Verteilung der Mikrofacetten mit einer Gauß'schen Verteilungsfunktion namens Beckmann-Verteilung. Cook und Torrance[3] verallgemeinerten diesen Ansatz, in dem die Verteilung auch durch andere Funktionen beschrieben werden kann, und nutzten beispielhaft die Verteilung nach Blinn.
Im Gegensatz zu diesen beiden Modellen beschrieben Oren und Nayar[14] die Mikrofacetten als kleine perfekte diffuse Reflektoren. Das dazugehörige Modell stellt eine generalisierte Form das Lambert'schen Modells dar.
Eine weitere Variante eines Facettenmodells ist das von Walter et al.[31]. Hier werden die Mikrofacetten als kleine Glas-Facetten betrachtet, und erweitert somit das Modell von Cook und Torrance um die Transmission. Im Gegensatz zu den anderen Modellen, die ausschließlich den BRDFs zuzuordnen sind, ist dieses Modell ein Vertreter der BSDFs, da neben Reflexion auch Transmission behandelt wird. Mit dieser BSDF lassen sich Effekte wie raues Glas modellieren.
Die Verteilungen der Mikrofacetten nach Blinn oder Beckmann erlauben es, *isotrope* Reflexionsmodelle zu beschreiben: Das Verhalten der Lichtstreuung hängt nur vom Winkel zwischen Halbwinkel-Vektor und Oberflächen-Normale ab. In diesem Zusammenhang wird auch von symmetrischen BRDFs gesprochen. Um Effekte wie gebürstetes Metall zu erzeugen, bedarf es jedoch einer speziellen Beschreibung der Facetten-Verteilung, die nun nicht mehr nur abhängig vom Winkel zwischen Halbwinkel-Vektor und Normale ist. Diese Verteilungsmodelle werden beim Einsatz von *anisotropen* Reflexionsmodellen benötigt.

Wird ein beliebiger Punkt auf der Oberfläche um die Normalen-Achse rotiert, ändert sich bei anisotropen BRDFs der Betrag des reflektierten Lichts. Isotrope Modelle reflektieren immer den gleichen Betrag, unabhängig von der Rotation um die Normale. Ashikhmin und Shirley[1] stellten hierzu ein Modell vor, das durch Angabe zweier Parameter die Ausrichtung der Facetten beeinflusst.

2.2.3.6 Zusammenfassung

In den vorangegangenen Abschnitten wurden einige Reflexionsmodelle behandelt. Hier folgt nun eine Übersicht an objektspezifischen Daten, die zur Evaluierung der BSDFs erforderlich sind:

- Lambert'sche Reflexion

  k_R (Farbe des Objektes)

- Spekulare Reflexion

  k_R (Farbe des Objektes)

  η (Brechungsindex)

  CauchyB (Dispersions-Koeffizient)

- Spekulare Transmission

  k_T (Farbe des Objektes)

  η (Brechungsindex)

  CauchyB (Dispersions-Koeffizient)

- Microfacetten

  k_R (Farbe des Objektes)

  η (Brechungsindex)

  e (Exponent, Rauheit)

Des Weiteren ist zu beachten, dass die BSDFs für spekulare Reflexion und spekulare Transmission in einem Material, bezeichnet als *Glas*, zusammengefasst werden. Die beiden Komponenten können dann beim Evaluieren des Materials beliebig, auf Zufallszahlen basierend, ausgewählt werden.
Abbildung 2.14 veranschaulicht die drei aufgezählten Materialien. Die dazu verwendeten Parameter sind der Tabelle 2.4 zu entnehmen.

2.3 Sunshine – Ein GPU Raytracer

Dieser Abschnitt des Kapitels soll einen kurzen Einstieg zum Programmieren auf der GPU geben. Da einige Aspekte Auswirkungen auf das Konzept und die Umsetzung haben, sind einige Dinge im Vorfeld zu klären. Anschließend erfolgt ein Einblick in Aufbau

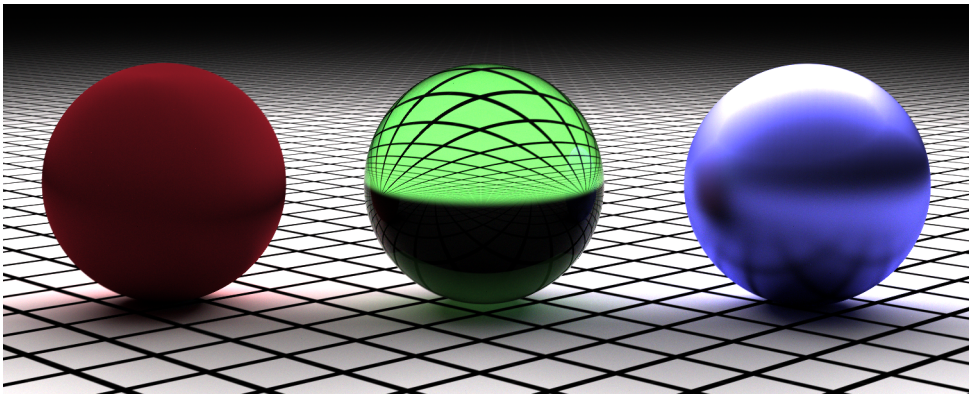


Abbildung 2.14: Beispiel der drei verschiedenen Materialien (von links nach rechts): Lambertian, Glas und Microfacetten.

	Material	Parameter
linke Kugel	Lambertian	k_R = rotes Spektrum
mittlere Kugel	Glas	k_R = weißes Spektrum k_T = grünes Spektrum $\eta = 1{,}5$ $cauchyB = 0$
rechte Kugel	Microfacetten	k_R = blaues Spektrum $\eta = 0{,}0$ $e = 100$

Tabelle 2.4: Auflistung der Parameter der verwendeten Materialien in Abbildung 2.14.

und Ablauf des bestehenden GPU Path Tracers *Sunshine*. Die aktuelle Umsetzung von Sunshine ist Ausgangspunkt für das Anwenden der entwickelten Konzepte. Sie unterscheidet sich jedoch in einigen Punkten von der Umsetzung, die vorgestellt wurde in der Arbeit *Stochastischer GPU-Strahlenverfolger für GroIMP*[8].
Sunshine basiert auf der Programmiersprache JAVA und nutzt die Schnittstelle OpenGL. Die Kommunikation zwischen JAVA und OpenGL erfolgt mit dem JAVA-Binding JOGL[8].

2.3.1 Zugrundeliegende Prinzipien für das Arbeiten auf der GPU

OpenGL ist eine Schnittstelle, die zur Entwicklung von GPU-Applikationen entworfen wurde[25, Seite 1]. Diese Schnittstelle wird in CPU-Anwendungen dazu genutzt, die GPU zu steuern. Die Steuerung der GPU erfolgt nach dem *Client-Server*-Prinzip. Es werden Nachrichten in Form von Kommandos durch Methoden-Aufrufe an die GPU übermittelt, dort in einem Puffer gespeichert und der Reihe nach abgearbeitet.

[8] `https://jogl.dev.java.net/` (Letzter Zugriff: 19.03.2009)

Neben dem Steuern über Kommandos gibt es auch die Möglichkeit, Programmcode direkt auf der GPU auszuführen. Die C-ähnliche Sprache *Graphics Library Shading Language, GLSL*, wurde dazu als Teil der OpenGL Spezifikation Version 2.0 mit aufgenommen[22, Seite 35].
Im Folgenden soll nun beschrieben werden, wie Programm-Code auf die GPU transferiert und ausgeführt wird, wie Daten bereitgestellt und gespeichert werden. Zusätzlich soll kurz auf Probleme und Engpässe hingewiesen werden, die erheblichen Einfluss auf den konzeptionellen Teil dieser Arbeit haben.

2.3.1.1 Texturen

Das Bereitstellen von Daten auf der GPU wird häufig durch Texturen realisiert. Allgemein formuliert handelt es sich dabei um n-dimensionale Felder (*Arrays*). Texturen werden in der Regel als 2D Array genutzt, um Bilddaten abzulegen. Diese Bilddaten können dann dazu genutzt werden, um ein sogenanntes *Texture-Mapping* durchzuführen. Dabei werden 2D Bilder auf 3D Objekte wie Kugeln oder Boxen projiziert.
Beim Erstellen der Textur kann neben der Größe – Breite und Höhe – auch die Formatierung festgelegt werden. Beispielsweise kann die Textur für ein RGB-Bild in drei „Kanäle“ pro Feldeintrag formatiert werden. Das bedeutet, dass jede Zelle drei Werte aufnimmt. Wird dann auf eine Stelle in der Textur zugegriffen, erhält man drei Werte für rot, grün und blau. Im Zusammenhang mit Texturen werden die einzelnen Zellen auch als Pixel (Picture Elements) bezeichnet.
Aus Wright et al. [33, Seite 310] ist eine Auflistung der verschiedenen Formatierungen zu entnehmen. Eine mögliche Formatierung für das eben genannte Bild-Beispiel wäre `GL_RGB32F_ARB`, oder `GL_RGBA32F_ARB`, soll ein Alpha-Wert für die Transparenz mit übergeben werden. Diese *interne Formatierung* besagt also, dass pro Pixel in der Textur ein 32 Bit *Fließkomma*-Wert hinterlegt werden kann[9]. Somit werden Fließkomma-Werte in einfacher Genauigkeit dargestellt.
Für das Auslesen der Texturen kann ebenfalls ein Datentyp angegeben werden. Für unsere Zwecke ist der Datentyp `GL_FLOAT` ausreichend, so dass die Daten aus der Textur von der CPU als Float-Werte interpretiert werden.
Die Genauigkeit lässt sich beim Erzeugen der Texturen separat angeben, was an dieser Stelle jedoch nicht weiter wichtig ist. Abbildung 2.15 zeigt zwei unterschiedlich formatierte Texturen beispielhaft.
Der Vorteil der Formatierung von Texturen ist, dass es den Zugriff beziehungsweise die Berechnung des Index erheblich erleichtern kann.
Das Erzeugen von Texturen verursacht eine Speicherbereitstellung auf der GPU. Während oder nach der Erzeugung von Texturen können Daten von der CPU auf die GPU transferiert werden. Dazu werden in JAVA von `java.nio.Buffer` erbende Klassen genutzt, in denen sich Daten „verpacken“ und transferieren lassen. Konkret kommt der `java.nio.ByteBuffer` als Datencontainer zum Einsatz.

[9] Das `*_ARB` ist an dieser Stelle nicht weiter wichtig, es besagt lediglich, dass es ein fester Bestandteil in einer sich weiterentwickelnden OpenGL-Spezifikation ist.

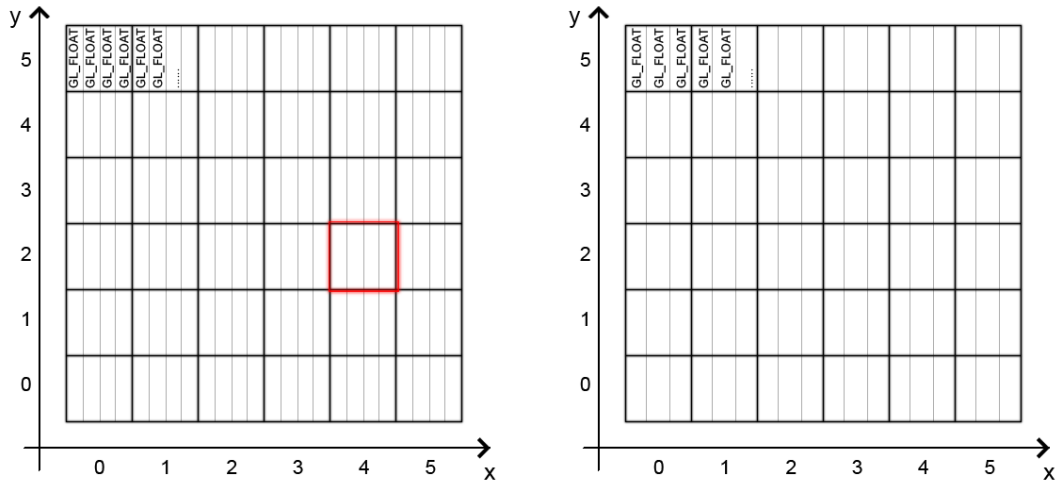


Abbildung 2.15: Die Abbildung zeigt zwei Texturen mit einer Größe von 6x6 Pixeln. Auf der linken Seite ist die Textur mit `GL_RGBA32F_ARB`, auf der rechten mit `GL_RGB32F_ARB` formatiert. Die Werte der Kanäle eines Pixels werden auf CPU-Seite auf Grund der Angabe durch `GL_FLOAT` als Float-Wert interpretiert. Das rot eingefärbte Kästchen (links) repräsentiert ein Pixel in dieser Textur und hat den 2D Index $x = 4$ und $y = 2$.

Weitere Implementierungsdetails zu Textur-Erzeugung und Verwaltung sind aus dem Tutorial von Dominik Göddeke[7] zu entnehmen.

2.3.1.2 Frame-Buffer-Objekt

In Abschnitt 1 wurde bereits erwähnt, dass die Grafikkarte als eine Pipeline arbeitet. Am Ende dieser Pipeline steht der *Frame-Buffer*. Das heißt, alle berechneten Werte werden in den Frame-Buffer geschrieben und können als Farbe interpretiert werden. Sie stehen nun für die Anzeige auf dem Bildschirm bereit. Sollten die Daten jedoch wieder verwendet werden können, müsste man diese Daten aus dem Frame-Buffer in eine Textur kopieren. Damit die zeitraubenden Lese- und Schreiboperation nicht ausgeführt werden müssen, bietet sich die sogenannte „Ping Pong“-Technik an. Diese Technik ist sozusagen ein Standard-Werkzeug in der GPGPU-Programmierung. Auch in der Arbeit von Huwe[8] kam sie zum Einsatz.
Mit Hilfe dieser Technik wird nicht in den Frame-Buffer geschrieben, sondern direkt in Texturen, die vorher angegeben wurden. Diese Texturen werden an ein *Frame-Buffer-Objekt* (FBO) gebunden. Vor der Ausführung eines Programms auf der GPU werden an das FBO diese Texturen gebunden. Dies können im Regelfall bis zu vier verschiedene Texturen sein. Diese vier Texturen werden des Weiteren zusammengefasst als *Textur-Set*

bezeichnet. Neuere Grafikkarten können auch mehr als vier Texturen an das FBO binden, aber aus Gründen der Abwärtskompatibilität in Bezug auf ältere Modelle beschränken wir die Anzahl auf vier.
Erzeugen wir nun ein Textur-Set A und ein Textur-Set B, kann eines der Sets, beispielsweise A, an den FBO gebunden werden. Das Set B dient derweil als Eingabe. Ist das Programm beendet und sind alle Ergebnisse in das Set A geschrieben, können nun die Rollen vertauscht werden. Das bedeutet, dass A als Eingabe dient und B das Set ist, in das das Ergebnis geschrieben wird. Das ständige Tauschen der Rollen von Sets ist auch der Anlass für die Namensgebung dieser Technik: *Ping Pong*. Dieses Prinzip ist schematisch in Abbildung 2.16 dargestellt.

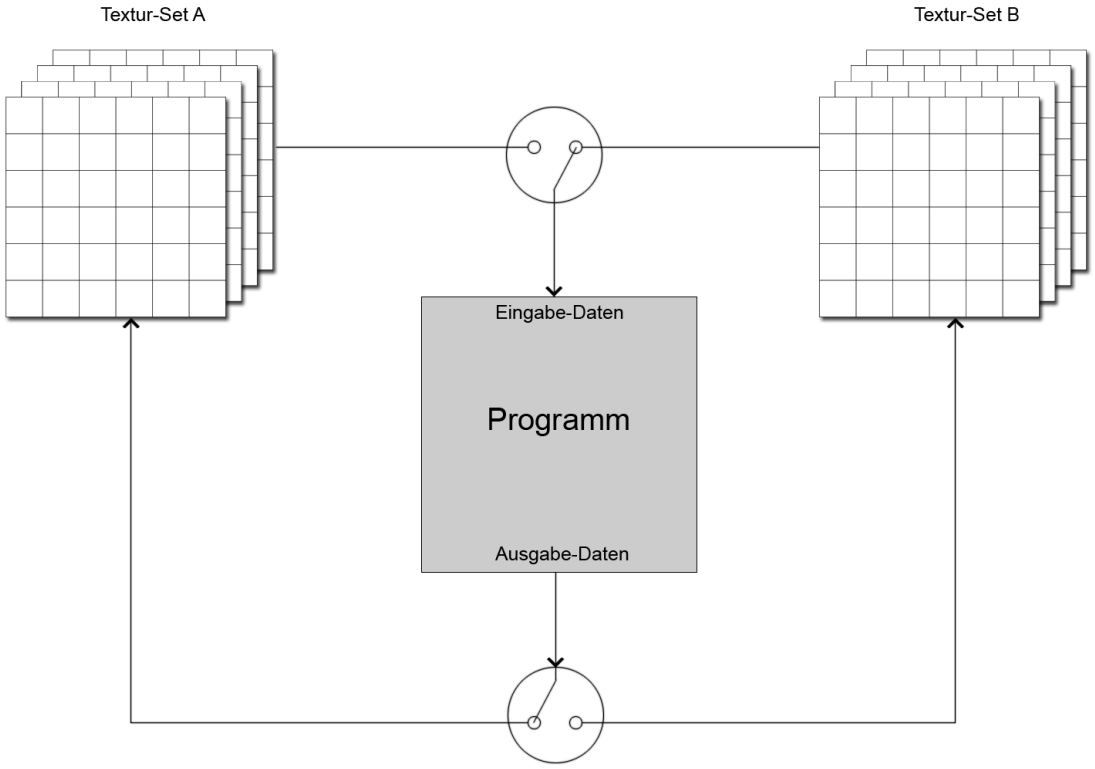


Abbildung 2.16: Die Abbildung zeigt das Prinzip der „Ping Pong“-Technik: Zwei Textur-Sets (A und B), mit je vier Texturen, tauschen nach der Ausführung des Programms ihre Rollen.

Ein an das FBO gebundenes Textur-Set kann nicht gleichzeitig als Eingabe fungieren. Jedoch können mehrere Textur-Sets oder einfach nur einzelne Texturen an das Programm als Eingabe weitergegeben werden. Neben den bereits genannten gibt es noch weitere Einschränkungen in Bezug auf die Menge an Daten. Das ausgeführte Programm kann zwar auf Eingabedaten beliebig zugreifen, hat jedoch eine ihm fest zugewiesene Pixel-Position für das Schreiben des Resultats in die einzelnen Texturen des Sets. Der Grund dafür wird im nächsten Abschnitt erläutert.

Die Einschränkung durch die feste Zuordnung der Adresse beim Schreiben führt auch dazu, dass nur eine begrenzte Anzahl an Daten nach einer Programm-Ausführung gespeichert werden kann. Wie bereits erwähnt, werden maximal vier Texturen in einem Set zusammengefasst und an das FBO gebunden. Sind diese Texturen mit `GL_RGBA32F_ARB` formatiert, kann also eine Textur vier 32 Bit Float-Werte aufnehmen. Bei vier Texturen pro Set macht dies also eine maximale Anzahl von 16 Float-Werten. Diese Limitierung ist eine erhebliche Beeinträchtigung: Informationen, die in darauffolgenden Durchläufen benötigt werden, aber aus Mangel an Platz nicht gespeichert werden können, gehen verloren und müssen somit in der nächsten Runde neu berechnet oder aus dem Grafikspeicher gelesen werden.

2.3.1.3 GLSL Programm - Einschränkungen

Bevor wir fortfahren, soll an dieser Stelle noch einmal kurz der „normale“ Ablauf auf einer Grafikkarte skizziert werden. Eine darzustellende Szene durchläuft die gesamte Pipeline, wie in Abbildung 1.1 dargestellt. Durch Transformationen beziehungsweise Annäherungen an Kurven oder Oberflächen werden aus den einzelnen Objekte der Szene Vertices erstellt. Dies geschieht in sogenannten Vertex-Prozessoren. Die Programme, die in diesen Prozessoren ausgeführt werden, heißen Vertex-Programme oder auch *Vertex-Shader*. Anschließend werden nicht mehr sichtbare Objekte am Rand abgeschnitten (Clipping). In der Rasterisierung werden aus den Vertex-Daten eine Reihe von *Fragmenten*, mit einer festen Adress-Zuordnung im Frame-Buffer, generiert. Fragmente beinhalten Informationen über Farbe, Tiefe und Textur-Daten und werden im Fragment-Prozessor ausgeführt. Die Fragmente müssen in der Pipeline eine Reihe von Tests überstehen und können entweder verworfen, direkt in den Frame-Buffer geschrieben, oder mit bereits existierenden Werten „vermischt“ werden. Somit wird am Ende aus einem oder mehreren Fragmenten ein Pixel im Frame-Buffer[25, Seite 4, 10 ff.].
Der Unterschied zwischen den Typen Vertex- und Fragment-Prozessor ist der, dass nur letzterer in der Lage ist, Texturen einzulesen. In dieser Arbeit spielt daher der Vertex-Prozessor keine Rolle.
Die in Abbildung 2.16 schlicht als „Programm“ bezeichnete Anwendung ist also tatsächlich ein Fragment-Programm. In der Rasterisierung werden für diese Programme Adressen des Frame-Buffers fest zugewiesen. Dies erklärt auch, warum Fragment-Programme nur an eine Stelle im Frame-Buffer schreiben können und somit das Speichern auf 16 Werte pro Ausführung limitieren. Die Adress-Zuordnung steht jedem Fragment-Programm als Variable zur Verfügung.

2.3.1.4 GLSL Programm - Erzeugung und Ausführung

Nun soll kurz verdeutlicht werden, wie GLSL-Programme erzeugt und ausgeführt werden. In unserem Fall sind GLSL-Programme ausschließlich für den Fragment-Shader gedacht und werden daher auch als Fragment-Programme bezeichnet.

Programm-Erzeugung Wie bereits erwähnt, ist GLSL der C-Sprache sehr ähnlich. Der Ausgangspunkt für ein Fragment-Programm ist daher ebenfalls eine Textdatei mit Programm-Code. Somit ist das Programmieren auf einer GPU ebenfalls mit einer Hoch-Sprache möglich. Jedoch ist GLSL keine objektorientierte Sprache, was das Abstrahieren von der Problembeschreibung erschwert. Dennoch ist es möglich, modular zu arbeiten, in dem Funktionen in extra Quellcode-Dateien ausgelagert werden. Bei dem Erstellen des Fragment-Programms werden dann die einzelnen Dateien gesammelt und zu einem *Shader* verpackt und in einem Programm dann ausgeführt. Konkret werden alle Quellcode-Dateien gesammelt und in ein `java.lang.String`-Array verpackt. Dieses wird dann an das Fragment-Programm übergeben. In Abbildung 2.17 ist eine beispielhafte Umsetzung aufgeführt.

```
// Definition des Quellcodes
String[] source = {
        loadSource("VariableDefinitions.frag"),
        loadSource("ExampleMain.frag")
};

// Zusammenfassung der OpenGL-API
GL gl;

// Erzeugen eines GLSL-Programmes
int program = gl.glCreateProgramObjectARB();

// In einem Programm werden Shader ausgefuehrt..
// dieser wird hier spezifiziert
int shader = gl.glCreateShaderObjectARB(GL.GL_FRAGMENT_SHADER_ARB);

// Verknuepfung des Quellcodes mit dem Shader
gl.glShaderSourceARB(shader, source.length, source, null);

// Compilieren des Quellcodes im Shader
gl.glCompileShaderARB(shader);

// Nun wird der Shader mit dem Programm verknuepft
gl.glAttachObjectARB(program, shader);

// Durch das Linken wird aus dem compilierten Code ausfuehrbarer Code
gl.glLinkProgramARB(program);
```

Abbildung 2.17: Pseudo-Code zur Erzeugung eines Fragment-Programms.

Wie bei gängigen Compilern für Hochsprachen können hier ebenfalls Syntax-Fehler festgestellt und angezeigt werden. Zumindest syntaktisch lässt sich ein GLSL-Programm debuggen. Defekte aufgrund von Fehlverhalten des Fragment-Programms zu finden, ist jedoch generell schwer[22, Seite 220]. Dennoch bietet sich die Möglichkeit, Werte an den vermuteten Stellen im Code abzufragen und direkt in den Frame-Buffer schreiben zu

lassen. Diese können dann auf CPU-Seite ausgelesen und in einer Text-Datei gespeichert und durch den Programmierer inspiziert werden.

```
// Das Ausfuehrung des Programms wird vorbereitet
gl.glUseProgramObjectARB(program);

// Eine uniform-Variable im GLSL-Code names "cpuVar"
int location    = gl.glGetUniformLocationARB(program, "cpuVar");

// Der uniform-Variable wird der Wert 3,5644 zugewiesen
gl.glUniform1fARB(location, 3.5644f);

// Zeichnet ein Rechteck durch Angabe von Breite, Hoehe und Position
gl.glBegin(GL_QUADS);
  // Angaben zur Groesse und Lage des Rechteck.
  ⋮

// Bezeichnet das Ende der Programmausfuehrung
gl.glEnd();

// Das Ausfuehrung des Programms wird beendet
gl.glUseProgramObjectARB(0);
```

Abbildung 2.18: Pseudo-Code zum Starten eines Fragment-Programms.

Programmausführung Die Ausführung des Programms erfolgt durch das Zeichnen eines Rechtecks mit Hilfe des OpenGL-Befehls `glBegin(GL_QUADS)`. Anzugeben sind dazu vor allem die Parameter Breite und Höhe. Breite und Höhe gibt in unserem Fall die Ausdehnung in Pixeln im Frame-Buffer an. Es wird dann für jedes Pixel des Frame-Buffers basierend auf dem Fragment-Programm ein Fragment-Shader erzeugt. Diese werden dann in den Fragment-Prozessoren ausgeführt. Da diese auf der Grafikkarte mehrfach vorhanden sind, können die Fragment-Programme stellenweise echt parallel zur Ausführung kommen. Abbildung 2.18 zeigt beispielhaft, wie Fragment-Programme gestartet werden. Nun wird auch deutlich, dass ein Fragment-Programm den „Rahmen“ für eine Ausführung vorgibt und die Fragment-Shader die einzelnen Einheiten bilden, die dann tatsächlich ausgeführt werden. Im Weiteren wird ein Fragment-Programm kurzer Hand als *Kernel* und ein Fragment-Shader als *Shader* bezeichnet. Ein wichtiger Punkt ist ebenfalls, dass die Größe des zu zeichnenden Rechtecks der Größe der Textur-Sets im Frame-Buffer entspricht.
Werden Kernels mehrmals benutzt, müssen sie nicht zwangsläufig jedes Mal neu compiliert und gelinkt werden. Dies genügt einmal zu Beginn der gesamten Anwendung. Anschließend muss lediglich der Kernel durch Zeichnen eines Rechtecks neu gestartet werden (siehe Abbildung 2.18).

```
// Variante 1: Zum Compile-Zeitpunkt festgelegte Variable
const float var1 = 1.0;

// Variante 2: Direkt von CPU-Seite setzbar: uniform-Variable
uniform float cpuVar;

// Variante 3: Ebenfalls uniform: Setzen der Textur-ID auf CPU-Seite
// Die Ausmaße entsprechen in diesem Fall denen des
// Textur-Sets, was als Ziel-Speicher fuer die Werte dient.
uniform texture2DRect texture1;

// Einstiegspunkt fuer GLSL-Programme
void main()
{
        // Position im Frame-Buffer
        vec2 xy          = gl_TexCoord[0].xy;

        // ID der betroffenen Textur (texture1) wird interpretiert
        // und in entsprechende Textur gelesen
        vec4 tmp         = texture2DRect(texture1, xy);

        // Irgend eine 4-fache Float-Variable
        vec4 vector      = vec4(1.0, 2.0, 3.0, 4.0);

        // Die vier an das FBO gebundenen Texturen..
        gl_FragData[0]   = tmp + vector;
        gl_FragData[1]   = tmp * vector;
        gl_FragData[2]   = vector;
        gl_FragData[3]   = vec4(0.0, 1.1, 2.2, 3.3);
}
```

Abbildung 2.19: Beispiel eines einfachen Fragment-Shader, in GLSL geschrieben. Mit `gl_FragData[*]` werden die an das FBO gebundenen Texturen adressiert. Jede dieser Texturen nimmt bis zu vier Werte auf. Somit können bei vier Texturen maximal 16 Werte in das FBO an Stelle (x, y) geschrieben werden.

Daten- und Parameterübergabe Die Steuerung des Programm-Ablaufs lässt sich durch Variablen umsetzen. Eine Möglichkeit ist es, diese direkt in den Code einzutragen (siehe Variante 1 Abbildung 2.19). Somit werden sie gemeinsam mit dem restlichen Code compiliert und gelinkt. Diese Methode bietet sich jedoch nur dann an, wenn sich diese Variablen nach Erzeugung der Kernel nicht mehr ändern. Würden sich die Inhalte dieser Variablen ändern, müssten die Kernel neu compiliert und gelinkt werden.
Um ein erneutes Compilieren zu verhindern, gibt es die Möglichkeit, sogenannte `uniform` Variablen zu nutzen (Variante 2 Abbildung 2.19). Sie ermöglichen das Setzen von Werten in Kerneln, die bereits compiliert und gelinkt sind, also Programme, die bereits auf die

GPU transferiert worden sind. Sie gelten dann für alle einzelnen Shader. Jedoch ist dieser Weg nur einseitig. Das heißt, werden diese Werte auf GPU-Seite geändert, werden sie nicht an die CPU zurückgegeben.
Diese Methode eignet sich gut, sollen globale Variablen für Shader gesetzt werden. Müssen aber in Abhängigkeit von vorher ausgerechneten Werten bestimmte Daten gelesen werden, bieten sich hier eher Texturen an. Jedes einzelne Fragment kann einen eigenen Kontrollfluss besitzen und somit unterschiedliche Werte benötigen. In diesem Fall sind Texturen besser geeignet (Variante 3 Abbildung 2.19). Für die Ping-Pong-Technik ist dies sogar notwendig.
Einem startbereiten Kernel können Texturen, die bereits auf der GPU angelegt worden sind, zugewiesen werden. Hier spielt die Größe und Anzahl der Texturen eine nicht so wichtige Rolle. Sogar die Formatierung kann sich unterscheiden.

2.3.2 Bestandteile

Nach der Einführung zum Programmieren auf der GPU und dem Raytracing im Allgemeinen soll nun gezeigt werden, wie beides miteinander verknüpft wurde. Das Konzept von Huwe[8] dient als Ausgangspunkt und sollte daher an dieser Stelle kurz verdeutlicht werden. Dazu werden wir zunächst den Ablauf grob skizzieren und anschließend die einzelnen Elemente genauer betrachten.
Im Teil zur GLSL-Einführung wurde bereits erwähnt, dass Programme in den Fragment-Prozessoren der GPU ausgeführt werden. Diese Programme werden in *Kernel* verpackt und in den Shadern ausgeführt. Die Kernel bilden die einzelnen Prozess-Elemente von Sunshine und werden je nach Bedarf aufgerufen.

2.3.2.1 Grundlegendes Konzept der Pipeline von Sunshine

Die einzelnen Elemente nach Huwe[8] lauteten `InitKernel`, `GenRayKernel` sowie `TraceRayKernel`. Abbildung 2.20 zeigt den Ablauf der Render-Pipeline von Sunhine.
Nachfolgend sollen nun die Aufgaben der einzelnen Kernel beleuchtet werden.

InitKernel Wie in Abschnitt 2.2.2 erwähnt, wird beim Path Tracing versucht, mit Hilfe zufälliger Werte die Rendergleichung numerisch zu lösen. Daher wird ein Mechanismus benötigt, der zufällige Werte liefert. Um Muster im Ergebnis zu verhindern, benötigt jeder Pfad einen eigenen Start-Seed. Dieser wird in diesem Kernel erzeugt.

GenRayKernel Nachdem der PRNG für jeden einzelnen Bildpunkt, voneinander unabhängig, initialisiert wurde, können nun die Strahlen für das Raytracing erzeugt werden. Der Ausgangspunkt ist eine virtuelle Kamera in der Szene. Dabei wird Ursprung und Richtung festgelegt und in der Textur gespeichert.
Um das Supersampling durchzuführen, werden die Richtungen in einem Bildpunkt leicht variiert.

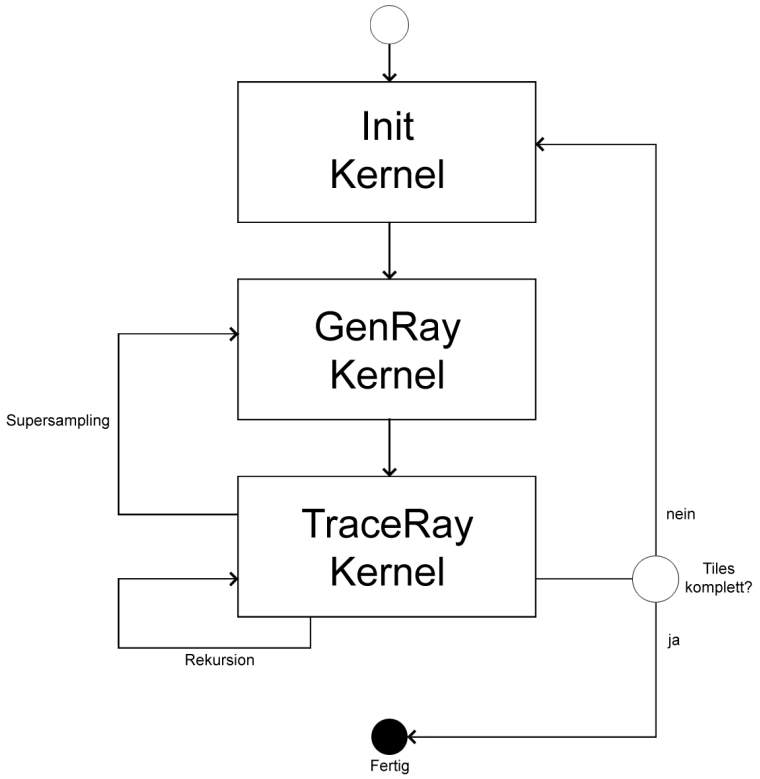


Abbildung 2.20: Die Abbildung zeigt den Ablauf des Renderprozesses nach Huwe[8].

TraceRayKernel Hier werden die erzeugten Strahlen mit der Szene getestet und nach Schnittpunkten gesucht. Werden Schnittpunkte gefunden, muss anschließend festgestellt werden, ob besagter Schnittpunkt bezüglicher einer Lichtquelle beleuchtet wird oder nicht. Ist der Schnittpunkt nicht im Schatten, wird das lokale Beleuchtungssystem evaluiert. Dies ist dann der Beleuchtungsbeitrag des Strahls in Bildpunkt (x, y). Anschließend kann der Sekundärstrahl erzeugt werden.
Der Beleuchtungsbeitrag wird als Farbe auf die Farbe an dem Bildpunkt addiert und zusammen mit dem Sekundärstrahl in der Textur gespeichert. Ist die maximale Rekursionstiefe noch nicht erreicht, wird dieser Kernel anschließend direkt nochmal ausgeführt. Andernfalls wird der `GenRayKernel` ausgeführt, um neue Primärstrahlen zu erzeugen.
Aufgrund von Speicherengpässen auf Grafikkarten wird ein zu erstellendes Bild in Kacheln, sogenannte *Tiles*, eingeteilt. Diese sind sozusagen kleine Teilbilder, die am Ende zusammengesetzt das fertige Bild ergeben.

2.3.2.2 Erweiterung der Pipeline

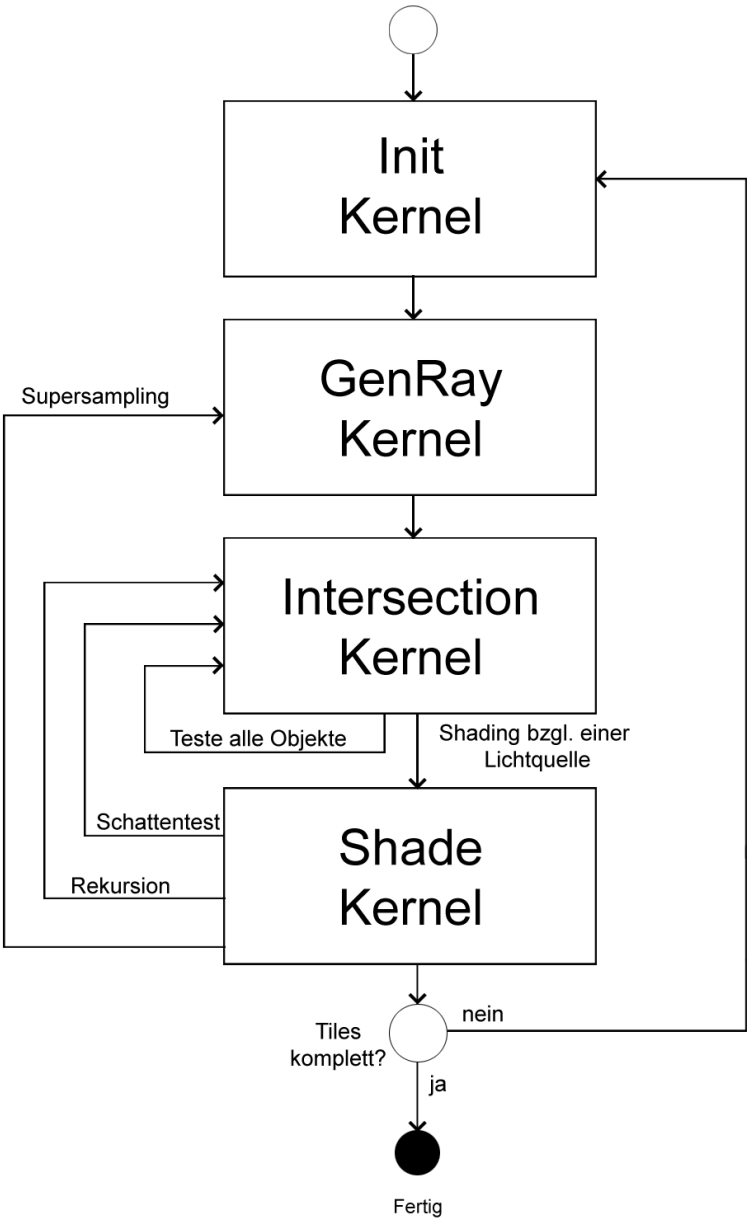


Abbildung 2.21: Modifizierte Pipeline: Der `TraceRayKernel` wurde in zwei Kernel (`IntersectionKernel` und `ShadeKernel`) aufgeteilt.

Die recht monolithische Konzipierung hatte den Nachteil, dass Sunshine zunächst nur auf NVIDIA-Grafikkarten lief, jedoch nicht auf älteren Grafikkarten von ATI. Außerdem konnten nur Szenen mit wenigen Objekten unterstützt werden. Das Problem ist die

Ausführungszeit eines Kernels, da im `TraceRayKernel` jeder Strahl mit allen Objekten der Szene getestet wurde. Dies geschah in einer Schleife; deren Länge hing dabei direkt von der Anzahl der Objekte in der Szene ab. Eine höhere Anzahl an Schleifendurchläufen erhöht jedoch auch die Anzahl der Instruktionen, die ausgeführt werden. Diese ist jedoch gerade bei älteren Modellen beschränkt.
Nicht nur die Grenze einer maximalen Anzahl an Instruktionen stellt ein Problem dar. Ebenso spielt die Dauer einer Ausführung eine erhebliche Rolle. Im FAQ von NVIDIA ist vor allem im Betrieb unter Windows von sogenannten *Watchdog-Timern* die Rede[10]. Diese alarmieren das Betriebssystem, wenn die Grafikkarte mehr als 5 Sekunden Ausführungszeit für einen Kernel benötigt. Das Betriebssystem reagiert darauf mit einer Fehlermeldung oder gar einen kompletten Absturz.
Die Skalierbarkeit der Szene und die plattformübergreifende Unterstützung des Raytracers sind der Grund dafür, den monolithischen `TraceRayKernel` in zwei Kernel aufzuteilen. Somit entstand also ein Kernel extra zum Finden der Schnittpunkte (`IntersectionKernel`) sowie ein Kernel zum Berechnen der lokalen Beleuchtungssysteme und zum Erzeugen der Sekundärstrahlen (`ShadeKernel`).
Zusätzlich wurde der `IntersectionKernel` so gestaltet, dass dieser nun unterbrechbar ist. Das heißt, dass er nun nach einem Aufruf nicht mehr alle Objekte in einem Durchgang testen muss. Dies war bei der vorläufigen Variante der Fall: Die Ausführungszeit stieg unter Umständen auf über 5 Sekunden an, beziehungsweise das Instruktionslimit wurde erreicht.
Damit weder das Instruktionslimit noch die maximale Ausführungszeit überschritten werden, bekommt der Kernel als Parameter die Anzahl der Objekte sowie einen Startwert übergeben, mit denen er die Objekt-Schleife beginnen beziehungsweise fortsetzen soll. Somit kann nun optional festgelegt werden, wie viele Objekte pro Kernel-Aufruf getestet werden. Dieser Parameterwert kann nun so gewählt werden, dass das Erreichen der Limitierungen nach Möglichkeit verhindert wird.
Durch das Aufspalten von `TraceRayKernel` reichte jedoch eine einzige Textur-Set-Gruppe nicht mehr aus. Die Informationen, in Abbildung 2.22 dargestellt, die benötigt werden, machten eine weitere Textur-Gruppe und eine Umordnung notwendig.
Die veränderte Pipeline, die auch gleichzeitig die Grundlage des spektralen Raytracers bildet, ist in Abbildung 2.21 zu sehen. Mit dieser Modifizierung ist es nun möglich, neben einer vergrößerten Basis an unterstützten Grafikkarten auch größere Szenen zu rendern.

2.3.3 Datenstruktur in FBO-Textur

Wie bereits erwähnt, ist die Anzahl der Werte, die am Ende eines Shaders gespeichert werden können, auf 16 begrenzt. Dies macht es notwendig, sich Gedanken über die Auswahl der zu speichernden Informationen zu machen. Dies geschieht, in dem ein aus vier Texturen bestehendes Textur-Set unterteilt wird. Diese Aufteilung wird des Weiteren als eine Textur-Set-Gruppe bezeichnet. Huwe[8] kam mit nur einer Textur-Set-Gruppe

[10] Entnommen aus `http://forums.nvidia.com/index.php?showtopic=84440` Punkt 41. Letzter Zugriff 25.03.2009

aus, um alle benötigten Informationen während der verschiedenen Kernel-Ausführungen zu speichern.
Auf Grund der Umgestaltung der Pipeline reichte jedoch eine Textur-Set-Gruppe nicht mehr aus, so dass eine weitere hinzugefügt werden musste. Abbildung 2.22 zeigt die Aufteilung der Informationen in den Textur-Set-Gruppen. Während der Ausführung des `IntersectionKernel` wird nur in die `SchnittTestDaten` Textur-Set-Gruppe geschrieben. Bei allen anderen Kernel-Aufrufen ist das Schreibziel `BildAusgabe`.

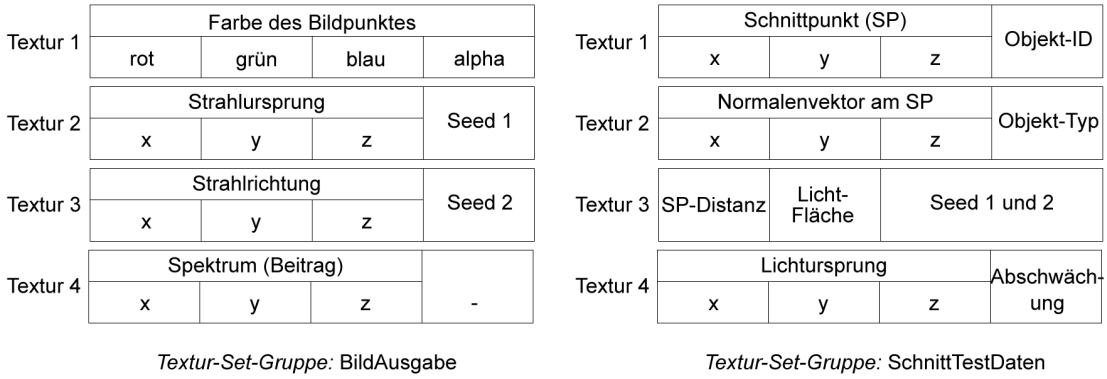


Abbildung 2.22: Die Abbildung zeigt die Aufteilung der einzelnen Informationen in den jeweiligen Textur-Set-Gruppen. Von jeder Textur-Set-Gruppe werden für das Ping-Pong mindestens zwei Vertreter benötigt. Somit sind insgesamt 4 Textur-Sets beim Rendern involviert. Seed 1 und 2 dienen der Zustands-Speicherung des Zufallszahlen-Generators.

2.3.4 Ablaufsteuerung

Um die einzelnen Kernel zum richtigen Zeitpunkt zu starten, ist eine zentrale Verwaltung notwendig, die die Reihenfolge der Kernel-Aufrufe steuert. Da ältere Grafikkarten nur einen Kernel zeitgleich ausführen können, muss nach Beendigung der Berechnungen auf der GPU die CPU entscheiden, was als nächstes geschehen soll.
Mit JOGL erfolgt das Ausführen von OpenGL-Befehlen nur über einen *AWT-Thread*[11]. Wird nun ein OpenGL-Befehl in Java außerhalb dieses Threads abgesetzt, muss erst ein Prozesswechsel erfolgen, was Zeit kostet. Dies verhindert man, indem man den Renderer in einem AWT-Thread ausführt, da die OpenGL-Befehle direkt an den Treiber weitergeleitet werden.
Aus diesem Grund wurde das Prinzip einer Ereignisschleife umgesetzt. Das Grundprinzip soll nun kurz erläutert werden: Der gesamte Renderer ist als ein Zustands-Automat aufgebaut. Dieser hat eine innere Schleife, die so lange durchlaufen wird, bis ein Endzu-

[11] Das Kürzel AWT steht für *Abstract Window Toolkit* und stellt eine plattformunabhängige API für das Erzeugen von GUI-Elementen dar.

stand erreicht wird. Zu einem aktuellen Zustand werden bestimmte Kernel ausgeführt und eventuell Daten von der CPU auf der GPU bereitgestellt. Ist ein Kernel ausgeführt und die Kontrolle wird an die Steuerung auf CPU-Seite zurückgegeben, ändert sich der Zustand. Somit wird im nächsten Durchgang eventuell ein anderer Kernel ausgeführt.
Eine schematische Darstellung der Ablaufsteuerung zeigt Abbildung 2.21. Zu Beginn des Renderporzesses werden die Kernel erstellt. Dazu werden die dazugehörigen Quellcode-Dateien den jeweiligen Kernel zugewiesen und kompiliert. Diese sind nun präpariert und stehen zur Ausführung bereit. Konstanten, wie zum Beispiel die Anzahl der Objekte oder die Position der Kamera, werden direkt in den Quell-Code mit eingebunden.
Die Szenedaten wurden zu diesem Zeitpunkt bereits „eingesammelt“ und in einer Textur auf die Grafikkarte transferiert. Die Vorbereitungsphase ist nun abgeschlossen und der erste Kernel (`InitKernel`) kann nun ausgeführt werden.
Wie auch bei Huwe[8] wird im `InitKernel` der initiale Zustand für den PRNG erzeugt. Anschließend können im `GenRayKernel` die Primärstrahlen für jeden Bildpunkt generiert werden.
Mit den erzeugten Strahlen kann nun im `IntersectionKernel` nach Schnittpunkten gesucht werden. Dabei kann der Kernel, wie oben besprochen, unterbrochen werden. Dies geschieht dann, wenn die Anzahl der Objekte pro Schleifendurchlauf geringer ist als die Gesamtanzahl der Objekte in der Szene. Der Kernel wird also in einer Schleife solange aufgerufen, bis alle Objekte einmal getestet wurden.
Außer den Daten, die in Texturen oder direkt im Code als Variablen deklariert stehen, gehen alle Informationen verloren. Um nun den Start der Objekt-Schleife festzulegen, werden uniform-Variablen genutzt. Auf CPU-Seite wird also mit jedem `Intersection-Kernel` mitgezählt, wie viele Objekte bereits getestet, wurden. Sind alle Objekte getestet kann, ebenfalls mit dem `IntersectionKernel`, ein Schattentest durchgeführt werden.
Im Schattentest wird für einen gefundenen Schnittpunkt überprüft, ob dieser bezüglich einer Lichtquelle von einem Objekt verdeckt wird. Dies entspricht ebenfalls der Aufgabe des Findens eines Schnittpunktes entlang eines Strahls.
Das Ergebnis des Schattentest wird anschließend ebenfalls in der Textur festgehalten. Im Feld `Abschwächung` der Textur-Set-Gruppe `SchnittTestDaten` (siehe Abbildung 2.22) wird durch den Wert 0.0 ausgedrückt, dass Lichtquelle x den betroffenen Punkt NICHT beleuchtet. Der Punkt liegt also im Schatten.
Mit den Information aus dem Schnittpunkt-Test kann nun das Shading durchgeführt werden. Existiert ein Schnittpunkt entlang eines Strahls und liegt dieser bezüglich einer Lichtquelle nicht im Schatten, wird das lokale Belechtungssystem ausgerechnet und ein eventueller Beleuchtungsbeitrag auf den Bildpunkt addiert, von wo aus der Strahl gestartet ist. Anschließend wird eventuell ein Sekundärstrahl erzeugt und zusammen mit dem Beleuchtungsbeitrag in das Textur-Set der Gruppe `BildAusgabe` eingetragen.
Ist der `ShadeKernel` fertig, übernimmt die CPU wieder die Kontrolle und überprüft, ob eine vorher angegebene Rekursionstiefe erreicht wurde. Ist dies nicht der Fall, wird die aktuelle Rekursionstiefe um eins erhöht und der `IntersectionKernel` direkt ausgeführt. Somit werden nun Schnittpunkt-Tests mit den eben erzeugten Sekundärstrahlen durchgeführt. Anschließend erfolgt wiederum der Schattentest und das Shading.
Ist die maximale Rekursionstiefe erreicht, wird überprüft, ob genug Strahlen durch einen

Bildpunkt gesendet wurden. Ist dem nicht so, wird der dafür zuständige Zähler um eins erhöht und es werden neue Primärstrahlen erzeugt, die auch den gleichen Weg durch die Pipeline nehmen wie die zuerst erzeugten Primärstrahlen. Das wiederholte Erzeugen von Strahlen im gleichen Bildpunkt wird auch als Supersampling bezeichnet.
Wie bereits erwähnt, wird ein zu erstellendes Bild zu Anfang in kleine Teilbilder (Tiles) zerlegt. Für jedes Tile wird nun also die komplette Pipeline durchlaufen, bis dann die maximale Anzahl pro Strahl im Bildpunkt des aktuellen Tiles erreicht wurde. Darauf folgt eine Verschiebung des aktuellen Tiles und die Strahlverfolgung beginnt wieder im `InitKernel`. Dies wiederholt sich, bis alle Tiles fertiggestellt wurden.

2.3.5 Repräsentation der Szene-Daten

Für das Raytracing werden Informationen über die einzelnen Objekte, wie zum Beispiel Größe, Lage oder Oberflächenfarbe benötigt. Diese Informationen werden in einer dafür vorgesehenen Textur, der Szenen-Textur, abgelegt. Deren Aufbau soll hier nun kurz erläutert werden, da dies für den konzeptionellen Teil der Materialorganisation wichtig ist.
Die Schnittpunktberechnungen werden in so genannten lokalen Koordinaten-Systemen durchgeführt, was die eigentliche Berechnung der Schnittpunkt mit den Objekten vereinfacht. Als Beispiel soll dies kurz an einer Kugel verdeutlicht werden. Zunächst betrachten wir die allgemeine Form der Gleichung einer Kugel mit Radius r und einem Mittelpunkt (a, b, c):

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2. \tag{2.50}$$

Die Gleichung vereinfacht sich, wenn die betrachtete Kugel im Koordinatenursprung liegt, also $a = b = c = 0$, und $r = 1$ ist. In diesem Fall spricht man von einer sogenannten Einheitskugel. Die vereinfachte Form der Gleichung 2.50 lautet demnach:

$$x^2 + y^2 + z^2 = 1. \tag{2.51}$$

Um das lokale Koordinatensystem nutzen zu können, werden Transformationsmatrizen benötigt. Diese definieren neben der Lage im Raum auch die Größe der Objekte. Soll nun mit Richtungsvektoren und Punkten aus dem globalen Koordinatensystem im lokalen System gearbeitet werden, müssen diese mit Hilfe einer Transformationsmatrix überführt werden[12]. Berechnete Punkte und Vektoren müssen ihrerseits wiederum in das globale System zurück transformiert werden. Dies geschieht mit einer *invertierten* Transformationsmatrix.
Solche Matrizen sind normalerweise mit 4x4, also 16 Werten, beschrieben. Da mit homogenen Koordinaten gearbeitet wird, kann die untere Zeile weggelassen werden. Gemäß der Formatierung von vier Float-Werten pro Pixel, kann somit eine Matrix, aus 12 Werten bestehend, in drei Pixel einer Textur abgelegt werden. Da die Transformation in beide Richtungen notwendig ist, wird auch die invertierte Matrix, ebenso mit 12 Werten, benötigt.

[12] Für Normalenvektoren muss die Transformationsmatrix transponiert werden, damit die Normale senkrecht auf der Oberfläche steht.

Des Weiteren besitzt jedes Objekt eine Beschreibung der Oberfläche. Im einfachsten Fall ist dies eine RGBA-Farbe, also eine RGB-Farbe mit einem vierten Kanal für die Transparenz. Diese vier Werte können in ein einziges Pixel abgelegt werden. Für komplexere Oberflächen-Beschreibungen wird auch das Farbfeld genutzt. Im Falle von Huwe's Path Tracer handelt es sich um das Phong-Modell. Im Farbfeld wird eine Adresse hinterlegt, die auf eine konkrete Beschreibung der Oberfläche verweist.
Jede unterstützte Primitive wird also mindestens durch zwei Matrizen und eine RGBA-Farbe in der Szene repräsentiert. Dies entspricht 28 Werten, oder 7 Pixeln in der Textur. Je nach Art der Primitive sind dies gegebenfalls mehr Werte. Genaueres ist aus Huwe[8] zu entnehmen.

3 Konzept zur Umsetzung

Aufbauend auf der Arbeit von Huwe[8] und den Vorüberlegungen in Kapitel 2 werden nun zwei Konzepte vorgestellt. Das erste Konzept stellt eine Modifizierung der Ablaufsteuerung des konventionellen Path Tracers dar. Mit dem zweiten Konzept soll eine Lösung präsentiert werden, die die Organisation komplexer Oberflächen-Beschreibungen ermöglicht.

3.1 Konzept der Ablaufsteuerung

Die allgemeine Form der Rendergleichung (siehe Gleichung 2.26) verbarg bisher den Bezug zur Wellenlängenabhängigkeit. Dies hat den Grund, dass die Gleichung häufig bezüglich RGB-basierter Verfahren angegeben wird. Wird die Abhängigkeit von der Wellenlänge, also λ, berücksichtigt, hat die Gleichung die Form:

$$L_o(p, \omega_o, \lambda) = L_e(p, \omega_o, \lambda) + \int_{S^2} f(p, \omega_o, \omega_i, \lambda)\ L_i(p, \omega_i, \lambda)\ |cos\theta_i|\ d\omega_i. \tag{3.1}$$

Dies bedeutet, dass von nun an alle zur Lösung der Rendergleichung beteiligten Terme bezüglich einer Wellenlänge λ angegeben werden. Somit ist die Beleuchtung $L_o(p, \omega_o, \lambda)$ für einen Bildpunkt P_{xy} ebenfalls eine spektrale Verteilung von Intensitäten.
Würde nun über alle Wellenlängen von $380 - 780nm$ in einem Intervall von $1nm$-Schritten integriert werden, müsste jeder Bildpunkt P_{xy} 400 Werte aufnehmen. Aufgrund der Einschränkungen, auf die bereits im Abschnitt 2.3 eingegangen wurde, können in den FBO nach der Ausführung eines Kernels nur 16 Werte pro Bildpunkt geschrieben werden. Die erste Frage, die sich dabei stellt: Wie könnten 400 Werte pro Bildpunkt aufgenommen werden, wenn nur Platz für 16 Werte ist?
Zur Lösung dieses Problems kommt die Erkenntnis aus Abschnitt 2.1 zum Tragen: Am Ende muss immer eine Umwandlung in einen für den Bildschirm darstellbaren Farbraum durchgeführt werden, daher ist es nicht zwingend notwendig das gesamte Spektrum für jeden Bildpunkt zu kennen. Hier reicht es völlig aus, das Spektrum in einen Farbraum zu überführen.
Jede Intensität $L_o(p, \omega_o, \lambda)$ wird in eine XYZ-Farbe umgewandelt und einzeln zu der resultierenden Farbe des Bildpunktes aufaddiert. So muss jeder Bildpunkt lediglich die drei Primärvalenzen der XYZ-Farbe mitführen. Der Vorteil ist, dass in dieser Form, unabhängig von der Schrittweite, mit der über das Spektrum integriert wird, immer nur drei Werte pro Bildpunkt für die bisher berechneten Farbinformationen mitzuführen sind.

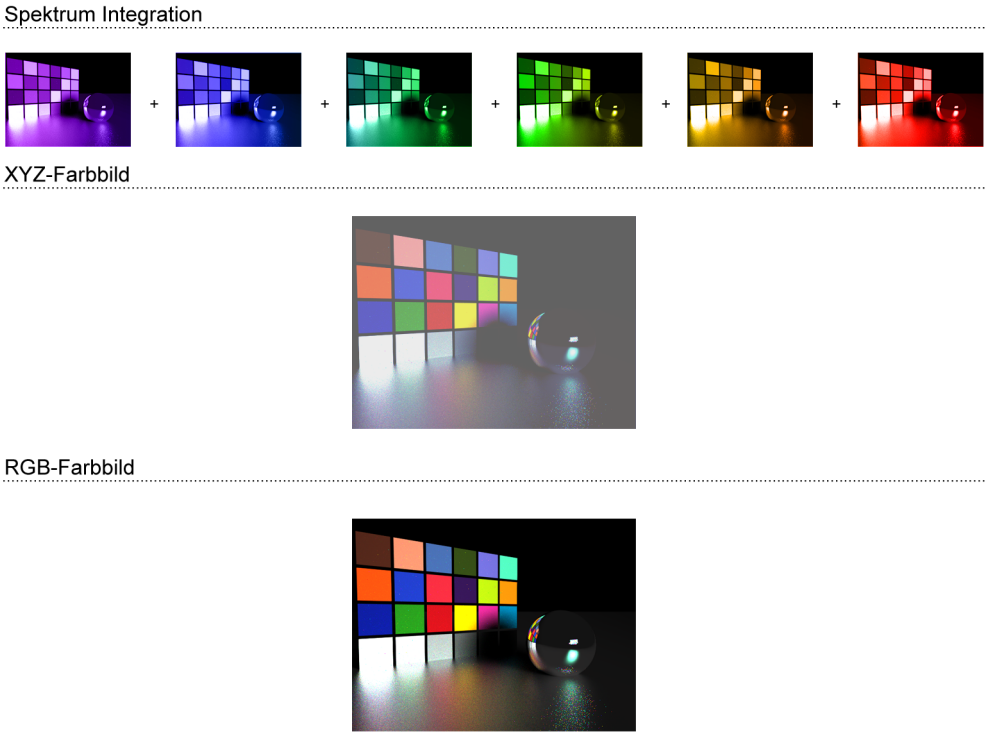


Abbildung 3.1: Prinzip der Integration über das Spektrum: Alle Berechnungen werden bezüglich einer Wellenlänge durchgeführt und zu einem XYZ-Farbwert hinzuaddiert. So entsteht nach der kompletten Integration über alle λ des Spektrums ein XYZ-Farbbild, was anschließend in ein RGB-Bild umgewandelt wird.

Die Integration kann auch vereinfacht als Summengleichung dargestellt werden (siehe Abschnitt 2.1.2). Das heißt, die Repräsentation einer XYZ-Farbe im Bildpunkt P_{xy} erfolgt durch die schrittweise Multiplikation einer Intensität aus L_o mit je einer der Funktionen $\overline{x}$, $\overline{y}$ oder $\overline{z}$. Die so entstehenden Werte für X, Y und Z werden auf die Farbe im Bildpunkt addiert. Abschließend kann dann eine Umrechnung in den RGB-Farbraum erfolgen. Abbildung 3.1 soll dieses Prinzip veranschaulichen: Die Evaluierung der BSDFs geschieht immer bezüglich einer Wellenlänge. Nach dem Erreichen der gewünschten Rekursionstiefe und Anzahl der Samples ist L_o, mit der die Kamera auf dem Bildpunkt beleuchtet wird, bekannt. Dieser Wert wird anschließend in eine XYZ-Farbe umgewandelt und für den Bildpunkt aufsummiert, sodass sukzessive ein XYZ-Farbbild entsteht. Ist das gesamte Spektrum betrachtet, erfolgt abschließend eine Transformation des Bildes in den RGB-Farbraum. Dies entspricht dem in in Sun et al.[28] vorgeschlagen Ansatz, dass das Ergebnis eines spektralen Rendervorgangs ein XYZ-Farbbild sein sollte, da dieses dann sehr leicht in andere Farbräume transformiert werden kann.

Die Integration des Spektrums beginnt mit der unteren Grenze des angegebenen Bereichs und endet bei der oberen Grenze. Der Bereich des Spektrums und die Größe der Schritte, mit der das Spektrum durchschritten werden soll, muss über die Benutzeroberfläche einstellbar sein. Im Falle eines Bereichs von $380 - 780nm$ heißt das, dass im blauen Wellenlängenbereich mit Integration begonnen wird und im roten Wellenlängenbereich endet. In Abbildung 3.1 ist dieser Sachverhalt angedeutet: Die Spektrale Integration beginnt mit blauen Licht und erzeugt zunächst blaue Bilder. Mit zunehmender Wellenlänge werden schlussendlich Bilder im roten Wellenlängenbereich erstellt.
Für die Pipeline bedeutet dies, dass sie um eine weitere äußere Schleife erweitert werden muss. Da die Berechnungen immer nur für eine konkrete Wellenlänge durchgeführt werden, muss die Pipeline so lange mit vorgegebener Schrittweite durchlaufen werden, bis der ganze Bereich des Spektrums erfasst wurde.

3.1.1 Modifizierung

Die in Abbildung 2.21 dargestellte Pipeline wird nun im Wesentlichen um zwei Komponenten erweitert. Die rot eingefärbten Erweiterungen der modifizierten Pipeline sind in Abbildung 3.2 zu erkennen.
Die Integration des Spektrums an jeden Bildpunkt geschieht in der Form, dass mit konstanter Schrittweite „Proben“ vom betrachteten Spektrum genommen werden. Anders ausgedrückt: Es werden die Beleuchtungsbeiträge für bestimmte λ-Werte ermittelt. Das Entnehmen von Proben wurde bereits mit dem Begriff des Sampling umschrieben, weshalb das Entnehmen von λ-Proben des Weiteren auch als λ- oder auch *lambda*-Sampling bezeichnet wird.
Das *lambda*-Sampling ist im Wesentlichen als eine „äußere“ Schleife zu betrachten: Es wird für jede Wellenlänge ein Pfad in der Kamera erzeugt und rekursiv verfolgt. Um Bildrauschen zu vermindern, wird ein Supersampling durchgeführt. Das heißt, es werden für die gleiche Wellenlänge verschiedene Pfade im Bildpunkt erzeugt. Ist das Supersampling und die Rekursion beendet, werden wieder Pfade erzeugt und bezüglich einer neuen Wellenlänge verfolgt.
Somit ist nun verdeutlicht, dass die Kernel immer für eine bestimmte Wellenlänge ausgeführt werden. Die Verwaltung übermittelt an die Kernel die aktuelle Wellenlänge, so dass diese global für alle auszuführenden Fragment-Programme gelten. Dies erleichtert des Weiteren die Bereitstellung von spektralen Informationen, da alle Fragmente auf gleicher λ-Basis arbeiten.
Neben dem λ-Sampling wird die Pipeline um einen Kernel erweitert, der die Konvertierung von XYZ- in RGB-Farben durchführt. Dies geschieht im `ConversionKernel` nach dem über alle Wellenlängen des Spektrums integriert wurde.
Bis auf den `ShadingKernel` haben sich die restlichen Komponenten gegenüber der RGB-basierten Variante von Sunshine nicht verändert. Jedoch wurde die Verwaltung auf der CPU um einige Elemente erweitert.

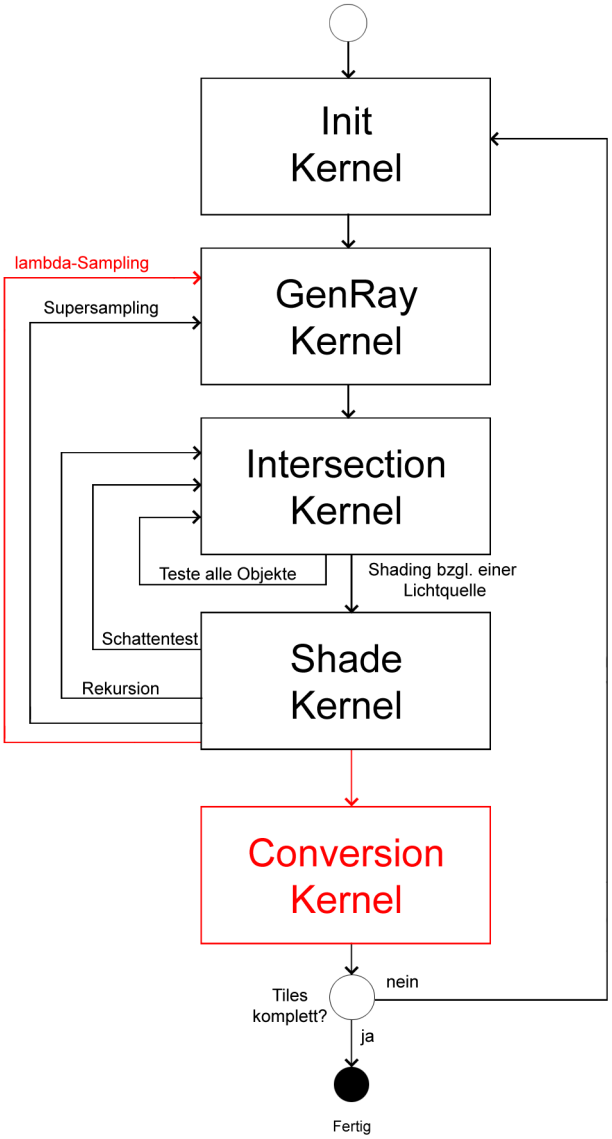


Abbildung 3.2: Modifizierte Pipeline: Die Integration wird mittels *lambda*-Sampling durchgeführt, wobei ein XYZ-Bild entsteht. Dieses wird anschließend im `ConversionKernel` in ein RGB-Bild konvertiert.

3.1.2 Vorbereitung

Bevor mit dem `InitKernel` der eigentliche Renderprozess gestartet wird, müssen die zum Rendern benötigten Daten „eingesammelt“ und in Texturen verpackt werden. Wie in Abschnitt 2.3.5 erwähnt, werden zu allen Objekten zunächst die beiden Transformationsmatrizen und eventuell zusätzliche geometrische Informationen erfasst und in der

Szenen-Textur hinterlegt. Die Beschreibung der Oberfläche wird ebenfalls in diese Textur eingetragen.
Die Oberflächen der Objekte, die mit Hilfe von Spektren beschrieben sind, müssen hier nun „vorbereitet“ in einer Textur verpackt und auf der GPU zur Verfügung gestellt werden. Mit einer angegebenen oberen und unteren Grenze des zu integrierenden Spektrums und der Schrittweite steht somit zu Beginn bereits fest, welche Wellenlängen betrachtet werden.
Diese Erkenntnis lässt sich nun nutzen, um auch nur die Daten auf der GPU bereitzustellen, die auch wirklich gebraucht werden. Dies ist ein großer Vorteil, vor allem für ältere Grafikkarten mit kleinem Speicher.
Nachdem alle Objekte eingesammelt wurden, ist auch bekannt, welche Spektren benötigt werden. Diese werden nun eventuell aus einer Datei gelesen, die betroffenen Werte entnommen und in einen Puffer hinterlegt. Abhängig von der Anzahl der Werte wird zunächst eine Textur auf der GPU erzeugt. Anschließend kann dann der Inhalt des Puffers auf die GPU transferiert werden.
Wie genau die spektralen Daten präpariert werden, wird in Abschnitt 3.2 besprochen. Damit ist die Vorbereitung abgeschlossen und der eigentliche Render-Vorgang kann beginnen.

3.1.3 Renderprozess

Die Hauptaufgabe der Verwaltung während des Renderprozess ist das Organisieren der Ablaufsteuerung. Die Reihenfolge der Kernel-Aufrufe entspricht der in Abschnitt 2.3.2.2 beschriebenen. Zunächst wird das zu erstellende Gesamtbild in möglichst gleich große quadratische Kacheln zerteilt. Die Größe der Kachel ist ebenfalls optional einstellbar. Für eine Kachel wird nun in jedem Bildpunkt ein Strahl erzeugt und rekursiv durch die Szene verfolgt bis eine vorher angegebene Rekursionstiefe erreicht wurde. Somit entsteht ein Pfad von der Kamera durch die Szene. Hat der Pfad die gewünschte Anzahl an Schnittpunkten, wird ein neuer Pfad in der Kamera erzeugt, sofern die vorher angegebene Anzahl an Pfaden durch den Bildpunkt (Supersampling) noch nicht erreicht wurde.
Ab dieser Stelle unterscheidet sich das spektrale vom konventionellen Konzept: Anstelle der Berechnung der nächsten Kachel, wird nun der Vorgang beginnend mit der Strahlenerzeugung für dieselbe Kachel, jedoch für eine andere Wellenlänge, wiederholt. Wie bereits zuvor erwähnt, steht bereits in der Vorbereitungsphase fest, welche Wellenlängen konkret betrachtet werden. Die Verwaltung hat nun die Aufgabe, sicherzustellen, dass alle angegebenen Wellenlängen betrachtet werden. Das bedeutet, dass für alle angegebenen Wellenlängen mindestens ein Pfad erzeugt und durch die Szene verfolgt werden muss.
Bleibt also festzuhalten, dass das XYZ-Farbbild einer Kachel dann komplett ist, wenn folgende drei Bedingungen in der Reihenfolge erfüllt sind:

- Jeder Pfad wurde bis zur gewünschten Rekusionstiefe verfolgt,
- die Anzahl der Pfade pro Bildpunkt einer Wellenlänge wurde erreicht und

- alle angegebenen Wellenlängen wurden betrachtet.

Eine weitere wichtige Aufgabe ist es, Parameter, die sich zur Laufzeit ändern zur Verfügung zu stellen. Dies kann, wie in Abschnitt 2.3.1 beschrieben, mittels `uniform`-Variablen geschehen. So wird beispielsweise die Konvertierung von spektralen Intensitäten in den XYZ-Farbbraum mit Hilfe der `uniform`s realisiert: Für die Konvertierung in den XYZ-Farbraum werden die Farbanpassungsfunktionen $\overline{x}$, $\overline{y}$ und $\overline{z}$ benötigt. Technisch betrachtet, sind diese Funktionen auch spektrale Verteilungen. Zwar entsprechen die einzelnen Einträge nicht dem eigentlichen Sinn einer SPD, sondern sind vielmehr Gewichte für das Skalieren von Intensitäten, entsprechend der menschlichen Wahrnehmung, dennoch werden diese Gewichte ebenfalls wellenlängenabhängig gehalten.
Da zu einem bestimmten Zeitpunkt für alle Pfade immer nur eine Wellenlänge betrachtet wird, bieten sich `uniform`-Variablen an. So wird aus diesen drei Funktionen jeweils ein Wert für die gerade geltende Wellenlängen entnommen und auf der GPU via `uniform`-Variable zur Verfügung gestellt. Dies erspart unnötige Texturzugriffe auf Seiten der GPU, da diese Parameter von allen Fragmenten bei der Ausführung des `ShadingKernel`s benötigt werden.

3.1.4 Finalisierung

Nachdem das Lambda-Sampling abgeschlossen wurde, ist das XYZ-Bild für die aktuelle Kachel fertig. Nun kann dies in den RGB-Farbraum überführt werden. Dazu wird der `ConversionKernel` aufgerufen. Hier wird mittels linearer Transformation ein RGB-Farbbild erstellt, was nun durch einen Bildschirm darstellbar ist.
Die Kachel ist nun fertig und es kann mit der nächsten Kachel begonnen werden. Hierbei ist darauf zu achten, dass die Verwaltung alle Zähler zurücksetzt, denn es muss wieder eine Rekursion, ein Supersampling, sowie ein Lambda-Sampling stattfinden, welches durch Zähler inspiziert werden kann.
Wenn alle Kacheln komplett sind, ist das gesamte Bild ebenfalls fertig und der Renderer kann beendet werden.

3.2 Organisation von Materialdaten

Eine der Herausforderung dieser Arbeit ist es, die für die Oberflächenbeschreibung, des Weiteren als Material bezeichnet, notwendigen Daten zur Verfügung zu stellen. Da intern Farben nicht mehr länger nur als RGB-Werte repräsentiert werden, muss man sich auch Gedanken über die Art und Weise der Datenhaltung machen. Wie bereits besprochen, brauchen Spektren in Abhängigkeit ihrer Auflösung entsprechend viel Platz in einem Speicher. Sind dann noch mehrere Spektren in die Berechnungen involviert, kann der begrenzte Speicher auf der GPU zu einem Hindernis werden. Daher ist die Überlegung wichtig, wann welche Daten auf der GPU zur Verfügung stehen sollen.
Des Weiteren soll der spektrale Teil von Sunshine um einige Shader erweitert werden. Da das bisher umgesetzte Konzept eine mögliche Erweiterung erschwert, muss dazu die Organisation der verschiedenen Materialien mit überdacht werden.

In der Arbeit von Huwe[8] konnten Oberflächen durch RGBA-Farben oder Phong-Shader beschrieben werden. Da der Platz in der Szenen-Textur auf vier Werte begrenzt wurde und der Phong-Shader mehr als vier Parameter benötigt, wurde dieser direkt im Quellcode des Fragment-Programms definiert. Im Farbfeld der Szenen-Textur stand dann nur ein Eintrag, der den definierten Shader adressierte.
Der Phong-Shader selbst ist einmal als eine Funktion umgesetzt, welche als Eingabe einige Parameter erwartet. Ein konkreter Vertreter des Phong-Shaders besteht dann aus einem Funktionsaufruf. Die benötigten Parameter für den Aufruf stehen fest im Quellcode. Diese werden beim Erzeugen der jeweiligen Kernel aus der Szene entnommen und in den Quellcode geschrieben.
Somit war es möglich mehrere verschiedene beziehungsweise einen Phong-Shader für mehrere Objekte zu definieren. Das Problem ist jedoch die Begrenzung in der Größe des Kernel-Programms. Für jeden zusätzlich definierten Phong-Shader wird der Quellcode größer und kann unter Umständen bei sehr vielen Objekten mit je einem eigenen Phong-Shader zu Problemen beim Compilieren oder Starten des Programms führen.
Die Anforderungen an die Materialorganisation ist, dass die einzelnen Materialien mit unterschiedlich vielen Parametern gefüttert werden müssen. Dies ist ein wichtiger Punkt, da auf die Anordnung der einzelnen Elemente im Speicher zu achten ist.
Die grundlegende Idee ist es, Materialdaten separat von der Szenen-Textur in einer eigens dafür vorgesehenen Textur zu halten. Dazu wird das Prinzip der Adressierung aus Huwe[8] aufgegriffen: Die Adressierung des Materials wird mit Hilfe des Farbfeldes realisiert. Ein Kanal des Farbfeldes hält die Adresse und ein weiterer Kanal den Typ des Materials in der Materialtextur.
Die Trennung der Daten zwischen der geometrischen Beschreibung und dem Material macht auch insofern Sinn, da die Materialdaten in der Regel für Schnittpunkt-Tests nicht benötigt werden.
In den folgenden Abschnitten soll nun das Konzept zur Materialorganisation vorgestellt werden.

3.2.1 Das Grundprinzip

Am Ende des Abschnitts 2.2.3.6 wurde zusammengefasst dargestellt, welche Parameter die vorgestellten Reflexionsmodelle benötigen. Diese Zusammenstellung soll als Grundlage diesen, um das Prinzip der Materialorganisation in diesem Abschnitt zu erklären.

3.2.1.1 Feldgrößen der Materialtypen

In Abbildung 3.3 ist dargestellt, wie viele Einträge pro Feld die einzelnen Materialien benötigen werden. Die Darstellung zeigt, dass diese Anzahl von Feld zu Feld variiert. Daher ist eine Typisierung der Materialien notwendig, damit die einzelnen Feldeinträge in der Material-Textur unterscheidbar sind. Wie bereits erwähnt, wurden die BSDFs spekulare Reflexion und spekulare Transmission zum Material *Glas* zusammengefasst, weshalb der Brechungsindex und der Dispersionskoeffizient nur einmal gespeichert werden müssen.

Lambertian	k_R			
Glas	k_R	k_T	η	CauchyB
Microfacetten	k_R	e	η	

Abbildung 3.3: Benötigte Felder der drei Materialien: Da spekulare Reflexion und Transmission zum Material *Glas* zusammengefasst wurde, muss Brechungsindex η und der Dispersionkoeffizient *CauchyB* nur einmal angegeben werden.

In einem Eintrag steht entweder ein Spektrum oder ein einzelner Wert. Daher werden diese Einträge durch Datenbehälter repräsentiert, die die Aufgabe des Speicherns der Daten übernehmen.

3.2.1.2 Adressierung von Einträgen

Wie in Abschnitt 2.3.1 bereits erläutert, kann der Speicher auf der GPU unterschiedlich formatiert werden. Da die kleinste Größe eines Materials gerade einmal aus einem Eintrag besteht, bietet sich hier eine Formatierung mit einem Kanal pro Pixel eher an, als eine, die ein Pixel in vier Kanäle aufteilt. Aufgrund dieser Festlegung müssen die Einträge der Felder auch einzeln ausgelesen werden, da pixelweise auf die Textur zugegriffen wird.
Wie häufig dann ein Texturzugriff vonstatten gehen muss bis das Material komplett ausgelesen ist, hängt von der Größe des Materials ab. Dazu wird die Definition der Feldgrößen (siehe Abbildung 3.3) genutzt.
Mit der Beispielszene aus Abbildung 2.14 soll nun die Adressierung veranschaulicht werden. Die genutzten Materialien werden zunächst hintereinander in einen Puffer gelegt. Somit hat jeder Eintrag, also jeder Datenbehälter, einen eindeutig zugeordneten Index, wobei der Index des ersten Datenbehälters der Adresse des Materials entspricht. Dieses Prinzip wird in Abbildung 3.4 dargestellt. Die verwendeten Materialien sind als Blöcke mit der jeweiligen Anzahl an Datenbehältern repräsentiert. Für eine bessere Unterscheidung wurden die zu einem Material gehörigen Felder entsprechend der farblichen Erscheinung in der Beispielszene eingefärbt.
Der Index der einzelnen Datenbehälter ist materialübergreifend fortlaufend. Dies ist zwar ein Nachteil, da Indizes zwischen den einzelnen Elementen der Materialien „verschwendet“ werden, jedoch erleichtert dies die Adressierung.
Wie bereits erwähnt, wird das Farbfeld der Szenen-Textur zweckentfremdet, da die hier abgelegten Werte nicht mehr länger als Farben, sondern als Zeiger auf einen anderen Speicherbereich beziehungsweise Typinformationen interpretiert werden.
Die Daten aus dem Puffer mit allen Materialien werden auf der GPU in einer Textur abgelegt. Auch wenn die Adressierung nun mit zweidimensionalen Indizes stattfindet, können dennoch eindimensionale Indizierungen genutzt werden. Durch die Angabe von Höhe und Breite der Textur lässt sich eine Umrechnung vom 1D-Index in einen 2D-Index und umgekehrt realisieren.

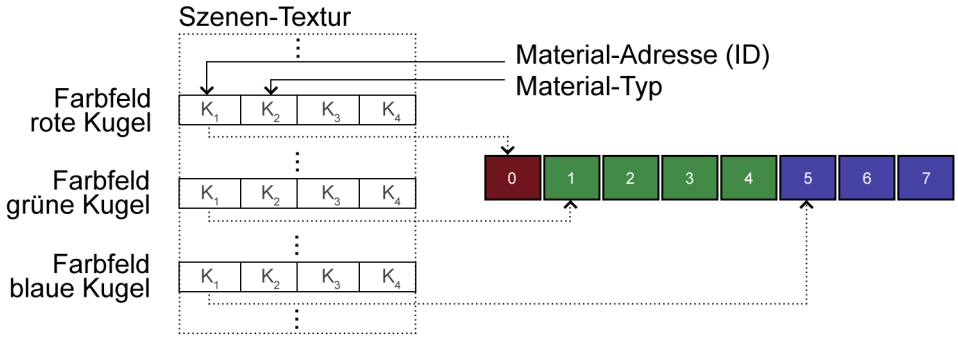


Abbildung 3.4: Prinzip der Adressierung: In der Szenen-Textur ist für jedes Objekt ein Farbfeld mit vier Kanälen zum Speichern von Oberflächenfarben angegeben. Der Kanal K_1 dieses Feldes wird zur Adressierung, K_2 zur Typisierung des Materials genutzt.

Beim Abbilden des Puffers auf eine Textur muss zunächst deren benötigte Größe ermittelt werden. In diesem Fall wird aus der Anzahl der Elemente im Puffer die Quadratwurzel berechnet und der erhaltene Wert aufgerundet. Dieser Wert entspricht der Anzahl an Zeilen und Spalten der Textur, welche somit quadratisch ist. Diese hat mindestens so viel Platz, dass jeder Eintrag im Puffer in die Textur passt. Durch das Aufrunden kann es dazu kommen, dass die Textur auch größer als benötigt sein kann. Der hier verschenkte Speicherplatz ist nicht relevant, da es nur ein kleiner Bereich ist. Abbildung 3.5 veranschaulicht das Abbilden eines 1D-Puffers auf eine 2D-Textur.

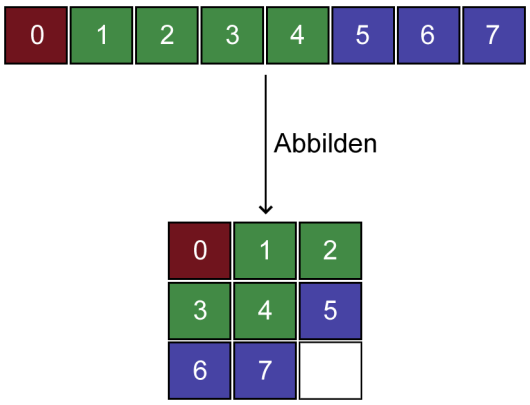


Abbildung 3.5: Abbilden eines 1D-Puffers auf eine 2D-Textur.

3.2.1.3 Blockzerlegung

Bisher wurden die einzelnen Datenbehälter relativ abstrakt betrachtet, da diese sowohl einzelne Werte als auch Spektren aufnehmen konnten. Spektren können jedoch nicht durch einen einzelnen Wert repräsentiert werden und passen somit nicht einen Pixel,

sondern müssen über mehrere Pixel verteilt werden. Zwar ist mit Beginn des Renderprozesses die Anzahl der Intensitäten für alle Spektren festgelegt und somit gleich, jedoch kann diese bei verschiedenen Aufrufen variieren. Dies hängt davon ab, mit welcher Schrittweite über das komplette sichtbare Spektrum integriert wird.
Damit ein Material in der Breite (Anzahl der Pixel hintereinander eines Materialtyps) nicht variiert, werden die Spektren in Materialienblöcke zerlegt. Abbildung 3.6 erklärt das Prinzip. Ein Materialblock wird in mehrere Blöcke zerlegt, so dass jeder dieser Blöcke für eine konkrete Wellenlänge Daten enthält.

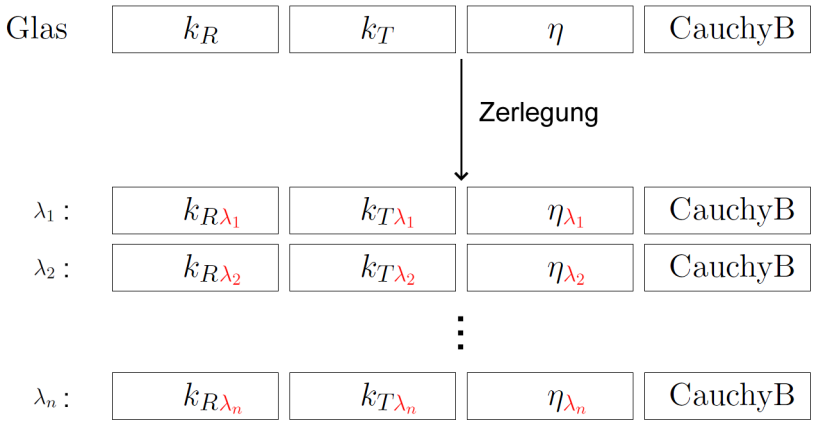


Abbildung 3.6: Darstellung der Blockzerlegung am Beispiel des Glas-Materials: Spektren werden nicht in einem Datenbehälter gehalten, sondern werden in ihre einzelnen Bestandteile zerlegt. Jeder Feldeintrag hält dann immer nur einen Wert des Spektrums. Alle hintereinander aufgeführten Materialien werden in mehrere Blöcke zerlegt, die jeweils für eine bestimmte Wellenlänge λ gelten.

Durch dieses Prinzip ist die Breite des Materials nicht mehr abhängig von der Anzahl an Intensitäten des Spektrums. Ein Nachteil ist, dass konstante, also wellenlängenunabhängige Werte, repliziert werden. Sie stehen in jedem Block separat und benötigen somit ebenso Speicher. Dieser Sachverhalt wird jedoch nicht weiter betrachtet, könnte aber beim Optimieren des Prinzips berücksichtigt werden.
Die einzelnen Blöcke aller Materialien werden nun sortiert und zu einem *Lambda-Block* zusammengefasst. Jeder dieser einzelnen Lambda-Blöcke kann nun nach dem Prinzip, wie in Abbildung 3.5 gezeigt, in eine Textur auf der GPU abgelegt werden. Um zu vermeiden, dass nach dem Supersampling für eine Wellenlänge jedesmal ein Transfer des Lambda-Blocks für die nächste Wellenlänge stattfinden muss, werden mehrere dieser Blöcke, möglichst alle, zu Beginn des gesamten Renderprozesses auf die Karte transferiert.
Ist die Größe dieses Lambda-Blocks bekannt, ist die Berechnung der genauen Position beziehungsweise die Auswahl eines Parameters für eine gegebene Wellenlänge nicht wesentlich komplizierter. Die Berechnung der Adresse innerhalb eines Blocks bleibt stets

gleich. Lediglich ein *Offset* ist bei der Adresse zu berücksichtigen. Die Anzahl an Lambda-Blöcken entspricht der Anzahl an Schritten beim Lambda-Sampling. Abbildung 3.7 verdeutlicht die Anordnung der einzelnen Blöcke in der Materialtextur.

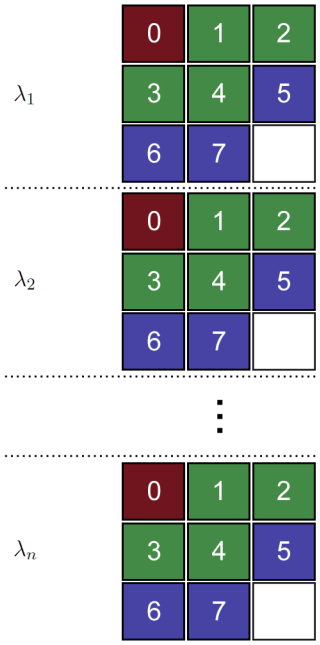


Abbildung 3.7: Darstellung der Anordnung der Lambda-Blöcke in der Materialtextur. Die lokale Adressierung, also innerhalb eines Lambda-Blocks, ist für alle anderen Blöcke die gleiche. Es muss nur die Position des entsprechenden Lambda-Blocks bei der Adressberechnung berücksichtigt werden.

Mit diesem Konzept ist es nun möglich Materialien zu nutzen, die spektrale Verteilungen als Parameter benötigen. Dabei spielt es keine Rolle, ob es sich um ein Spektrum zur Repräsentation einer Farbe oder beispielsweise eines Brechungsindizes handelt.

3.2.2 Behandlung von RGB-Farben

In der Regel liegen Farbinformationen nicht in Spektren, sondern häufig als RGB-codierte Farbwerte vor. Um Szenen dennoch spektral rendern zu können, müssen diese zunächst, wie in Abschnitt 2.1.4 beschrieben, umgewandelt werden. Die Frage ist, wo und wann diese Umwandlung stattfinden sollte.
Werden RGB-Farbwerte vor dem eigentlichen Renderprozess in Spektren umgewandelt, so muss zum einen mehr Speicher bereitgestellt und die Daten auf die GPU transferiert werden. Andererseits bedeutet ein Umrechnen während des Rendernvorgangs zusätzlichen Aufwand. Daher ist im Vorfeld abzuwägen, wann die Umrechnung durchgeführt werden muss.

Generell gilt der Ansatz, Texturzugriffe auf der GPU zu vermeiden, da diese Zeit kosten. Da für die Umrechnung nach Smits[26] Basisspektren benötigt werden und diese in einer Textur hinterlegt werden müssen, käme man um das Lesen der Textur nicht herum. Daher sollten so viel RGB-Farbwerte wie möglich auf Seiten der CPU in Spektren umgewandelt und in der Materialtextur abgelegt werden.
Somit landen bereits vordefinierte sowie aus RGB-Farbwerten gewonnene Spektren in der Materialtextur. Die Quelle, aus der die Spektren stammen, ist somit aus Sicht des Renderers transparent.

3.2.3 Behandlung von Bild-Texturen



Abbildung 3.8: Beispielszene mit verschiedenen Materialien: Zur Definition der Rauheit der Oberfläche wurde eine Bild-Textur verwendet. In ihr ist auf einem grauen Hintergrund mit schwarzen Buchstaben das Wort „Sunshine“ geschrieben. Entsprechend der Bild-Textur erscheinen die Stellen auf der Oberfläche als spiegelnd glatt, an denen die schwarzen Buchstaben stehen. Der graue Hintergrund sorgt dagegen für eine raue Oberfläche.

Eine weitere Möglichkeit Oberflächen von Objekten zu beschreiben sind Texturen. Auch sie müssen bei der Konzipierung der Materialorganisation mit in Betracht gezogen werden. Das Problem hierbei ist, dass eine Textur in der Regel nicht aus einem einzelnen sondern aus vielen verschiedenen Farbwerten besteht. Die hier vorgestellte Anordnung der Parameter im Speicher ist jedoch für einzelne Farb- oder Parameterwerte konzipiert. Daher werden Texturen nicht mit in der Material-Textur, sondern wiederum separat in einer eigenen Textur abgelegt. Dennoch wird in der Material-Textur für den entsprechenden Parameter ein Adresseintrag vorgenommen, damit die jeweilige Bild-Textur dem Objekt zugeordnet werden kann. Je nachdem für welchen Parameter eine Bild-Textur angegeben wurde, wird ein Eintrag mit der Adresse der Bild-Textur in das entsprechende Feld des Materials hinterlegt und zwar für alle Wellenlängen.
Dies bedeutet zwar wiederum Speicherplatzverschwendung, da die Adresse der Textur

redundant gespeichert wird, jedoch erleichtert diese Art und Weise die Handhabung. Auf Grund des begrenzten Speicherplatzes, werden RGB-Bild-Texturen im Vorfeld nicht in Spektren umgewandelt, sondern im gleichen Zustand auf die GPU transferiert. Beim Lesen der Pixel aus der Bild-Textur müssen diese deshalb auf der GPU während des Rendervorgangs umgewandelt werden. Somit ist ebenso der Mechanismus zur Konvertierung von RGB-Farben in Spektren auf der GPU umzusetzen.
Jeder Parameter kann durch eine Bild-Textur definiert werden, denn es spielt keine Rolle, um was für einen Parameter eines Materials es sich handelt. Dies erlaubt zum Beispiel für das Microfacetten-Material die Oberflächenrauheit durch eine Bild-Textur festzulegen.
Die Abbildung 3.8 demonstriert diese Möglichkeit: Auf dem hinteren Block ist der Schriftzug „Sunshine“ zu lesen. Hierbei handelt es sich um das Microfacetten-Material, das zur Definition der Rauheit eine Textur mit einem Schwarz-Weiß-Bild erhalten hat. Die Buchstaben sind in diesem Bild schwarz und der Hintergrund weiß. Während der Evaluierung des Materials werden nun Helligkeitswerte des Bildes als Rauheits-Parameter interpretiert.
Das Bild wird auf das Objekt projiziert. Dadurch können die zweidimensionalen UV-Koordinaten ermittelt und anschließend die entsprechenden Pixel aus der Bild-Textur gelesen werden. Der Helligkeitswert dieses Pixels wird nun interpretiert: Überall da, wo das Bild schwarz ist, ist die Oberfläche glatt und spiegelt somit ideal spekular. Die Farbe Weiß hingegen lässt die Oberfläche an der Stelle rau erscheinen, wodurch die Umgebung „verschmiert“ dargestellt wird.

4 Implementierung

In den vorangegangenen Kapiteln wurden Grundlagen gegeben und Konzepte vorgestellt, welche nun in die Umsetzung eines spektralen Raytracers einfließen. Analog zu Kapitel 3 wird dabei zunächst die Ablaufsteuerung und anschließend die Organisation der Materialdaten betrachtet. Schlussendlich ist ein nicht unerheblicher Anteil dieses Kapitels die Implementierung der Shader-Programme, die letztendlich auf der GPU ausgeführt werden.
Die Implementierung des spektralen Raytracers wird in das bereits bestehende Paket `de.grogra.ext.sunshine` der GroIMP-Erweiterung „Sunshine“ integriert. Bis auf die Klasse zur Verwaltung und Ablaufsteuerung werden alle Komponenten in Unterpaketen von `de.grogra.ext.sunshine.spectral` abgelegt. Dieses Paket dient als Ausgangspunkt und wird zur Vereinfachung nur noch als `.spectral`-Paket bezeichnet.

4.1 Ablaufsteuerung

Die Umsetzung des Path Tracers nach Huwe[8] kapselte alle zur Steuerung nötigen Methoden in eine dafür vorgesehene Klasse. Im Laufe der Modifizierung wurde diese Struktur aufgespalten. Es entstand die abstrakte Basisklasse `SunshineRaytracer`, von der die konkreten Vertreter abgeleitet werden. Neben der Klasse `SunshineStandardRaytracer` für den RGB basierten Path Tracer kommt nun eine weitere Klasse für das spektrale Raytracing hinzu:
Diese Klasse soll in diesem Abschnitt nun näher betrachtet werden, um die Umsetzungen der Konzepte zu verdeutlichen. Den Anfang macht die Ablaufsteuerung mit der Umsetzung der Ereignisschleife. Hierzu soll nun eine kurze Erläuterung der Implementierung erfolgen.
Die Basisklasse `SunshineRaytracer` hält in `GLAutoDrawable drawable` alle auf den OpenGL-Kontext bezogenen Informationen, welche zum Absetzen von OpenGL-Befehlen nötig sind. Des Weiteren ist in `SunshineRaytracer` eine Klasse namens `SunshineAction` gekapselt. Diese erbt wiederum von `java.lang.Runnable` und implementiert die Methode `run()`. Die Hauptaufgabe dieser Methode ist der Aufruf von `drawable.display()`. Einen Auszug der Implementierung ist in Code-Auflistung 4.1 zu sehen.
Von dieser gekapselten Klasse wird nun in `SunshineRaytracer` ein Objekt `SunshineAction sunshineAction` angelegt. Beim Starten des Renderers wird dieses Objekt auf die Ereigniswarteschlange von `java.awt.EventQueue` gelegt. Der AWT-Thread ist für das Ausführen der OpenGL-Befehle verantwortlich und arbeitet die Warteschlange ab. Ereignisse werden von der Warteschlange genommen und beginnen ihre Arbeit mit dem Aufruf von `drawable.display()`. Bevor das Ereignis beendet wird, kann es sich selbst

```
class SunshineRaytracer implements GLEventListener
{
  ...
  protected class SunshineAction implements Runnable
  {
    public void run()
    {
      try
      {
        // Ausfuehrung eines einzelnen Schritts
        drawable.display();
      }
      catch (Exception e)
      {
        synchronized (sunshineAction)
        {
          sunshineAction.notify();
        }
      }
    }
  }

  // Erzeugen eines sunhineAction-Objekts
  protected SunshineAction sunshineAction = new SunshineAction();
  ...
```

Abbildung 4.1: Auszug aus Implementierung der Klasse `SunshineRaytracer` zur Umsetzung der Ereignisschleife.

wieder auf die Warteschlange legen und wird dann durch den AWT-Thread wiederum gestartet. Somit entsteht eine Schleife, in der die Methode `display()` immer wieder aufgerufen wird. Der Beispielcode in Code-Auflistung 4.2 zeigt einen Auszug aus der Implementierung der Klasse `SunshineRaytracer`.
Die Klasse implementiert die Schnittstelle `GLEventListener` aus dem Paket `javax.media.opengl` teilweise. Die Methode `display` delegiert den Aufruf vom AWT-Thread zur Methode `processNextKernel()`. Diese ist abstrakt und muss somit von allen abgeleiteten Klassen implementiert werden (siehe Code-Auflistung 4.2).
Somit wird die Methode `processNextKernel()` immer wieder aufgerufen. Sie dient nun als Einstiegspunkt in die Ablaufsteuerung. Der Methoden-Kern besteht im Wesentlichen aus einem `switch`-Konstrukt und behandelt die auftretenden Fälle. Dazu wird ein globaler Zustand in `int state` gehalten und in `processNextKernel()` überprüft. Darauf hin wird eine entsprechende *process*-Methode ausgeführt. Zur Identifizierung der Status müssen diese definiert werden. Sie haben die Form `STATE_*`, wobei * für den Namen des jeweiligen Status steht. Abbildung 4.3 zeigt die Umsetzung.
Die Struktur der Umsetzung entspricht im Wesentlichen der in Abbildung 3.2 darge-

```
  protected void startRenderer()
  {
    // Legt sunshineAction auf Ereigniswarteschlange vom AWT-Thread.
    EventQueue.invokeLater(sunshineAction);

    // Ab hier auf das Ende des Renderns warten.
    synchronized(sunshineAction)
    {
      try
      {
        sunshineAction.wait();
      }
      catch (InterruptedException e) { ... }
    }
  }

  // Ereignisse werden von der AWT-Warteschlange geholt und
  // durch Aufruf dieser Methode ausgefuehrt.
  protected void display(GLAutodrawable drawable)
  {
    processNextKernel();
  }

  // Alle von dieser Klasse abgeleiteten Klassen muessen diese
  // Methode implementieren, da sie in display() aufgerufen wird.
  abstract void processNextKernel();

  ...
}
```

Abbildung 4.2: Fortsetzung Auszug aus Implementierung der Klasse `SunshineRaytracer` zur Umsetzung der Ereignisschleife.

stellten Pipeline. In den einzelnen *process*-Methoden werden die entsprechenden Kernel ausgeführt und benötigte Parameter beziehungsweise Daten an die GPU übergeben. Durch das Ausführen der einzelnen Methoden kann sich dann der Status ändern, so dass beim nächsten Schritt eventuell andere Methoden und somit andere Kernels zum Zuge kommen.
Da sich die Neuerungen im Wesentlichen nur auf `STATE_SHADE` und `STATE_CON` beziehen, wird an dieser Stelle nicht auf die implementierungstechnischen Details zu den anderen Methoden eingegangen.
Der Initialzustand des Renderers ist `STATE_STARTUP`. So wird im ersten Schritt, dass heißt beim ersten Aufruf von `processNextKernel()`, die Methode `processStartup()` einmal zu Beginn ausgeführt. In ihr werden die Texturen an den FBO gebunden und aktiviert. Ist dies geschehen, ändert sich der Status von `STATE_STARTUP` auf `STATE_INIT`. Mit diesem Status wird im nächsten Schritt `processInit()` aufgerufen. Hier kommt es

```
class SunshineSpectralPT extends SunshineRaytracer
{
  ...
  // Definitionen der einzelnen Status "STATE_*"

  // Speichert den aktuellen Zustand des Raytracers
  protected int state = STATE_STARTUP;

  public void processNextKernel()
  {
    switch(state)
    {
      case STATE_STARTUP:       { processStartup();    break;  }

      case STATE_INIT:          { processInit();       break;  }

      case STATE_GENRAY:        { processGenRay();     break;  }

      case STATE_INTERSECT:     { processIntersect();  break;  }

      case STATE_SHADOW_TEST:   { processIntersect();  break;  }

      case STATE_SHADE:         { processShade();      break;  }

      case STATE_CON:           { processCon();        break;  }

      case STATE_FINAL:         { processFinal();      return; }

      default: // Fehler: Nicht definierter Status!
    }

    // Vor dem Beenden wieder auf die Warteschlange legen.
    EventQueue.invokeLater(sunshineAction);
  }
  ...
}
```

Abbildung 4.3: Auszug aus Implementierung der Methode `processNextKernel()` aus der Klasse `SunshineSpectralPT`. Dies entspricht der Umsetzung des Konzepts aus Abbildung 3.2.

zu einem ersten Kernel-Aufruf. Dadurch wird,wie bereits in Abschnitt 2.3.1 beschrieben, die Textur „vorbereitet“. Nach der Kernel-Ausführung ändert sich der Status auf `STATE_GENRAY`, so dass als nächstes die Primärstrahl-Erzeugung stattfinden kann.
In der Methode `processGenRay()` wird der `GenRayKernel` aufgerufen. Die Position der Kachel bestimmt den Bildausschnitt, für den die Primärstrahlen pro Bildpunkt zu erzeugen sind. Anschließend kann der Status auf `STATE_INTERSECT` gesetzt werden.

Im nächsten Schritt werden nun mit `processIntersect()` die Schnittpunkttests durchgeführt. Dabei wird der `IntersectionKernel` gestartet. Durch die angegebene maximale Anzahl an Objekten, mit die pro Kernel-Aufruf getestet werden dürfen, muss hier kontrolliert werden, wann alle Objekte der Szene getestet wurden. Dies wird über Zähler bewerkstelligt. Wurden alle Objekte betrachtet, kann der Status auf `STATE_SHADOW_TEST` geändert, andernfalls muss der Kernel im nächsten Schritt nochmals ausgeführt werden. In diesem Fall ändert sich der Status nicht.
Liegt der Status `STATE_SHADOW_TEST` vor, wird abermals die Methode `processIntersect()` ausgeführt. Da jedoch ein anderer Status anliegt, also ein Schattentest bezüglich eines Schnittpunktes und einer Lichtquelle durchzuführen ist, muss dieser Zustand dem Kernel übermittelt werden. Dies geschieht mittels `uniform`-Parameter. Nach Beendigung des Schattentests kann nun das Shading stattfinden. Dazu wird der Status auf `STATE_SHADE` gesetzt. Abbildung 4.4 stellt die Implementierung dieser Methode dar.

```
protected void processShade()
{
  shadingKernel.setUniform("recursionDepth",   currentRecursionDepth);
  shadingKernel.setUniform("superSample",      currentSuperSample    );
  shadingKernel.setUniform("lambda",           lambda                );

  shadingKernel.setUniform("colorRGB2SPDRow", colorRGB2SPDRow       );
  shadingKernel.setUniform("matRowOffset",     matRowOffset          );

  shadingKernel.setUniform("CIE_X", SPDConversion.getXMatch(lambda));
  shadingKernel.setUniform("CIE_Y", SPDConversion.getYMatch(lambda));
  shadingKernel.setUniform("CIE_Z", SPDConversion.getZMatch(lambda));

  execute(shadeKernel);

  nextStep();
}
```

Abbildung 4.4: Implementierung der Methode `processShade()`. Wesentlicher Bestandteil dieser Methode ist es, die zur Ausführung des `ShadingKernel`s nötigen Parameter zu übergeben und anschließend die Kernel selbst zu starten.

Der wesentliche Teil dieser Methode ist es, die zur Ausführung benötigten Parameter zu übergeben. Jede Implementierung der `Kernel`-Klasse bietet dazu die Methode `setUniform(String, Object)` an. Durch den ersten Parameter dieser Methode wird die betroffene `uniform`-Variable im Fragment-Programm adressiert. Der zweite Parameter enthält den Wert, der übergeben werden soll und darf nur vom Typ `int` oder `float` sein. Jedoch können auch Felder, also `int[]` und `float[]` übergeben werden, was jedoch hier keine Verwendung findet.
Da nach dem Ausführen eines Kernels, bis auf die im FBO gespeicherten Werte, keine

weiteren Informationen gespeichert werden, müssen Zustandsinformationen von außen zugeführt werden. Neben dem Zustand des Supersamplings und der Rekursionstiefe ist vor allem der Zustand von `lambda`, also die aktuell zu betrachtende Wellenlänge, als Parameter zu übergeben. Eine Änderung dieser Zustände erfolgt nur auf Seite der CPU, also in der Verwaltung des Renderer und muss daher bei jeder Kernel-Ausführung mitgeteilt werden.
Bei den nächsten beiden `uniform`-Parametern handelt es sich um einen Adress-*Offset*, also um einen Adressabstand zur ersten Adresse eines Adressraums. Dieser Offset dient dazu, das Adressieren von Texturen zu vereinfachen. Der erste Offset `colorRGB2SPDRow` dient zur Adressierung der Textur, die die sieben Basis-Spektren zur Konvertierung von RGB-Farbwerten in einzelne Intensitäten ermöglicht. Da diese Textur auch nach den einzelnen Wellenlängen sortiert, wird auch immer nur wellenlängenabhängig in einem bestimmten Bereich dieser Textur gelesen. Gleiches gilt auch für den Offset `matRowOffset`. Durch die Blockzerlegung in einzelne λ-Blöcke ist die lokale Adressierung, also innerhalb dieser Blöcke, immer gleich. Sofern die Startadresse des aktuellen λ-Blocks bekannt ist, ist die Adressberechnung einfach, da der Offset lediglich zur lokalen Adresse aufzuaddieren ist.
Genauso wichtig, wie die Übergabe der aktuellen Wellenlänge, ist die Übergabe der drei Farbanpassungsfunktionen, genauer gesagt, die wellenlängenabhängigen Werte. Alle drei Farbanpassungsfunktionen werden in SPDs in Form von spektralen Kurven gehalten. Es gibt im Wesentlichen zwei Arten von spektralen Kurven: reguläre und irreguläre Kurven. Bei den regulären Kurven wird zunächst die untere und obere Grenze angegeben. Des Weiteren wird festgelegt, wie groß die Schrittweite, also die Differenz der Wellenlängen zwischen zwei aufeinander folgenden Einträgen, ist. Anschließend kann ein Feld bestehend aus einzelnen Werten – den Intensitäten – der regulären Kurve übergeben werden.
Da die Grenzen und Schrittweite der Kurve nun bekannt sind, kann innerhalb dieser Grenzen für jedes λ ein Wert entnommen werden. Nicht zu jeder Wellenlänge müssen Werte vorhanden sein. Ist ein Wert für eine Wellenlänge nicht direkt vorhanden, sprich der gewünschte Wert liegt zwischen zwei Einträgen, kann er mittels linearer Interpolation von den beiden angrenzenden Werten gewonnen werden.
Bei irregulären Kurven muss zusätzlich ein Feld übergeben werden, welches Informationen darüber enthält, zu welchen Wellenlängen die Intensitäten gehören. Da die Abstände der Wellenlängen unter Umständen nicht konstant sind, ist diese zusätzliche Information notwendig, um die Zugehörigkeit genau ermitteln zu können. Auch hier können nicht vorhandene Werte mittels linearer Interpolation gewonnen werden.
Im Fall der drei Farbanpassungsfunktionen liegen reguläre Kurven mit einer Wellenlängen-Schrittweite von $1nm$ vor. Hier muss also beim Auslesen keine lineare Interpolation erfolgen. In der Klasse `SPDConversion` sind dazu statische Methoden definiert, die für eine gegebene Wellenlänge Werte zurückgeben. Die Methoden `getXMatch(lambda)`, `getYMatch(lambda)` und `getZMatch(lambda)` liefern die Werte, die für die Umrechnung der Intensitäten in eine XYZ-Farbe benötigt und als Parameter dem `ShadingKernel` übergeben werden.
Nach der Übergabe dieser Parameter kann der Kernel gestartet werden. Anschließend

erfolgt nun die Überprüfung des Status des Renderes in der Methode `checkForNextState()` (siehe Abbildung 4.5).

```
protected void checkForNextState()
{
  currentRecursionDepth++;

  if(currentRecursionDepth >= recursionDepth)
  {
    currentSample++;

    if(currentSample < superSample)
    {
      state = STATE_GENRAY;

    } else {

      setNextLambda();

      if(lambda >= lambdaMax )
        state = STATE_CON;
      else
        state = STATE_GENRAY;

      currentSample = 0;
      currentRecursionDepth = 0;
    }

  } else {

    state = STATE_INTERSECT;
  }
}
```

Abbildung 4.5: Implementierung der Methode checkForNextState(). Je nach Zustand der Rekursion, des Supersamplings oder der Lambda-Schleife wird der nächste Zustand gesetzt.

Hier wird nach der Ausführung des `ShadingKernel` überprüft, was als nächstes geschehen soll. Dies hängt zunächst von den Zählständen der Rekursions-, Supersampling- und Lambda-Schleifen ab. Als erstes wird der Rekursionszähler um eins erhöht. Ist die maximale Rekursionstiefe noch nicht erreicht, wird der Status auf `STATE_INTERSECT` gesetzt und es gibt nichts weiter zu tun. Dies würde zur Ausführung der Methode `processIntersect()` führen.
Ist jedoch die maximale Rekursionstiefe erreicht, wird der Supersampling-Zähler inkrementiert und anschließend überprüft. Sind nicht genügend Pfade erzeugt (`currentSample < superSample`), wird der Status auf `STATE_GENRAY` gesetzt. Somit

wird ein neuer Strahl in der Kamera erzeugt.
Sind jedoch genügend Pfade pro Bildpunkt erzeugt, kann die nächste Wellenlänge betrachtet werden. Dazu wird mit `setNextLambda()` der Zähler um eins erhöht. Anschließend erfolgt eine Überprüfung, ob sich `lambda` innerhalb der angegebenen Grenze befindet. Ist dies nicht der Fall müssen neue Pfade verfolgt, also neue Strahlen in der Kamera erzeugt werden (`state = STATE_GENRAY`).
Wurden alle Wellenlängen betrachtet, ist die aktuelle Kachel fertig und kann vom XYZ- in den RGB-Farbraum konvertiert werden (`state = STATE_CON`). Unabhängig davon, ob alle Wellenlängen betrachtet sind, sobald der Lambda-Zähler hochzählt, müssen der Rekursions- und Supersampling-Zähler zurückgesetzt werden.
Wird der Status auf `STATE_CON` gesetzt, kommt es zur Ausführung des `ConversionKernels`. Hierzu sind keine Parameter an die GPU zu übergeben. Alle nötigen Parameter zur Konvertierung von XYZ nach RGB sind in den Quellcode des Kernels statisch integriert. Die einzige Besonderheit dieser Methode ist es, alle drei Schleifenzähler zurückzusetzen. Des Weiteren muss nun überprüft werden, ob alle Kacheln berechnet wurden. Ist dies nicht der Fall, wird die Kachel auf die nächste Position verschoben, damit das nächste Teilbild berechnet werden kann.
Ist das Gesamtbild fertig, sprich keine weiteren Kacheln sind zu betrachten, ändert sich der Status auf `STATE_FINAL`. In der dazugehörigen Methode werden alle Texturen auf der GPU gelöscht. Danach wird kein weiteres Ereignis in die Warteschlange des AWT-Threads gelegt: Der Renderer ist fertig.

4.2 Darstellung der Materialien in der GroIMP 3D-Ansicht

Zum Modellieren von 3D-Szenen ist ein wichtiger Punkt die Zuordnung der Materialien zu den Objekten. Durch die Integration der Implementierung des Renderers in das bestehende Programm GroIMP müssen diese Materialien nicht nur Render-intern sondern ebenso für die Anwendung selbst sichtbar sein. Das heißt, dass der Benutzer einzelne Materialien in GroIMP auswählen und Objekten zuordnen kann.
Dazu wurden einige dieser Materialien als *GroIMP-Shader* umgesetzt. Des Weiteren wurden Schnittstellen definiert, die eine eventuelle Erweiterung um andere GroIMP-Shader erleichtert. Bevor die Implementierung der Materialorganisation betrachtet werden kann, soll ein kurzer Überblick über die Architektur dieser Shader gegeben werden. Abbildung 4.6 zeigt ein Klassendiagramm dieser Architektur.
Damit GroIMP Shader in der 3D-Ansicht anzeigen kann, implementieren diese die Schnittstelle `de.grogra.imp3d.shading.Shader`. Die Umsetzungen der Shader für den spektralen Raytracer – des Weiteren Spektral-Shader genannt – benötigen beim Prozess zum Einsammeln der Szenen-Daten eine Sonderbehandlung. Daher müssen diese Shader sich von den Standard-Shadern unterscheiden. Dies geschieht mit der Schnittstelle `SunshineSpectralShader`. Sie ist Bestandteil des Pakets `.spectral.shader`.
Alle spektralen Shader halten die für den Renderer relevanten Daten in Kanälen. Diese Kanäle implementieren die Schnittstelle `SunshineChannel`.

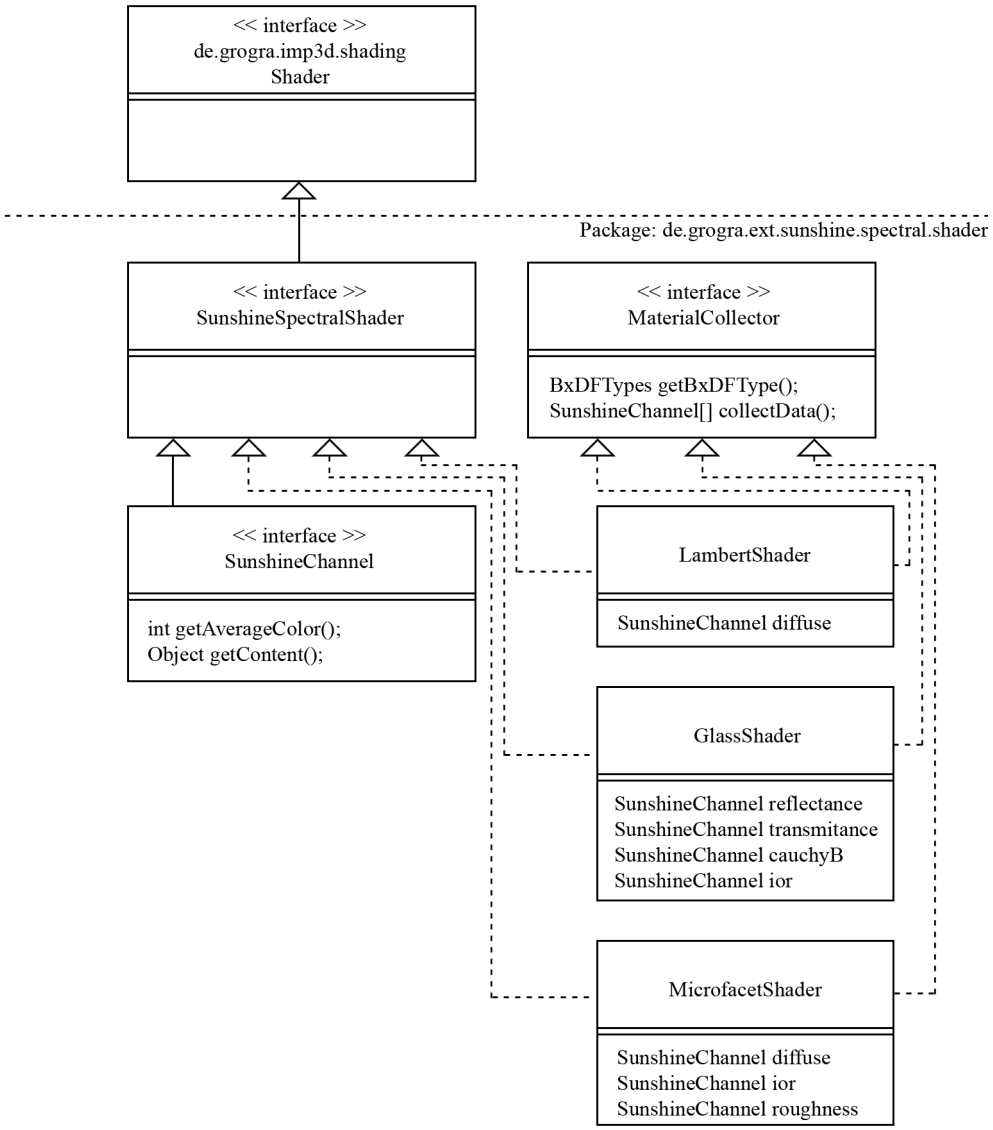


Abbildung 4.6: Architektur der Shader für den spektralen Renderer in GroIMP.

Der Inhalt eines Kanals kann dabei

- ein einzelner Wert,
- eine spektrale Kurve
- oder eine Textur sein.

Für jeden dieser drei Fälle gibt es eine dazugehörige Implementierung der Schnittstelle `SunshineChannel`, die jedoch nicht in Abbildung 4.6 dargestellt ist.

Eine weitere Schnittstelle des selben Pakets heißt `MaterialCollector`. Alle spektralen Shader müssen diese Schnittstelle implementieren, die alle Kanäle in einem Feld aus mehreren Implementierungen der Schnittstelle `SunshineChannel` zurück gibt.

4.3 Organisation von Materialdaten

Wie in Abschnitt 3.2.1 wurde das Prinzip der Blockzerlegung zur Organisation der Materialien erläutert. Dies ermöglicht ein unkompliziertes Zugreifen auf Materialdaten während der Kernel-Ausführung. Die Herausforderung der Material-Organisation ist die Anordnung der Daten gemäß des Konzepts der Blockzerlegung im Textur-Speicher.
Diese Aufgabe übernimmt die Klasse `MaterialHandler` im `.spectral`-Paket: Es müssen zunächst alle `SunshineSpectralShader` eingesammelt, sortiert und die Werte entsprechend der Wellenlängenabhängigkeit angeordnet werden.
Das Einsammeln der Daten einer Szene übernimmt die Klasse `SunshineSceneVisitor` (SSV) aus dem Paket `de.grogra.ext.sunshine` und wurde bereits in der Arbeit von Huwe[8] implementiert. Lediglich für die Behandlung der spektralen GroIMP-Shader muss eine kleine Anpassung vorgenommen werden.
Der SSV durchsucht den Szenegraph von GroIMP nach sichtbaren Objekten. Jedes Mal, wenn ein solches Objekt gefunden wird, kommt es zum Aufruf der Methode `getShader(Shader shader)` des SSV. Als Parameter wird der dem Objekt zugeordnete Shader übergeben. Der Rückgabewert ist eine in `Color4f` hinterlegte Farbe. Dieser Rückgabewert dient jedoch nicht nur als Träger der Farbinformation von Objekten, sondern kann auch als Zeiger auf komplexere Oberflächenbeschreibungen dienen.
In ihr kann nun überprüft werden, ob der Shader die Schnittstelle `SunshineSpectralShader` implementiert. Ist dies der Fall, wird, sofern noch nicht geschehen, eine Instanz der Klasse `MaterialHandler` erzeugt. Auf ihr wird nun die Methode `addShader()` aufgerufen und besagter Shader als Parameter übergeben. Damit ist die Modifikation des SSV komplett.
Die einzelnen Kanäle der spektralen GroIMP-Shader werden zunächst in einzelnen Behältern verpackt. Die Behälterfunktion übernimmt die in `MaterialHandler` gekapselte Klasse `MatElemContainer`. Die Implementierung ist auszugsweise in Abbildung 4.7 dargestellt.
Für jeden Kanal eines Materials ist nun ein `MatElemContainer` erzeugt. Alle zu diesem Material gehörenden Container werden nun in eine Liste gelegt. Anschließend wird dieser Liste eine eindeutige Identität zugeordnet. Das Tupel Identität und Container-Liste kann nun in eine Hash-Tabelle (`materials`) gespeichert werden. Auf Grund der variierenden Anzahl an Elementen eines Materials ist diese Form der Datenhaltung notwendig.
Eine weitere Aufgabe des Containers ist die Behandlung unterschiedlicher Daten. Wie bereits erwähnt, können die Daten einem einzelnem Wert, einer spektralen Kurve oder einer Bild-Textur entsprechen. Daher überprüft die Methode `setData(SunshineChannel channel)` dieser gekapselten Klasse den Inhalt des zu speichernden Kanals und handelt entsprechend des festgestellten Datentyps.
Eine Besonderheit bildet hier der Fall einer Bild-Textur. Hier wird nicht das gesamte Bild

```
public class MaterialHandler
{
  ...

  public class MatElemContainer
  {
    // Typ dieser Container-Instanz.
    public MatElemTypeCon type;

    // Einzelner Wert.
    private float value;

    // Regulaere Spektralkurve.
    private SunshineRegularSpectralCurve curve;

    // Daten und Typ setzen.
    public void setData(SunshineChannel channel)
    {
      // Auslesen des Kanalinhalts eines spektralen Shader.
      Object obj = channel.getContent();

      // Erwartete Datentypen der Inhalte:

      // Spektrale Kurve
      if(obj instanceof SunshineRegularSpectralCurve) { ... } else

      // Bilddaten (Bild-Texturen)
      if(obj instanceof Image) { ... } else

      // Einzelwert.
      if(obj instanceof Float) { ... }

      else { //FEHLER: Dieser Datentyp wird nicht unterstuetzt };
    }
  }

  // Ablage fuer alle eingesammelten Material-Daten.
  private HashMap<Integer, ObjectList<MatElemContainer>> materials;
  ...
}
```

Abbildung 4.7: Auszug aus Implementierung der Klasse `MaterialHandler`: Behälter für die einzelnen Kanäle.

mit aufgenommen, sondern lediglich ein einzelner Wert gesetzt. Dieser Wert entspricht der Identität des Bildes, welches später auf die GPU in eine dafür vorgesehene Textur transferiert wird. Mit Hilfe dieser Identität ist das Bild in der Textur adressiert.

Damit die Identität einer Bild-Textur von einem einfachen Einzelwert auseinander zu halten ist, muss diese „markiert“ werden. Dazu wird das Vorzeichen genutzt. Alle Werte größer gleich 0 sind normale einfache Werte. Sind sie kleiner 0 handelt es sich um eine Bild-Textur-Adresse. Somit muss eine andere Behandlung dieses Datums auf der GPU stattfinden.
Wie bereits oben erwähnt, wird im Falle eines spektralen Shaders im SSV die Methode `addShader()` der Klasse `MaterialHandler` aufgerufen. Hier wird das Verpacken der Informationen in Container durchgeführt. Alle erzeugten Behälter werden anschließend in einer Liste zusammengefasst und in einer Hash-Tabelle gespeichert. In Abbildung 4.8 ist ein Teil dieser dargestellt.

```
public Color4f addShader(SunshineSpectralShader shader)
{
  Color4f color = new Color4f();

  // Erzeugen einer Liste fuer die Container
  ObjectList<MatElemContainer> tmp = ...;

  ...

  // Adressierung des Materials im Farbfeld der Szenen-Textur
  color.x = ...;
  // Angabe des Materialtyps
  color.y = ...;

  // Auslesen aller Kanaele...
  SunshineChannel[] container;

  container = ((MaterialCollector) shader).collectData();

  // ... mit anschliessendem Verpacken in Container.
  for(int i = 0; i < container.length; i++)
  {
    MatElemContainer elem = new MatElemContainer();

    elem.setData(container[i]);

    tmp.add(elem);
  }

  // Farbfeld mit Adresse und Typ des Materials.
  return color;
}
```

Abbildung 4.8: Auszug aus Implementierung der Methode `addShader()`: Alle Daten werden in Container verpackt und in Hash-Tabellen abgelegt.

Ein besonderes Merkmal dieser Methode ist hier der Rückgabewert `Color4f`. Wie bereits erwähnt, wird durch den Aufruf von der Methode `getShader()` des SSV ein `Color4f` Farbwert zurückgegeben und direkt in das Farbfeld der Szenen-Textur geschrieben. Diese Methode führt im Falle eines spektralen Shaders wiederum die Methode `addShader()` des `MaterialHandler`s aus. Zuletzt genannte Methode erzeugt ihrerseits einen `Color4f`-Rückgabewert. Dieser besteht jedoch nicht aus Farbinformationen, sondern aus Adressierungs- und Typisierungsinformationen eines Materials für ein bestimmtes Objekt.
Wie in Abbildung 4.8 zu sehen ist, wird hierzu eine Instanz `color` von `Color4f` angelegt. In ihr wird die Adresse (`color.x`) sowie der Typ (`color.y`) des Materials hinterlegt.
Die Adresse eines Materials lässt sich durch einfaches Mitzählen der bis dahin erzeugten Container errechnen. Grundsätzlich gilt hier, dass die Adresse eines Materials der Adresse seines ersten Elements beziehungsweise Containers entspricht. Somit muss lediglich mitgezählt werden, wie viele Material-Elemente bereits in `materials` eingetragen wurden. Der aktuelle Zählerstand ist dann die Adresse des Materials.
Die Angabe des Typs ermöglicht es später auf der GPU festzustellen, welcher Material-Typ vorliegt. Dies ist wichtig, um vorher entscheiden zu können, wie oft eine Leseoperation ausgeführt werden muss, um ein Material zu komplettieren. Alle Typen von Materialien sind fest in einer bestimmten Reihenfolge definiert. Die Positionsnummer eines Materials dient als Wert für die Angabe des Typs.
Ist der SSV fertig, also alle Objekte der Szene wurden betrachtet, liegen im `MaterialHandler` nun alle Materialien, die einem spektralen Shader entsprechen. Diese Daten liegen jedoch in einer Hash-Tabelle vor und können nicht in der Form auf die GPU transferiert werden. Dazu müssen die Daten in einen `ByteBuffer` verpackt werden. In Abbildung 4.9 ist die Implementierung des Verpackens der Daten in einen `ByteBuffer` dargestellt.
In dieser Methode erfolgt die Realisierung des Konzepts der Blockzerlegung, wie es in Abschnitt 3.2.1 erklärt wurde. Die Methode wird während der Initialisierungsphase des Renderers aufgerufen. Zu diesem Zeitpunkt liegen alle Materialien in der Hash-Tabelle. Der Methode werden die Grenzen des betrachteten Spektrums sowie die Schrittweite, mit der über dieses Spektrum intergriert werden soll als Parameter übergeben. Ebenso wird die Anzahl der Stichproben (`lambdaSamples`) übergeben. Dieser Parameter wird benötigt, um die Anzahl der λ-Blöcke zu bestimmen, denn zu jeder betrachteten Wellenlänge soll ein Block existieren. Außerdem sollen nicht mehr Blöcke als notwendig auf die GPU transferiert werden.
Die Material-Textur besteht aus n λ-Blöcken, die untereinander stehen. Deren Ausdehnung ist quadratisch dimensioniert. Somit ist die gesamte Textur so breit wie ein Block, aber so hoch wie n Blöcke. Aus dieser Überlegung heraus lässt sich nun die gesamte Textur-Größe leicht berechnen und ein entsprechend großer `ByteBuffer` erzeugen.
Dieser `ByteBuffer` wird nun mit Hilfe von drei ineinander verschachtelten Schleifen aufgefüllt. Die äußerste Schleife geht durch das vorgegebene Spektrum in einer vorgegebenen Schrittweite. In ihr werden wieder mit einer Schleife alle Materialien aus der Hash-Tabelle gelesen. Die innerste Schleife geht über alle Elemente des Materials. In dieser Schleife wird nun von jedem Element des Materials ein Wert entnommen. Handelt

```
public int prepareData(int lambdaMin,
      int lambdaMax, int lambdaStep, int lambdaSamples)
{
  // Liste mit alle Elementen eines Materials.
  ObjectList<MatElemContainer> material;

  // Ausdehnung der Material-Textur berechnen.
  int numOfColumns  = Math.ceil(Math.sqrt(materialElements));
  int numOlElements = numOfColumns * numOfColumns;

  // Anlegen des ByteBuffer
  bbMaterials       = BufferUtil.newByteBuffer(
                          numOlElements * BufferUtil.SIZEOF_FLOAT *
                          lambdaSamples);

  // Fuer die Wellenlaengen des angegebenen Bereichs ...
  for(int l = lambdaMin; l < lambdaMax; l+= lambdaStep)
  {
    // .. sowie fuer jedes Material:
    for(int i = 0; i < materialsCount; i++)
    {
      // Lese in der HashMap des Materials an Stelle i ...
      material = materials.get(i);

      // und entnehme alle Elementes des Materials.
      for(int e = 0; e < material.size; e++)
      {
        // Lese den Wert fuer eine Wellenlaenge l ...
        float value = material.get(e).getValue(l);

        // ... und schreibe diesen Wert in den ByteBuffer.
        bbMaterials.putFloat(value);
      }
    }
    // Quadratische Bloecke: Eventuell leere Stellen auffuellen.
    for(int r = 0; r < (numOlElements - materialElements); r++)
      bbMaterials.putFloat(0.0f);
  }

  // Anzahl der Zeilen und Spalten eines Blocks (quadratisch).
  return numOfColumns;
}
```

Abbildung 4.9: Auszug aus Implementierung der Methode `prepareData`: Materialien werden aus der Hash-Tabelle gelesen und in λ-Blöcke aufgeteilt.

es sich bei dem Element um ein Spektrum, so ist der entnommene Wert wellenlängenabhängig. Hat das Element nur einen einzelnen Wert, so wird dieser für jede Wellenlänge gleich zurückgegeben.
Sind alle Daten für eine Wellenlänge entnommen und in den `ByteBuffer` verpackt, muss nun eventuell der Rest dieses Blocks mit Nullen aufgefüllt werden. Der Grund ist hier die quadratische Anordnung der einzelnen Blöcke. Dies bedeutet zwar, dass Speicherplatz verschenkt wird, kann aber vernachlässigt werden.
Mit einer weiteren Methode kann nun der `ByteBuffer` übergeben werden und steht somit zum Transfer auf die GPU bereit. Die Methode `prepareData()` ist so entworfen, dass sie nicht zwangsläufig das gesamte zu betrachtende Spektrum in einen `ByteBuffer` verpackt. Durch die Angabe der unteren und oberen Grenze könnten die λ-Blöcke einzeln, nach jeder Berechnung auf die GPU transferiert werden. Dies ermöglicht es, Szenen mit sehr vielen verschiedenen Materialien zu berechnen, da nicht alle Materialdaten zeitgleich auf der GPU zur Verfügung stehen müssen.

4.4 Shader-Programme

Nach der Betrachtung der Implementierung auf Seite der CPU ist dieser Abschnitt den Implementierungen der Fragment-Shader-Programme gewidmet. Hier wird größtenteils auf einige Aspekte der `ShadeKernel`-Implementierung eingegangen. Als Einstieg dazu soll zunächst ein Blick auf die `main()` Methode des Kernels gerichtet werden. Im Laufe dieses Abschnitts wird dann auf die einzelnen Aspekte genauer eingegangen. Abbildung 4.10 soll dazu als Ausgangspunkt dienen.
Die erste Aufgabe im `ShadeKernel` besteht darin, die Daten aus den Texturen zu lesen, die in vorangegangen Durchläufen mit Werten aufgefüllt wurden. Diese Daten sind abhängig von der Position des ausgeführten Fragmentes auf der Bildebene. Daher muss zunächst diese Koordinate mit Hilfe von `gl_TexCoord[0].xy` ermittelt und in der Variable `xy` gehalten werden.
Mit dieser Variable werden die Daten, in verschiedenen Texturen vorhanden, ausgelesen. Dies entspricht zum Beispiel der Richtung eines Strahls oder der Schnittpunktkoordinate mit einem Objekt. Ebenfalls wird die aktuelle Farbe des Bildpunktes in einer dieser Texturen gespeichert. Diese Texturen wurden in Abschnitt 2.3.1 als Textur-Sets bezeichnet, da sie Teil der Ping-Pong-Strategie sind.
Als nächstes werden Konstrukte mit Hilfe der gelesenen Daten erzeugt. Die Konstrukte dienen hier lediglich als eine Art Kapselung von Daten, um die Anzahl an Parametern bei Funktionsaufrufen zu reduzieren.
So ist zum Beispiel der aktuelle (eingehende) Strahl `wo` ein solches Konstrukt, dass mit den Daten für Strahl-Ursprung, -Richtung und Beitrag gefüllt wird.
Des Weiteren wird für die Konvertierung von spektralen Intensitäten in XYZ-Farbwerte die Variable `CIE_XYZ` angelegt. Sie enthält zur Ausführung des Kernels die benötigten Farbanpassungsfunktionswerte von $\overline{x}$, $\overline{y}$ und $\overline{z}$. Die Werte werden per `uniform`-Parameter übergeben.
Des Weiteren wird ein `TraceData td` angelegt. Es enthält alle nötigen Daten wie Schnitt-

```
uniform int lambda;                                    // Aktuelle Wellenlänge

void main()
{
  vec2 xy          = gl_TexCoord[0].xy;                // Texturkoordinaten
  vec4 tmpOI0      = texture2DRect(outTex, xy);// Lese in Textur-Set
       ...
  vec3 CIE_XYZ     = vec3(CIE_X, CIE_Y, CIE_Z);// CIE-Farbanpassung-Fkt
    .
  vec4 pixel       = tmpOI0;                           // Bisheriger Farbwert

  LRay wo;                                             // Eingehender Strahl
  TraceData td     = ...                               // Objekt bezogene Daten
  float intensity;                                     // Intensitaet auf S-
    Punkt

  if(td.id >= 0) // Wenn Objekt getroffen, dann ist id groesser 0
  {
    Material mat = getMaterial(td);                    // Material auslesen
    LLight light = ...                                 // Lichtquellenauswahl
    if(!inShadow)                                      // S-Punkt beleuchtet?
    {
      //Evaluierung der BSDFs und Gewichten mit Strahlbeitrag.
      intensity  = sample_BxDFs(wo, light, td, mat) * wo.intensity;

      // Umrechnung der errechneten Intensitaet in einen XYZ-Farbwert
      vec3 xyzCOL= intensity * CIE_XYZ;

      //Normierung dieser Farbe und zum Bildpunkt hinzuaddieren.
      pixel     += (vec4(xyzCOL, 1.0) / superSample) / lambdaSamples;
    }
    // Berechnung des neuen Sekundaer-Strahls
    wi           = sample_Directions(wo, td, mat);
  }
  gl_FragData[0] = pixelColor;                         // Index 0 im FBO
  //Analog fuer Index 1 bis 3 der Textur-Anhaenge im FBO
}
```

Abbildung 4.10: Auszug aus Implementierung `main` Methode des `ShadeKernels`.

punktskoordinate mit einem Objekt, Objekt-Identitätsnummer und den Normalenvektor auf dem Schnittpunkt.
Die Identitätsnummer wird in `td.id` abgelegt und dient als Indikator, ob der Strahl `wo` überhaupt etwas getroffen hat. Ist dem nicht so, ist `td.id` kleiner Null und es gibt nichts zu tun, da dieser Strahl nichts zur Beleuchtung des Bildpunktes beiträgt. Gibt es jedoch mit dem Strahl einen Schnittpunkt, so ist `td.id` größer gleich Null. Nun wird für den

Schnittpunkt das Material aus der Materialtextur gelesen. Die Implementierungsdetails dazu sind in Abschnitt 4.4.1 erläutert.
Anschließend wird das Konstrukt `LLight light` erstellt. Hier werden Daten zu Lichtquellenart, -position, -farbe und -stärke gehalten. Auswahl der Lichtquelle erfolgt über einen mitgeführten Wert `lightID`. Dieser Wert wird bereits beim Schattentest festgelegt und über die beteiligten Textur-Sets zum `ShadeKernel` durchgereicht. Dabei entspricht der Wert der Adresse der Lichtquelle in der Szenen-Textur. Pro Ausführung des `ShadeKernel`s wird nur eine Lichtquelle betrachtet. Wenn mehrere Lichtquellen in der Szene vorhanden sind, wird eine zufällige Auswahl getroffen[1].
Ist eine Lichtquelle ausgewählt, wird nun überprüft, ob besagter Schnittpunkt bezüglich der Lichtquelle im Schatten liegt oder nicht. Die Variabel `inShadow` wurde im vorangegangenen Schattentest während der `IntersectionKernel`-Ausführung ermittelt. Liegt der Punkt im Schatten, wird keine Evaluierung des BSDFs durchgeführt.
Wird der Schnittpunkt beleuchtet, so erfolgt die Evaluierung der BSDF. In diesem Fall wird die Funktion `sample_BxDFs()` aufgerufen. Auf den Inhalt dieser Methode wird in Abschnitt 4.4.2 eingegangen. Als Parameter wird der aktuelle Strahl `wo`, die Lichtquelle `light` und Informationen zum Schnittpunkt `td` sowie die Materialdaten `mat` übergeben.
Die errechnete Intensität auf dem Schnittpunkt wird in einen XYZ-Farbwert umgewandelt. Dazu reicht die Multiplikation von `intensity` und der dreiwertigen Variable `CIE_XYZ`. Dieser Farbwert wird nun normiert auf die Anzahl der Stichproben vom Super-Sampling und Lambda-Sampling. Dies ist notwendig, damit eine größere Anzahl an Stichproben das Bild nicht heller werden lässt. Der normierte Farbwert wird nun auf den Farbwert des Bildpunktes addiert.
Als nächstes muss ein Sekundär-Strahl erzeugt werden. Dies geschieht mit dem Aufruf `sample_Directions()`. Die Parameter entsprechen denen, die auch der Funktion `sample_BxDFs()` übergeben wurden. Ein Sekundärstrahl wird unabhängig davon erzeugt, ob der Schnittpunkt im Schatten liegt oder nicht, daher ist dieser Aufruf der Funktion nicht an die Bedingung `!inShadow` gebunden.
Schlussendlich bleibt nur noch das Schreiben der errechneten Werte in die dafür vorgesehenen vier Texturen des FBO. Hier wird nicht nur die aktuelle Farbe des Bildpunktes sondern auch beispielsweise der neue Ursprung und die neue Richtung des Strahls gespeichert.
Ein weiterer wichtiger Punkt ist, dass bei allen Berechnungen bei der Evaluierung der BSDFs und Erzeugung des Sekundärstrahls die aktuelle Wellenlänge `lambda` berücksichtigt wird. Bei jeder `ShadeKernel`-Ausführung wird diese per `uniform`-Parameter übergeben.

[1] Dies entspricht der Umsetzung einer Strategie zum Evaluieren der BSDFs bezüglich zufällig ausgewählter Lichtquellen. Somit wird vermieden, dass auf jedem gefundenen Schnittpunkt für jede Lichtquelle ein Schattentest sowie eine Evaluierung des BSDFs durchgeführt werden muss. Genauere Informationen dazu gibt es in Veach[30] und Pharr und Humphreys[18].

4.4.1 Auslesen der Materialdaten

In diesem Abschnitt soll nun ein Blick auf die Implementierung des Mechanismus zum Auslesen der Materialdaten geworfen werden. Zunächst werden die dazu benötigten Deklarationen in Abbildung 4.11 dargestellt.

```
int BXDF_LAMBERTIAN    = 0;
int BXDF_GLASS         = 1;
int BXDF_MICROFACET    = 2;

struct Material {
  int bxdf_type;
  float v1;
  float v2;
  float rough;
  float ior;
  float cauchy;
};
```

Abbildung 4.11: Definition der verschiedenen Material-Typen sowie eines `struct`s zum Kapseln mehrerer Material-spezifischer Daten.

Bei der Definition der Materialien ist darauf zu achten, dass die Typisierung der Materialien auf Seiten der GPU mit der Typisierung auf Seiten der CPU übereinstimmt.
Wie bereits erwähnt, wird im `ShadeKernel` vor dem Evaluieren der BSDFs die Funktion `getMaterial()` aufgerufen. Hier wird für ein Objekt, entsprechend der Identität `td.id`, das Material ausgelesen. Die Abbildung 4.12 bietet einen Einblick in die Implementierung dieser Funktion.
Zunächst wird das Farbfeld in der Szenentextur des betroffenen Objektes ausgelesen. Das Ergebnis dieser Leseoperation ist ein RGBA-Farbwert. Ist die x-Komponente dieses Farbwertes negativ, handelt es sich um ein spektrales Material. In diesem Fall wird dieser Wert als Adresse (`matID`) interpretiert und an entsprechender Stelle in der Materialtextur nachgeschlagen. Handelt es sich bei der ausgelesenen Farbe nicht um ein spektrales Material, muss eine Konvertierung von RGB in eine von `lambda` abhängige Intensität erfolgen. Dies geschieht mit Hilfe der Funktion `getReflectanceIntensity()`. Deren Implementierung entspricht im Wesentlichen der Umsetzung des Pseudo-Codes aus Abbildung 2.1.4 in Abschnitt 2.6 und wird daher hier nicht noch einmal aufgeführt.
Da die Elemente eines Materials einzeln ausgelesen werden müssen und die Anzahl dieser von Material zu Material verschieden ist, muss daher bekannt sein, um welchen Typ von Material es sich an der Stelle mit der Adresse `matID` handelt. Dies lässt sich mit Hilfe des Wertes aus der y-Komponente feststellen. Ist die Adresse und der Typ bekannt, werden dementsprechend wiederum Leseoperationen, diesmal auf der Materialtextur, ausgeführt. Dies geschieht mit Hilfe des Aufrufs der Funktion `readFromTex()`.
Die Adresse des Materials entspricht gleichzeitig auch dem ersten Element. Die Reihenfolge der Elemente ist immer gleich, wodurch jedes Element implizit eine feste Positions-

```
Material getMaterial(TraceData td) {

  Material mat;                                  // Erzeugen eines Materials
  vec4 color     = getObjectColor(td.id);// Farbfeld auslesen
  float matID    = abs(color.x) - 1.0;    // Adresse des Materials
  int matType    = int(color.y);          // Typ des Materials

  if(color.x < 0)
  {
    if (matType == BXDF_LAMBERTIAN) {
      mat.bxdf_type = BXDF_LAMBERTIAN;
      mat.v1        = readFromTex(td, matID + 0.0f);

    } else if (matType == BXDF_GLASS) {
      mat.bxdf_type = BXDF_GLASS;
      mat.v1        = readFromTex(td, matID + 0.0f);
      mat.v2        = readFromTex(td, matID + 1.0f);
      // Analog fuer ior und cauchy

    } else if (matType == BXDF_MICROFACET) {
      mat.bxdf_type = BXDF_MICROFACET;
      ...
    }
  } else {
    mat.v1          = getReflectanceIntensity(color.xyz);
  }
  return mat;
}
```

Abbildung 4.12: Auszug aus Implementierung zum Erzeugen der Materialien.

nummer innerhalb dieser Reihenfolge hat. Daher ist das erste Element eines Materials immer in `mat.v1` abgelegt. Die Adressen der nachfolgenden Elemente werden durch die Summe aus `matID` und Positionsnummer bestimmt.
Die Funktion `readFromTex()` führt hierbei die eigentliche Leseoperation aus. Ihr wird neben dem `TraceData` auch die Adresse eines Elements übergeben, das aus der Materialtextur zu lesen ist. Abbildung 4.13 stellt einen Teil der Implementierung dar.
Als erstes muss die Zeile ermittelt werden, in der sich das gesuchte Element befindet. Die Variable `matCols` steht mit dem Erzeugen der Materialtextur fest und kann daher bei der Quelltextbindung des `ShadeKernel` statisch in den Code eingebunden werden. Mit der Division der Adresse des Elements und der Anzahl an Spalten ist die Zeilennummer berechnet. Anschließend kann die Position auf der Zeile ermittelt werden.
Abschließend muss die lokale Zeilenadresse auf den globalen Adressraum angewendet werden. Dazu dient der per `uniform` gesetzte Parameter `matRowOffset`. Der Wert entspricht immer der Startadresse des aktuellen λ-Blocks. Somit muss dieser Wert lediglich auf die Zeilenadresse addiert werden.

```
float readFromTex(TraceData td, float matID ) {

  int line     = int(matID / float(matCols));
  int pos      = int(matID - int(matCols * line));
  line        += matRowOffset;
  float value = texture2DRect(materialTex, vec2(pos, line)).x;

  if(value <= -1.0 )
    // Bild-Textur: UV-Mapping und RGB-Wert konvertiern.

  return value;
}
```

Abbildung 4.13: Auszug aus Implementierung zum Auslesen der Materialien.

Mit der Zeilenadresse und der Position auf der Zeile des Elements ist die Koordinate, in der sich das besagte Element befindet, bekannt und es kann mit dieser Koordinate auf `materialTex` zugegriffen werden.
Ist der ausgelesene Wert jedoch negativ, handelt es sich um eine Bild-Textur. In diesem Fall wird der ausgelesene Wert als Adresse auf eine Textur interpretiert. Außerdem muss eine UV-Koordinate berechnet werden, mit deren Hilfe auf die Bild-Textur zugegriffen wird. Dies entspricht dem bereits in Huwe[8] vorgestellten Prinzip und wird daher hier nicht näher beleuchtet. Einzig eine Konvertierung des gefundenen RGB-Werts in eine von `lambda` abhängige Intensität muss abschließend erfolgen. Dies geschieht ebenfalls mit der Funktion `getReflectanceIntensity()`.
Nachdem alle Daten eines Materials gelesen wurden, liegen alle Informationen nun in Abhängigkeit einer Wellenlänge λ vor. Alle nachfolgenden Funktionen müssen nun keine Unterscheidung mehr zwischen spektralen und nicht-spektralen Daten oder einzelnem Farbwert oder Bild-Textur vornehmen. Somit ist die Sicht auf die Materialdaten abstrahiert.

4.4.2 BSDF-Modelle

In den Erläuterungen zur `main` Funktion des `ShadeKernel`s kommt es zu den Funktionsaufrufen `sample_BxDFs()` und `sample_Directions()`, welche nun etwas genauer betrachtet werden. Das Prinzip der beiden Funktionen ist jeweils ähnlich: Je nach Materialtyp werden wiederum verschiedene Funktionen aufgerufen, die die Eigenschaften des jeweiligen Materials implementieren.
Bevor es zu dem Aufruf der Material-spezifischen Funktionen kommt, muss die Richtung des einfallenden Strahls gedreht und in das lokale Shading- Koordinatensystem transformiert werden. Das Umdrehen der Richtung des Strahls ist der Konvention geschuldet, dass alle Strahlen von einem Punkt auf einer Oberfläche weg zeigen müssen. Abbildung 4.14 stellt die Implementierung der Funktion `sample_BxDFs()` dar.
Mit Hilfe des Normalenvektors (`td.normal`) wird ein Shading-Koordinatensystem er-

```
float sample_BxDFs(LRay ray, LLight light,
                   TraceData td, Material mat) {

  vec3 woW      = -ray.direction;
  ShaderCoor sc = createShadeCoorSys(td.normal);
  vec3 wiW      = normalize(light.origin - td.iPoint).xyz;
  vec3 wo       = globalToLocalVec(sc, woW);
  vec3 wi       = globalToLocalVec(sc, wiW);
  float f       = 0.0;

  if (mat.bxdf_type == BXDF_LAMBERTIAN)
    f           = computeBRDF_Lambertian(mat);
  else if(mat.bxdf_type == BXDF_GLASS)
    f           = 0;
  else if(mat.bxdf_type == BXDF_MICROFACET)
    f           = computeBRDF_Microfacet(wo, wi, mat);

  return f * light.intensity;
}
```

Abbildung 4.14: Funktion `sample_BxDFs` zum Aufruf der verschiedenen BSDFs je nach Materialtyp.

stellt. Dies geschieht wie in Abschnitt 2.2.3 beschrieben. In `ShaderCoor` sind dann drei Vektoren `nn`, `sn` und `tn` abgelegt. Der Vektor `nn` entspricht dem Normalenvektor `td.normal`. Die anderen beiden Vektoren `sn` und `tn` stehen im rechten Winkel zueinander, wobei der Vektor `nn` senkrecht auf der von `sn` und `tn` aufgespannten Ebene steht. Somit sind alle drei Vektoren zueinander orthogonal.
Ist das Shading-Koordinatensystem erzeugt, kann mit Hilfe der in Abbildung 4.15 dargestellten Funktionen eine Transformation zwischen globalem Koordinatensystem und lokalem Shading-Koordinatensystem erfolgen.
So wird die Richtung das Strahls `ray` negiert und in `woW` abgelegt, wobei dieser Vektor noch für das globale Koordinatensystem gilt. Für die Angabe der Richtung in lokalen Shading-Koordinaten erfolgt eine Transformation mit Hilfe der `globalToLocalVec()`. Dieser Wert wird im Vektor `wo` gehalten.
Für die Evaluierung der BSDF ist jedoch noch die Richtung zur Lichtquelle entscheidend. Die Positionen der Lichtquelle sowie des Schnittpunktes sind bekannt und liegen als globale Koordinaten vor. Durch Subtraktion des Schnittpunktes von der Lichtquellenposition und anschließende Normalisierung ist die Richtung zur Lichtquelle bekannt (`wiW`). Auch dieser Vektor wird in das lokale Koordinatensystem transformiert und in `wi` abgelegt.
Nun kann das eigentliche Evaluieren der BSDFs beginnen. Dazu werden die materialspezifischen Funktionen aufgerufen (siehe nachfolgende Abschnitte).
Im Vergleich zur Funktion `sample_BxDFs()` ist der Aufbau der Funktion `sample_Directions()` ähnlich. Der Unterschied liegt jedoch darin, dass der ausgehende

```
vec3 globalToLocalVec(ShaderCoor sc, vec3 vec)
{
  return vec3(dot(vec, sc.sn), dot(vec, sc.tn), dot(vec, sc.nn));
}

vec3 localToGlobalVec(ShaderCoor sc, vec3 vec)
{
  return vec3( sc.sn.x * vec.x + sc.tn.x * vec.y + sc.nn.x * vec.z,
               sc.sn.y * vec.x + sc.tn.y * vec.y + sc.nn.y * vec.z,
               sc.sn.z * vec.x + sc.tn.z * vec.y + sc.nn.z * vec.z);
}
```

Abbildung 4.15: Funktionen zum Transformieren von Vektoren aus dem globalen in das lokale Shading-Koordinatensystem (`globalToLocalVec()`) sowie aus dem lokalen in das globale Koordinatensystem (`localToGlobalVec()`).

Strahl `wi` nicht bekannt ist. Dieser soll mit Hilfe der BSDFs der Materialien gefunden werden. Ist die Richtung gefunden, wird der Beleuchtungsbeitrag, den der Strahl für den Bildpunkt liefert, ausgerechnet. Dies geschieht im Prinzip in der Form, dass mit dem gefundenen Strahl `wi` die BSDF erneut evaluiert wird.
Dieser Beitrag wird ebenso, wie die Richtung oder der Ursprung, mitgeführt. Nach der Erzeugung des neuen Beitrag des Strahls wird dieser auf den alten Beitrag `wi.intensity *= //neuer Beitrag` aufmultipliziert. In der Abbildung 4.10 ist zu sehen, wo dieser Wert zum Einsatz kommt: Der neu erzeugte Strahl `wi` ist im nächsten Rekursionsschritt `wo`. Mit dem Wert `wo.intensity` werden die Beträge der evaluierten BSDFs skaliert.
Nachdem der Beitrag für den Strahl `wi` bestimmt wurde, muss abschließend die Richtung `wi.direction` in das globale Koordinatensystem transformiert werden.

4.4.2.1 Lambertian

Evaluierung Die Theorie zum Gesetz von Lambert wurde bereits in Abschnitt 2.2.3.2 gegeben. Die Evaluierung dieser BRDF ist daher denkbar einfach: Lediglich die Intensität der diffusen Farbe des Materials ist durch π zu dividieren. Abbildung 4.16 zeigt die Implementierung zur Evaluierung der Lambertian BRDF.

```
float computeBRDF_Lambertian(Material mat)
{
  return mat.v1 * _PI;
}
```

Abbildung 4.16: Evaluierung der Lambertian BSDF.

Sekundärstrahlerzeugung Die Erzeugung der Richtung des Sekundärstrahls ist dagegen etwas aufwendiger. Hier wird eine Strategie gewählt, die auf der Beobachtung basiert, dass mit zunehmenden θ_o der Beleuchtungsbetrag abnimmt.
Generell möchte man keine Strahlen erzeugen, die wenig zur Beleuchtung des Bildpunktes beitragen. Würde jetzt zufällig eine Richtung ausgewählt werden, könnte unter Umständen θ_i groß und damit der Beleuchtungsbeitrag auf diesem Punkt klein werden. Da der Wert des Kosinus θ mit zunehmenden Winkel abnimmt, werden also Strahlen bevorzugt gesucht, die einen möglichst großen Wert $cos\,\theta$ liefern.
Diese Strategie wurde mit der Funktion `CosineSampleHemisphere()` realisiert. Sie bekommt als Parameter zwei Zufallszahlen übergeben. Das Ergebnis ist dann eine Richtung, wobei solche Richtungen bevorzugt werden, die ein kleines θ_i liefern. Abbildung 4.16 stellt die Implementierung der Funktion zur Sekundärstrahlerzeugung dar.

```
vec3 computeDir_Lambertian(vec3 wo)
{
  vec2 uv = rand2D();
  float u = uv.x;
  float v = uv.y;
  vec3 wi = CosineSampleHemisphere(u, v);

  return wi;
}
```

Abbildung 4.17: Erzeugung der Richtung des Sekundärstrahls.

4.4.2.2 Glas

Evaluierung Das Material Glas setzt sich aus den beiden Komponenten der spekularen Reflexion und Transmission zusammen. Wie in den dazugehörigen Abschnitten 2.2.3.3 und 2.2.3.4 erläutert, erfolgt keine Evaluierung dieser BRDF beziehungsweise BTDF, da es praktisch gesehen keine Möglichkeit gibt aus einer gegebenen Richtung die Lichtquelle mit einem spekular reflektierten Lichtstrahl zu treffen. Daher ist der Wert der Evaluierung für Glas Null.

Sekundärstrahlerzeugung Wie bereits erwähnt setzt sich das Glas-Material aus zwei Komponenten zusammen. Da jedoch immer nur eine Strahlrichtung weiterverfolgt werden kann, muss also zuvor eine Wahl getroffen werden, welche dieser beiden Komponenten zur Anwendung kommen soll. Dies kann mit Hilfe einer Zufallszahl geschehen. Der Wert dieser reellen Zahl liegt im Bereich zwischen 0 und 1. Damit beide Komponenten mit gleicher Wahrscheinlichkeit ausgewählt werden können, wird dieser Wertebereich halbiert, also in zwei Bereiche eingeteilt. Per Definition wird nun die Reflexion zum ersten, die Transmission zum zweiten Bereich zugeordnet.
Mit Hilfe der Zufallszahl kann nun die Auswahl der Komponente stattfinden: ist der Wert dieser Zahl kleiner 0.5, wird der Strahl reflektiert sonst transmittiert. Durch die

Verwendung der Zufallszahl und Einteilung in zwei Bereiche haben beide Komponenten die selbe Wahrscheinlichkeit ausgewählt zu werden.
Nun soll ein Blick auf die Implementierung dieser Komponenten geworfen werden. Mit der Funktion `computeDir_SpecularReflection()` soll die Reflexion als erstes betrachtet werden. Die Implementierung ist in Abbildung 4.18 zu sehen.

```
vec3 computeDir_SpecularReflection(Material mat, vec3 wo,
                                   inout float f)
{
  float specular = mat.v1;
  float ior      = mat.ior;
  vec3 wi        = wo;
  wi.x          *= -1.0;
  wi.y          *= -1.0;
  float F_rs     = fresnel_dielectric(CosTheta(wo), 1.0, ior);
  f              = F_rs * specular / abs(CosTheta(wi));

  return wi;
}
```

Abbildung 4.18: Erzeugung des Sekundärstrahls, der in eine Richtung ideal spekular reflektiert wird.

Neben dem Material und dem einfallenden Strahl wird ebenso ein Parameter `inout float f` übergeben. Für die Berechnung des Strahlbeitrages wird `f` aufgefüllt und ist somit für den Aufrufer dieser Funktion nutzbar. Wie schon in Abschnitt 2.2.3.3 zur Reflexion erläutert, ist die Erzeugung des reflektierten Strahls `wi` lediglich eine Spiegelung von `wo` um die Normale. Abschließend wird mit dem Richtungen `wo` und `wi` die BSDF $f_r(p, \omega_i, \omega_o)$ mit Hilfe einer dielektrischen Frenselfunktion evaluiert und in `f` abgelegt.
Analog zur Reflexion ist die Implementierung der idealen spekularen Transmission eine direkte Umsetzung der Gleichung aus Abschnitt 2.2.3.4. Daher wird hier nicht mehr auf jede Zeile der in Abbildung 4.19 dargestellten Implementierung eingegangen.
Dennoch gibt es hier eine Besonderheit, die zum Erreichen eines Dispersionseffektes essentiell ist: Die Implementierung der Cauchy-Gleichung. Zunächt gilt die Festlegung, dass jedes Objekt von Luft umgeben ist. Somit ist η_i (`ei`) gleich eins. Der Brechungsindex `ior` des Objektes entspricht η_t (`et`). Ist ein Cauchy-Wert `cb` größer 0, so kommt die Dispersionsgleichung nach Cauchy zum Einsatz[2].
Die Zeile Code bewirkt, dass der Brechungsindex η_t mit wachsender Wellenlänge abnimmt. Durch den sich veränderten Index wird der Strahl in eine andere Richtung abgelenkt. Dadurch entstehen Regenbogeneffekt da die Lichtstrahlen in Abhängigkeit ihrer Wellenlänge durch das Medium unterschiedlich transmittiert.

[2]Die Implementierung wurde aus dem Quellcode des *Lux Render* entnommen. Dies ist ein quelloffenes Projekt, dass auf Grundlage des PBRT-Renderers entwickelt wird. Link: `http://www.luxrender.net/` (Letzter Zugriff: 14.05.2009)

```
vec3 computeDir_SpecularTransmission(Material mat, vec3 wo,
                                     inout float f)
{
  float specular        = mat.v2;
  float ior     = mat.ior;
  float cb      = mat.cauchy;
  vec3 wi       = vec3(0.0);
  bool entering = CosTheta(wo) > 0.;
  float ei      = 1.0;
  float et      = ior;

  if(cb != 0.0)
    et        += (cb * 1E6) / (lambda * lambda);

  if (!entering)
    swapIOR(ei, et);

  float sini2 = SinTheta2(wo);
  float eta   = ei / et;
  float sint2 = eta * eta * sini2;

  if (sint2 > 1.)
    return vec3(0.);

  float cost = sqrt(max(0.f, 1.f - sint2));
  if (entering)
    cost      = -cost;

  float sintOverSini = eta;
  wi          = vec3(sintOverSini * -wo.x,
                     sintOverSini * -wo.y,
                     cost);
  float F_r   = fresnel_dielectric(CosTheta(wo), 1.0, ior);
  f           = (ei*ei)/(et*et) * (1.0 - F_r) * specular / abs(
     CosTheta(wi));

  return wi;
}
```

Abbildung 4.19: Erzeugung des Sekundärstrahls, der in eine Richtung ideal spekular transmittiert.

4.4.2.3 Microfacetten

Evaluierung Als letztes Material ist nun noch die Implementierung der Microfacet BRDF zu betrachten. Auch diese entspricht der direkten Umsetzung in Abschnitt 2.2.3.5 gegebenen Gleichungen. Damit die Angabe der Rauheit intuitiver ist, wird in der GUI

der reziproke Wert des `exponent` angegeben. Somit kann man den Parameter als Rauheitsparameter bezeichnen, denn je kleiner der Wert der Rauheit, desto glatter ist die Oberfläche. Abbildung 4.20 stellt die Umsetzung der BRDF des Microfacetten-Materials dar.

```
float computeBRDF_Microfacet(vec3 wo, vec3 wi, Material mat)
{
  float specular  = mat.v1;
  float exponent  = 1.0 / mat.urough;
  float n         = mat.ior;
  float cosThetaO = abs(CosTheta(wo));
  float cosThetaI = abs(CosTheta(wi));
  vec3 wh         = normalize(wi + wo);
  float cosThetaH = dot(wi, wh);
  float cG        = G(wo, wi, wh);
  float d         = blinn_D(wh, exponent);
  float F_r       = fresnel_dielectric(cosThetaH, 1.0, n);

  return specular * d * cG * F_r / (PI * cosThetaI * cosThetaO);
}
```

Abbildung 4.20: Evaluierung des Microfacetten-Modells.

```
vec3 computeDir_Microfacet(vec3 wo, inout vec3 wi, Material mat,
    inout float f)
{
  float specular  = mat.v1;
  float exponent  = 1.0 / mat.urough;
  float n         = mat.ior;
  vec3 wh         = blinn_getHalfwayVec(exponent, wo);
  wi              = -wo + 2.f * dot(wo, wh) * wh;
  f               = computeBRDF_Microfacet(wo, wi, mat);

  return normalize(wi);
}
```

Abbildung 4.21: Erzeugung des Sekundärstrahls mit Hilfe einer Blinn-Microfacettenverteilung.

Sekundärstrahlerzeugung Die Erzeugung der Sekundärstrahlen wird mit Hilfe der Microfacetterverteilung umgesetzt. Mit der Funktion `blinn_getHalfwayVec()` wird in Abhängigkeit des Exponenten und der einfallenden Strahlrichtung `wo` eine Facette – beliebig orientiert – erzeugt. Je nach Größe des Betrags von `exponent` entspricht die Ausrichtung mehr oder weniger der Ausrichtung der Gesamtoberfläche. Anders ausgedrückt: Wie weit

weicht der Normalenvektor der Facette vom Normalenvektor der gesamten Oberfläche ab?
Mit Hilfe der Facetten-Ausrichtung `wh` wird dann die reflektierte Richtung `wi` erzeugt. Abbildung 4.21 zeigt die Implementierung der Funktion `computeDir_Microfacet()`.

5 Ergebnis und Bewertung

In diesem Kapitel soll die Leistungsfähigkeit des spektralen Raytracers Sunshine mit anderen Raytracern verglichen werden. Dazu wird im folgenden Abschnitt eine Auswahl für diesen Vergleich vorgestellt.
Nach dem Vorstellen der Auswahl soll die Geschwindigkeit der spektralen CPU Raytracer mit der des spektralen GPU Raytracer verglichen werden. Renderer miteinander zu vergleichen, ist grundsätzlich eine schwierige Aufgabe. Nicht selten sind mit ihnen unterschiedliche Verfahren umgesetzt worden. Da diese unterschiedliche Resultate liefern können, sind Vergleiche auf Messdaten basierend nicht immer sinnvoll. Daher soll mit Hilfe des subjektiven Eindrucks verglichen werden.
Im anschließenden Abschnitt wird Sunshine mit einem nicht-spektralen Raytracer verglichen.

5.1 Renderer-Auswahl für Vergleich

Bei der Auswahl der Renderer ist zu beachten, dass diese auf einem Path Tracing Verfahren basieren beziehungsweise dieses in den Einstellungen auszuwählen ist. Andere Verfahren könnten das Rauschverhalten verändern, was das Ziehen von Vergleichen somit erschwert. Außerdem sollten möglichst keine Filter zur Glättung des berechneten Bildes gentuzt werden, da auch hiermit Rauschen unterdrückt werden kann.
Der bereits angesprochene *Lux Render*[1] ist eine Weiterentwicklung des in Pharr und Humphreys[18] vorgestellten Renderers und unterstützt eine komplette spektrale Darstellung der Farben. Somit sind wellenlängenabhängige Effekte mit diesem Renderer darstellbar.
Ein weiterer spektraler Raytracer ist der *Indigo Renderer*[2]. Dieser unterstützt ebenfalls spektrales Raytracing und ist ebenfalls frei erhältlich, jedoch im Gegensatz zum Lux Renderer nicht quelloffen.

5.2 Qualitäts- und Geschwindigkeitsvergleich

Der in der Motivation genannte Geschwindigkeitsvorteil gegenüber CPU-basierten Lösungen soll nun durch einen Vergleich zwischen dem spektralen GPU Raytracer und den beiden genannten CPU-basierten Renderern gezeigt werden. Dazu wird eine *Cornell-Box*[3] Szene erstellt, die als Referenz dienen soll. Für diese Szene liegen die genauen

[1] `http://www.luxrender.net`, Letzter Zugriff: 22.05.2009
[2] `http://www.indigorenderer.com`, Letzter Zugriff: 22.05.2009
[3] `http://www.graphics.cornell.edu/online/box/data.html`, Letzter Zugriff: 22.05.2009

Grafikkarte	Betriebssystem	Prozessor	RAM
* NVIDIA GeForce 8800 GT, 512 MB	Windows XP 32 Bit, SP2	AMD Dual Core Prozessor 5000+, 2.61 GHz	3 GB
NVIDIA GeForce GX 260, 512 MB	gentoo 2.6.26 R3 64 Bit	Intel(R) Core(TM)2 Quad CPU Q9450, 2.66GHz	4 GB
ATI Mobility Radeon X1600, 256 MB	Windows XP 32 Bit, SP3	Intel(R) Core(TM)2 CPU T7400, 2.66GHz	3 GB
NVIDIA GeForce 7600 GT, 512 MB	Windows XP 32 Bit, SP2	AMD Dual Core Prozessor 1600+, 1.4 GHz	2 GB

Tabelle 5.1: Auflistung der verwendeten Grafikkarten mit den jeweiligen Systemkomponenten. Die mit einem * gekennzeichnete Zeile ist die Konfiguration, die als Referenzsystem während der Vergleiche dient.

Spezifikationen zu Positionen und Größen der Objekte vor. Außerdem liegen alle in der Szene verwendeten Farben als Spektren vor und müssen daher nicht umgerechnet werden.
Die Systeme, der in den Vergleichen verwendeten Grafikkarten, besitzen die in der Tabelle 5.1 angegebenen Konfigurationen. Alle CPU-Renderer wurden ausschließlich auf dem System der Referenzkarte getestet.

5.2.1 Referenzbilder

Zum Erstellen der Referenzbilder werden auf dem eben genannten PC die Einstellungen so gewählt, dass die festgelegten Zeitvorgaben erreicht werden, wobei die Dauer der Initialisierungs- und Finalisierungsphasen mit enthalten ist. Für alle erstellten Referenzbilder, wurden die nun folgenden Einstellungen verwendet:

- Supersampling 3x3
- Rekursionstiefe 5
- Untergrenze Spektrum: 400nm
- Obergrenze Spektrum: 700nm.

Zum Einhalten der Zeitvorgaben werden nur die zur Integration über das Spektrum benötigten Schrittweiten angepasst. Diese Werte sind in der mittleren Spalte der Tabelle 5.2 aufgeführt. Mit der Angabe der Schrittweite ist wiederum die Anzahl der zu betrachtenden Wellenlängen festgelegt. Multipliziert man die Anzahl mit den 9 Pfaden, die durch das 3x3-fache Supersampling entstehen, ergibt sich die Gesamtanzahl an Pfaden pro Bildpunkt über alle Wellenlängen.

Bild Nr.	Zeit	Schrittweite der Spektrum-Integration [nm]	Anzahl an Pfaden pro Bildpunkt
1	15 Sekunden	53	54
2	30 Sekunden	24	117
3	1 Minute	10	270
4	5 Minuten	2	1350
5	10 Minuten	1	2700

Tabelle 5.2: Auflistung der Schrittweiten zur Integration über das vorgegebene Spektrum und der daraus resultierenden Anzahl an Pfaden pro Bildpunkt.

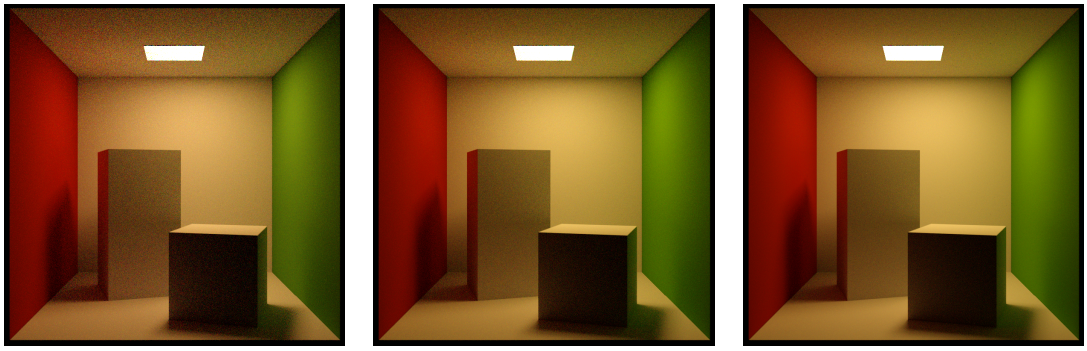
Zeit	Sunshine	Lux
15 Sekunden	54	2
30 Sekunden	117	5
1 Minute	270	10
5 Minuten	1350	47
10 Minuten	2700	94

Tabelle 5.3: Auflistung der Anzahl verfolgter Pfade (Samples) pro Bildpunkt für die Raytracer Sunshine und Lux.

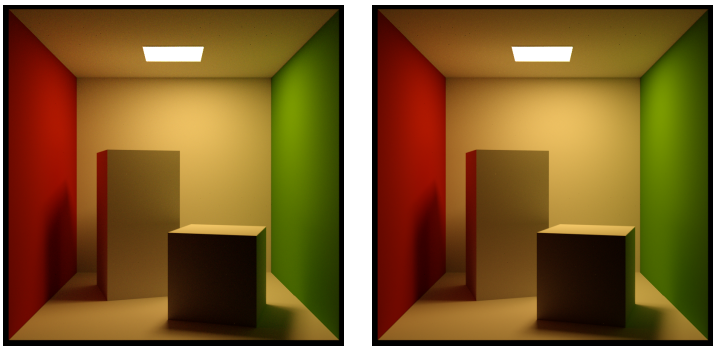
In Abbildung 5.1 sind die Bilder mit den entsprechenden Zeiten dargestellt. Mit allen anderen Renderern werden bezüglich der Zeitvorgaben die Bilder in einer Auflösung von 800x800 erstellt.
Damit die Unterschiede der Bildqualitäten besser zu erkennen sind, wird jeweils ein Ausschnitt vergrößert dargestellt. Dieser ist in allen dargestellten Bildern gleich. Abbildung 5.2 verdeutlicht diesen Sachverhalt.

5.2.2 Vergleich mit spektralen CPU Raytracern

Die Ergebnisse der Renderer zu den jeweiligen Zeitvorgaben sind in der Abbildung 5.3 für den Lux Render sowie in Abbildung 5.4 für den Indigo Renderer dargestellt.
Bereits auf dem ersten Blick fällt auf, dass die mit Sunshine erstellten Bilder weniger rauschen als die von den Renderern Lux und Indigo. Damit die Qualitätsunterschiede besser zu erkennen sind, werden die bereits erwähnten Bildausschnitte miteinander verglichen. Dieser Vergleich ist in Abbildung 5.5 dargestellt.
Es ist deutlich zu erkennen, dass zu jedem der fünf vorgegebenen Zeitpunkte Sunshine das geringste Rauschen im Bild aufweist und somit die beste Bildqualität im Vergleich zu Lux und Indigo bietet. Dies liegt daran, dass im Vergleich zu den CPU-Varianten in der gleichen Zeit mehr Pfade erzeugt werden konnten, was zu einer Reduzierung des Rauschens führt. In Tabelle 5.3 werden die Anzahl der verfolgten Pfade pro Bildpunkt zwischen Sunshine und Lux Render verglichen.



(a) 15 Sekunden (b) 30 Sekunden (c) 1 Minute



(d) 5 Minuten (e) 10 Mintuen

Abbildung 5.1: Mit dem spektralen GPU Raytracer Sunhine erstellte Bilder. Unter jedem Bild ist die Dauer der Erstellung mit aufgeführt.

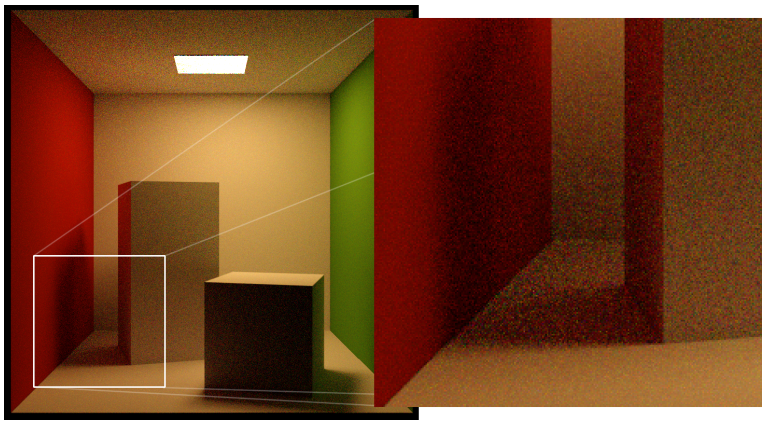
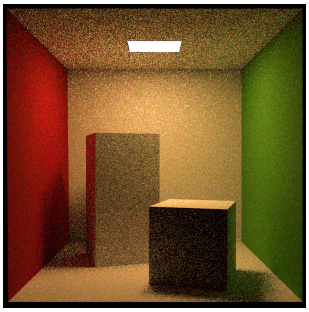
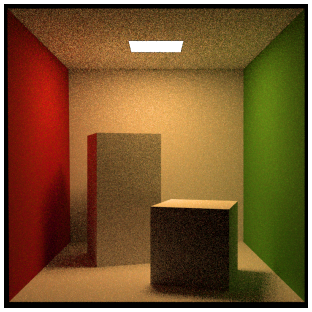


Abbildung 5.2: Durch Vergrößerung ist das Bildrauschen besser zu erkennen.

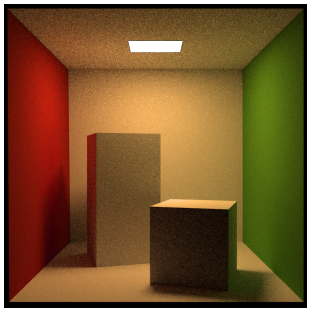
Mit den Daten aus dieser Tabelle ist festzuhalten, dass Sunshine in der selben Zeit wie Lux um den Faktor 28 mehr Pfade verfolgen konnte. Dennoch ist der Unterschied zwischen den Bildern Nummer 5 von Sunshine und Nummer 5 von Lux nicht so groß, wie



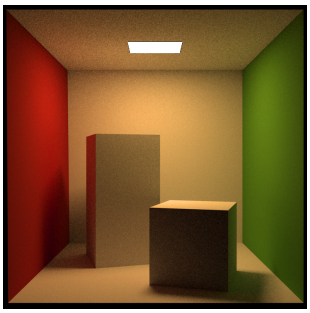
(a) 15 Sekunden



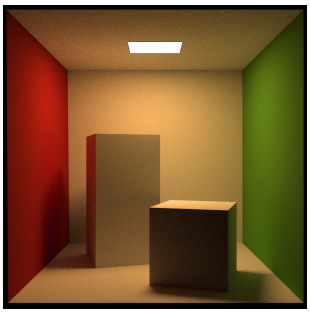
(b) 30 Sekunden



(c) 1 Minute



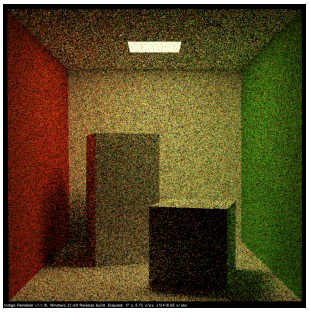
(d) 5 Minuten



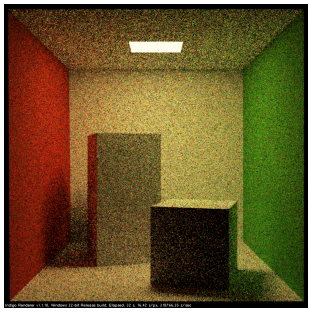
(e) 10 Mintuen

Abbildung 5.3: Mit Lux Render erstellte Bilder.

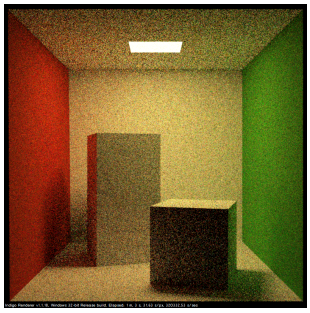
dieser Faktor vermuten lassen würde. Dies lässt auf eine bessere Umsetzung der Varianzreduktion auf Seiten des Lux Renderer schließen. Das bedeutet, hier wäre Sunshine noch optimierungsfähig.
Die Qualität des mit Lux erstellten Bildes Nummer 5 ist qualitativ vergleichbar mit der Bildnummer 2 des spektralen Sunshine Renderers. Deshalb lässt sich feststellen, dass Sunshine für diese Art von Szenen etwa 20 mal schneller Bilder berechnen kann, als die CPU Lösung des Lux Renderer.
Die schlechteste Bildqualität lieferte der Indigo Renderer, da die Bilder zu den jeweiligen Zeitpunkten erheblich verrauschter sind als die von Sunshine und Lux. An dieser Stelle wäre die Anzahl an Pfaden pro Bildpunkt im Vergleich zu den anderen Renderern interessant. Hierzu bietet Indigo zwar einen Wert für die erreichte Anzahl an Samples pro Bildpunkt an, doch wird dieser anscheinend anders ermittelt als oben beschrieben. Aus diesem Grund tauchen in der Tabelle 5.3 keine Werte von Indigo auf, da sie in diesem Zusammenhang keine Aussagekraft besitzen.
Betrachtet man die Qualität des vom Indigo Renderer erstellten Bildes Nummer 5, fällt das starke Rauschen auf. Das nach 15 Sekunden erstellte Bild von Sunshine hat zu diesem Zeitpunkt bereits weniger Rauschen und somit eine bessere Qualität. Das würde bedeuten, dass Sunshine im Vergleich sogar ungefähr 40 mal schneller ein Bild ähnlicher Qualität erzeugen kann.



(a) 15 Sekunden



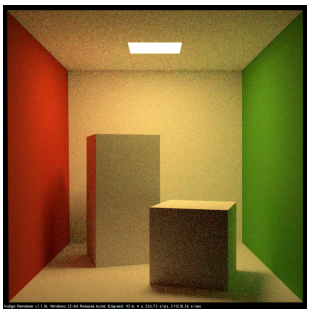
(b) 30 Sekunden



(c) 1 Minute



(d) 5 Minuten



(e) 10 Mintuen

Abbildung 5.4: Mit Indigo erstellte Bilder.

5.2.3 Vergleich mit anderen GPU

Neben der Zeitmessung auf einer NVIDIA GeForce 8800 GT wurden auch andere GPU-Vertreter getestet und in der Tabelle 5.4 aufgelistet. Der aktuellen und leistungsstarken GeForce GX 280 sind zwei Vertreter älterer Generationen von Grafikkarten gegenübergestellt. Dazu wurden ebenfalls jeweils fünf Bilder mit verschiedenen Vertretern, unter Verwendung der genannten Einstellungen, berechnet.
Es fällt auf, dass im Vergleich zur Referenz-GPU GeForce 8800 GT, die aktuellere GeForce GX 280 mehr als doppelt so schnell ist. Deutlich ist auch der Unterschied zu den älteren Grafikkarten, die in diesem Vergleich 20 - 25 mal langsamer sind als die Referenz-GPU. Somit befinden sich die älteren Grafikkarten auf ungefähr dem Niveau des Lux Renderer, sind jedoch deutlich schneller als der Indigo Renderer.
Für den letzten Vergleich dieser drei Renderer dient eine Testszene, die etwas komplexer ist als die Cornell-Box Szene. Sie beinhaltet nicht nur ideal diffuse reflektierende, sondern auch verschmiert-spiegelnde Oberflächen. Auch diese Szene wurde mit allen drei genannten Renderern getestet.
In der Szene ist die *MacBeth* Farbtafel nachgebaut, welche beispielsweise zum Justieren von Kameras genutzt wird. Die spektrale Zusammensetzung der einzelnen Elemente dieser Farbtafel ist bekannt und kann daher in der Szenenbeschreibung jeweils mit verwendet werden. In der Abbildung 5.6 sind die Ergebnisse der drei Renderer dargestellt.

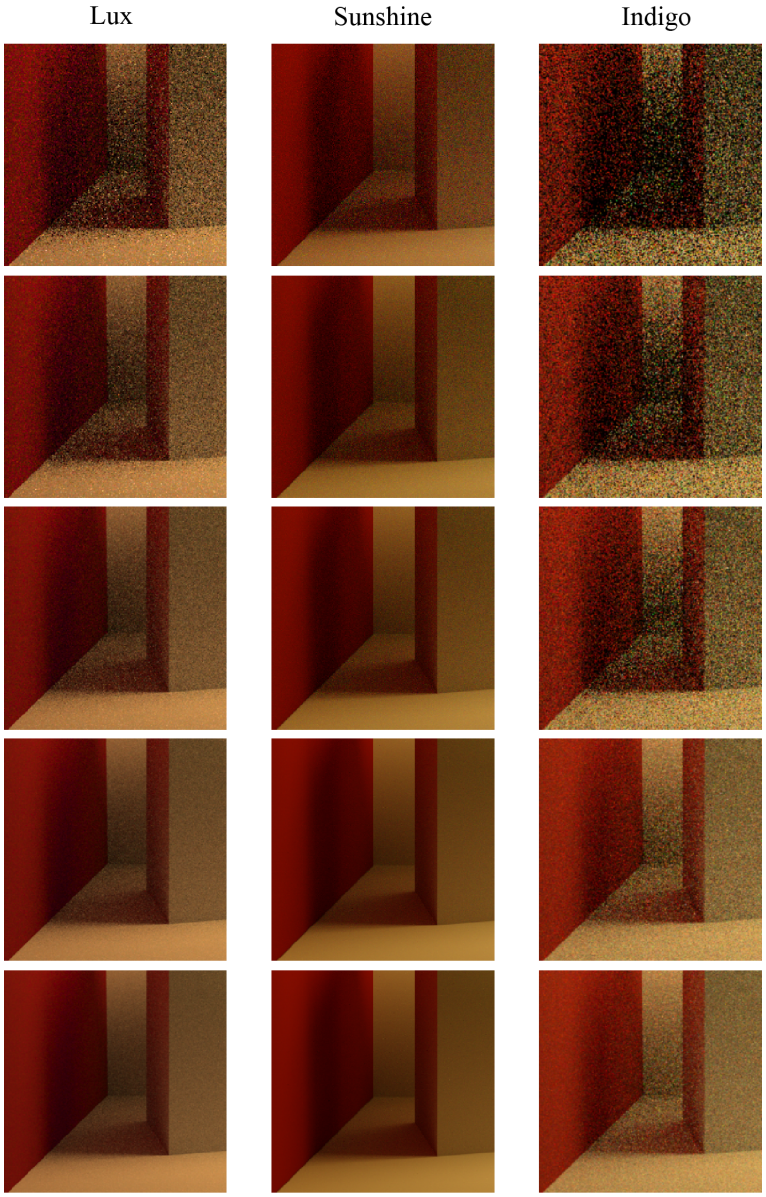
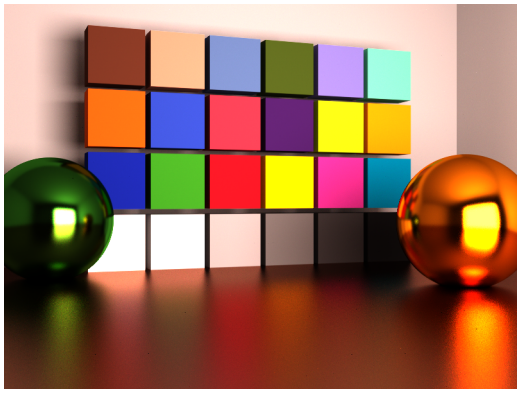


Abbildung 5.5: Vergleich der drei Renderer. Von oben nach unten: 15 Sekunden, 30 Sekunden, 1 Minute, 5 Minuten und 10 Minuten. Deutlich zu erkennen ist, dass die Bilder von Sunshine zu jedem Zeitpunkt weniger Rauschen besitzen.

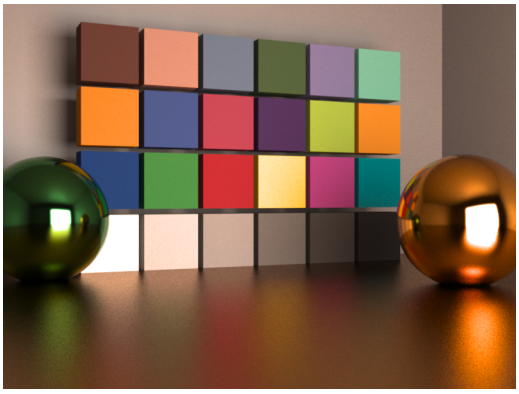
Die beiden CPU Renderer Lux und Indigo benötigten ungefähr 4 Stunden Zeit, um die Bilder zu erstellen. Das qualitativ vergleichbare Bild von Sunshine konnte auf der Referenzkarte in nur 13 Minuten erstellt werden.

GPU	Bild 1	Bild 2	Bild 3	Bild4	Bild 5
NVIDIA GeForce GX 260	8 Sek.	15 Sek.	32 Sek.	2:27 Min.	4:51 Min.
ATI Mobility X1600	3:38 Min.	7:44 Min.	17:42 Min.	87:39 Min.	173:14 Min.
NVIDIA GeForce 7600 GT	4:59 Min.	10:43 Min.	24:11 Min.	125:01 Min.	257:56 Min.

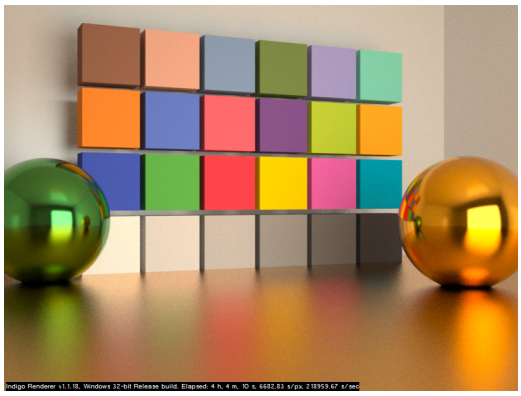
Tabelle 5.4: Auflistung der jeweils benötigten Zeit zum Erstellen der Bilder zu den vorgegebenen Zeiten. Die Einstellungen für die einzelnen Bilder entsprechen denen aus Tabelle 5.2.



(a) Sunshine



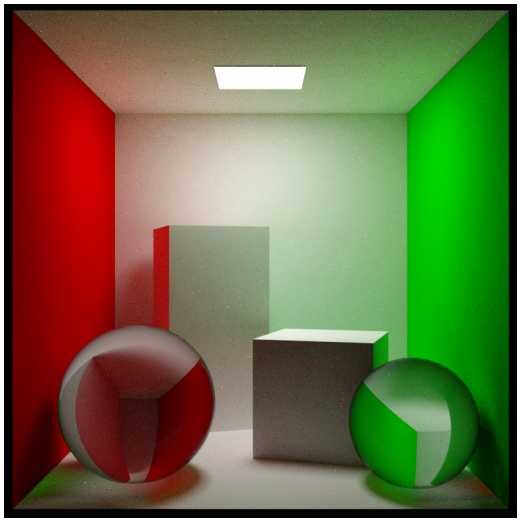
(b) Lux



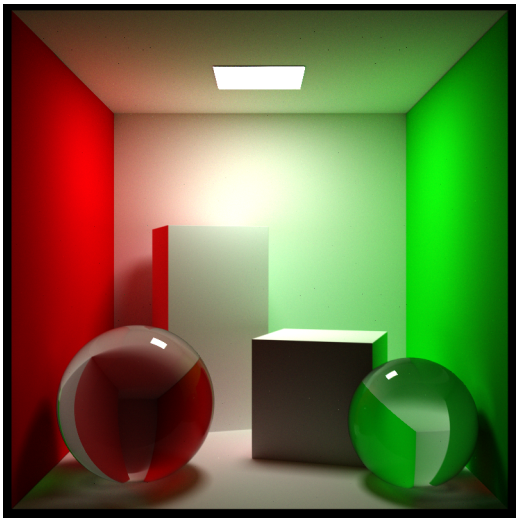
(c) Indigo

Abbildung 5.6: MacBeth Farbtafel mit den Renderern Sunshine, Lux und Indigo erstellt.

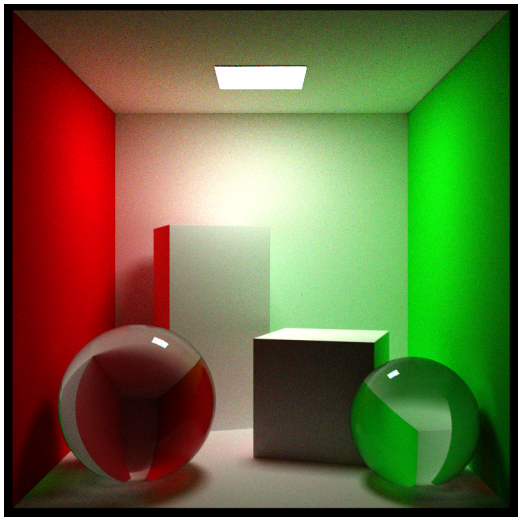
Als einen letzten Vergleich dienen die in Abbildung 5.7 gezeigten Bilder. Es wurde die Cornell-Box Szene modifiziert und um zwei Glaskugeln erweitert. Die Szene wurde an-



(a) RGB-basiertes Verfahren: 400 Pfade (20x20 Supersampling), 2:04 Min.



(b) Spektrales Verfahren: 4800 Pfade (20x20 Supersampling * 12 Wellenlängen pro Pfad)



(c) Spektrales Verfahren: 432 Pfade (6x6 Supersampling * 12 Wellenlängen pro Pfad)

Abbildung 5.7: Vergleich zwischen dem RGB-basierten und spektralen Verfahren des Renderers Sunshine.

schließend mit dem RGB-basierten (siehe Abbildung 5.7(a)) und dem spektralen Verfahren (siehe Abbildungen 5.7(b) und 5.7(c)) von Sunshine in einer Auflösung von 800x800 auf der Referenz-GPU berechnet. Das Supersampling ist bei beiden Varianten auf 20x20 eingestellt. Somit werden pro Bildpunkt 400 Pfade erzeugt.
Da bei dem spektralen Raytracing für jeden Pfad nur eine Intensität berechnet wird,

müssen daher mehrere Pfade erstellt werden, um einen Farbwert zu erhalten. Dies hängt von der Schrittweite, mit der über das Spektrum integriert wird, ab. Für das Bild in der Abbildung 5.7(b) wurde das Spektrum von 400nm bis 700nm begrenzt und eine Schrittweite von 25nm eingestellt. Damit sind 12 Pfade zu erzeugen, um einen Farbwert zu erhalten. Dies bedeutet, dass bei gleichem Supersampling der Aufwand 12 mal größer ist als beim RGB-basierten Raytracing. Außerdem müssen $400 * 12 = 4800$ Pfade pro Bildpunkt verfolgt werden.
Dies ist ein großer Nachteil des spektralen Raytracings, da der Aufwand, um die gleiche Bildqualität zu erreichen, größer ist.
Zum besseren Vergleich wurde das Bild 5.7(c) das Supersampling des spektralen Renderers angepasst, so dass die Gesamtanzahl an Pfaden pro Bildpunkt des spektralen etwa dem des RGB-basierten Verfahren entspricht. Dadurch ist auch im Vergleich zu Bild 5.7(a) ein stärkeres Rauschen zu erkennen.

5.3 Dispersive Effekte

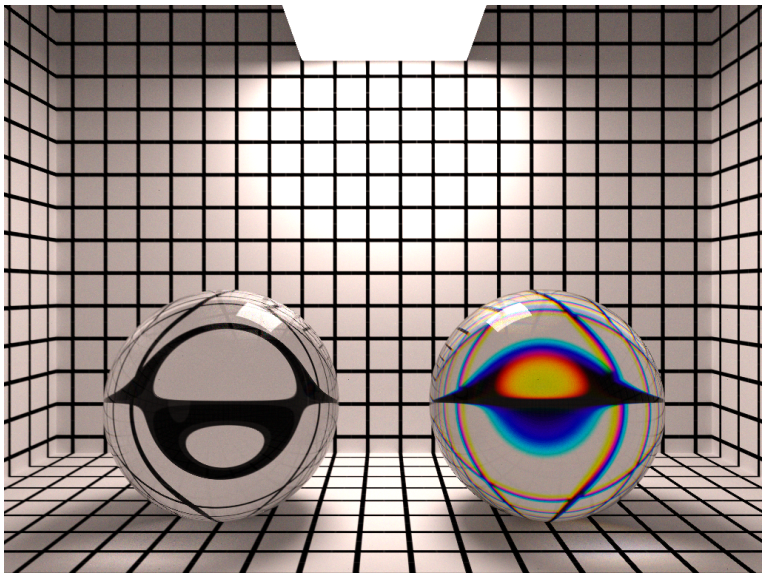


Abbildung 5.8: Beispiel zur Verdeutlichung des Dispersionseffektes: Beide Kugeln brechen das Licht, wobei die rechte Kugel das Licht in Abhängigkeit seiner Wellenlänge bricht, wodurch Dispersion entsteht.

Die Dispersion ist eine der Eigenschaften der spektralen Strahlenverfolgung, die sich mit diesem Verfahren darstellen lässt. Die Spektralzerlegung von weißem Licht in einem Diamanten oder hinter einem Prisma lässt sich nicht ohne größeren Aufwand mit einem RGB basierten Verfahren umsetzen. Daher wird in diesem Abschnitt der Dispersionseffekt getestet.

Zunächst wird der Unterschied zwischen dispersiver und nicht-dispersiver Lichtbrechnung an einem Beispiel verdeutlicht. In der Abbildung 5.8 ist dieser Unterschied zu erkennen.
Beide Kugeln streuen das Licht in Abhängigkeit von einem angegebenen Brechungsindex. Der rechten Kugel wurde zusätzlich ein Dispersionsparameter zugewiesen, so dass sich der Brechungsindex in Abhängigkeit zur Wellenlänge ändert. Dadurch entsteht der für diesen Effekt typische Regenbogeneffekt. Dieser ist deutlich in der rechten Kugel zu erkennen.
Am besten lässt sich mit diesem Effekt ein Diamant darstellen. Dieser hat naturgemäß einen hohen Brechungsindex und Dispersionswert. In der Abbildung 5.9 sind mit Sunshine erstellte Bilder von Diamanten dargestellt.



Abbildung 5.9: Darstellung von Diamanten mit Dispersionseffekten. Alle Bilder wurden mit Sunshine erstellt.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

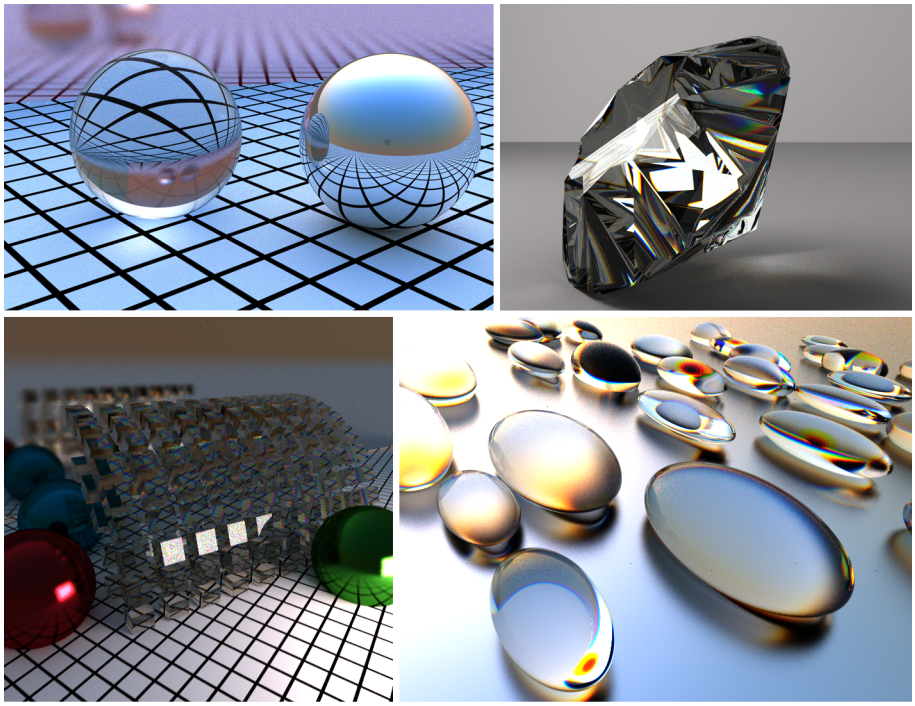
Ziel dieser Arbeit war es, einen spektralen Strahlenverfolger auf der GPU auszuführen. Dazu wurden die benötigten Konzepte vorgestellt und umgesetzt. Ebenso wie die RGB-basierte Variante von Sunshine ist der spektrale Renderer ein vollständig integrierter Bestandteil der Modellierplattform GroIMP.
Die Realisierung baute auf eine bereits bestehende Umsetzung eines stochastischen Strahlenverfolgers auf und wurde um die Komponenten zur Behandlung von Spektren erweitert. Daher ist nun das Verfolgen der Strahlen wellenlängenabhängig. Dies ermöglicht das Darstellen verschiedener in der Natur beobachtbarer Effekte. Beispielhaft wurde dies am Dispersionseffekt demonstriert.
Jeder Strahl codiert nun die Farbinformationen nicht länger im RGB-Farbraum, sondern in Form einzelner Intensitäten bezüglich einer Wellenlänge. Mehrere dieser Intensitätswerte werden in Spektren zusammengefasst. Während des Renderprozesses werden für die festgelegten Wellenlängen eines zuvor definierten Wellenlängenbereichs, Intensitäten im Bildpunkt ermittelt. Somit ist der Renderprozess als eine Integration über das angegebene Spektrum aufzufassen.
Die durch das Path Tracing entstehenden Pfade werden in XYZ-Farben transformiert und im FBO auf der Grafikkarte aufsummiert. Nachdem alle Pfade für alle Wellenlängen verfolgt wurden, kann abschließend die Transformation in den RGB-Farbraum erfolgen.
Neben der Umsetzung der Ablaufsteuerung war ein weiterer wichtiger Punkt die Organisation von Materialien. Da mehr Informationen zu speichern sind, sind mussten Konzepte zur Bereitstellung dieser Daten entwickelt und umgesetzt werden. Durch die Blockzerlegung in der Material-Textur ist das Auslesen der Materialdaten auf der GPU relativ einfach.
Diese Organisation war notwendig, da einige verschiedene Beleuchtungsmodelle implementiert werden mussten. Da sich das bisher umgesetzte Phong-Modell als wenig korrekt erwies, wurden physikalisch genauere Modelle, wie in PBRT[18] vorgestellt, in den spektralen Raytracer Sunshine aufgenommen.
Für die Integration dieser Modelle und der daraus resultierenden Materialien wurden für GroIMP eigens Shader definiert. Mit ihnen lassen sich nun Materialien für den spektralen Raytracer erzeugen, die auf physikalisch genauen Beleuchtungsmodellen basieren. Die Auswahl an Materialien kann durch die Art der Organisation leicht erweitert werden.
In der Auswertung wurde gezeigt, dass aktuelle Grafikkarten gegenüber CPU basierten Lösungen wesentlich mehr Strahlen pro Bildpunkt in einer vorgegebenen Zeit verfolgen können. Selbst ältere Generationen von GPUs sind zumindest gleich schnell. Stieg jedoch

die Anzahl an Objekten in der Szene, reduzierte sich dieser Geschwindigkeitsvorteil von Sunshine gegenüber CPU basierter Renderer.
Der Vergleich zwischen der spektralen und RGB-basierten Variante von Sunshine zeigte deutlich den Nachteil des größeren Aufwandes vom spektralen Raytracing. Damit Farben korrekt dargestellt werden, müssen wesentlich mehr Pfade verfolgt werden, was sich direkt auf die Berechnungsdauer eines Bildes auswirkt.
Bei der Umsetzung des spektralen Raytracers wurde auch darauf geachtet, eine größere Bandbreite an GPUs zu unterstützen. Die ursprüngliche Form von Sunshine lief beispielsweise nur auf Grafikkarten der Firma NVIDIA. Für andere Grafikkarten, wie die der Firma ATI, musste Sunshine überarbeitet werden. Die monolithische Struktur wurde aufgebrochen und der Renderprozess in mehrere kleinere Kernels verteilt. Somit reduzierte sich die Ausführungszeit der einzelnen Kernels.
Ein Vorteil der Aufspaltung war nicht nur die Unterstützung von ATI Grafikkarten, sondern auch eine bessere Skalierbarkeit der Szenengröße. Da Programme auf der GPU nicht beliebig lang ausgeführt werden können, mussten diese unterbrechbar sein.
Diese Modifizierungen bedeuten jedoch auch gleichzeitig Leistungsverlust, da zwischendurch in Texturen geschrieben beziehungsweise von ihnen gelesen werden muss.

6.2 Ausblick

Der Renderer Sunshine wurde um das spektrale Path Tracing erweitert. Dies ermöglicht nun das Erzeugen von Effekten, die abhängig von Wellenlängen sind. Ein solcher Effekt wurde mit der Dispersion demonstriert.
Dennoch gibt es für Sunshine im Allgemeinen und speziell für das spektrale Raytracing Ansätze für weitere Überlegungen und Umsetzungen, die in diesen Abschnitt kurz besprochen werden sollen.
Generell wäre für Sunshine eine Beschleunigungsstruktur äußerst notwendig. Denn es hat sich gezeigt, dass der Vorteil durch das schnelle parallele Verarbeiten der GPU verloren geht, wenn jeder Strahl in jedem Schnittpunkttest mit allen Objekten getestet werden muss. Die CPU Lösungen haben hier einen großen Vorteil, wo solche Beschleunigungsstrukturen standardmäßig mit implementiert sind. Dennoch hat sich in einer parallel zu dieser Arbeit laufenden Masterarbeit gezeigt, dass eine Umsetzung sehr schwierig ist.
Ein weiterer Punkt wäre die Umsetzung geeigneter *Tonemapping*-Verfahren, um den bestmöglichen Kontrast aus einem erstellten Bild heraus zu holen. In diesem Zusammenhang sollte auch über das Einführen von weiteren Bildfiltern nachgedacht werden. Der Vergleich zwischen Sunshine und dem Lux-Renderer hat gezeigt, dass weitaus weniger Pfade pro Bildpunkt notwendig sind, um ein rauscharmes Bild zu erzeugen.
In der Literatur wird zum Erklären des Dispersionsbegriffes häufig ein Prisma gezeigt, der weißes Licht spektral zerlegt und einen Regenbogen auf eine Fläche projiziert. Diesen Effekt mit einem Path Tracer zu erzielen, ist eine schwierige Aufgabe. Denn der Strahl muss von der Kamera einen Weg von der Stelle des Regenbogens durch das Prisma zur Lichtquelle finden. Zum Finden solcher Pfade von der Kamera hin zur Lichtquelle sind andere Verfahren besser geeignet. Daher bietet sich die Umsetzung eines spektralen bidi-

rektionalen Path Tracers an, der solche komplizierten Lichtsituationen besser handhaben kann.
Ebenso wichtig für das Erlangen eines hohen Grades an Realismus bei der Bildsynthese sind Kameramodelle. Sie simulieren Objektive, die den Einfall der Lichtstrahlen auf die Bildebene beeinflussen. Damit sind Effekte wie Tiefenschärfe darstellbar. Im Zusammenhang mit spektralem Raytracing könnten hier sogar Messdaten von echten Objektiven genutzt werden. Somit könnte man das Verhalten dieser Objektive am PC simulieren und nutzt dabei die Schnelligkeit der GPU.
Ein Teil des *Sunsky*-Modells[19] ist ebenfalls umgesetzt worden. Jedoch fehlt die Behandlung der Streuung von Lichtstrahlen in der Atmosphäre. Mit einem kompletten Atmosphärenmodell und des spektralen Renderers könnten dann „echte“ Regenbögen im Himmel dargestellt werden.
Nicht nur die Dispersion ist ein wellenlängenabhängiger Effekt. Auch Interferenzen an dünnen Schichten sind mit spektralem Raytracing darstellbar. Dieser Effekt ist bei Seifenblasen oder Ölfilmen zu beobachten.
Interferenzen an Kanten von Objekten führen zur Beugung von Lichtstrahlen und es entstehen Muster, wie beispielsweise konzentrische Kreise. Diesen Effekt bezeichnet man auch als Diffraktion. Damit dieser Effekt erzeugt werden kann, muss neben der Wellenlänge auch die Phase und Polarisation der Lichtstrahlen berücksichtigt werden. Dies bedeutet jedoch auch zusätzlichen Berechnungsaufwand.
Es bleibt festzuhalten, dass das hier umgesetzte Konzept eines spektralen Path Tracers nur ein Ausgangspunkt für Diskussionen, Überlegungen und Umsetzungen weiterer Verfahren ist. Die Herausforderungen werden darin bestehen, existierende Konzepte von der CPU auf die GPU zu transferieren und dabei möglichst wenig Kompromisse eingehen zu müssen, die den Geschwindigkeitsvorteil mindern könnten.



Literaturverzeichnis

[1] Ashikhmin, Michael ; Shirley, Peter: An anisotropic Phong BRDF model. In: *J. Graph. Tools* 5 (2000), Nr. 2, S. 25–32. – ISSN 1086–7651

[2] Carr, Nathan A. ; Hoberock, Jared ; Crane, Keenan ; Hart, John C.: Fast GPU ray tracing of dynamic meshes using geometry images. In: *GI '06: Proceedings of Graphics Interface 2006*. Toronto, Ont., Canada, Canada : Canadian Information Processing Society, 2006. – ISBN 1–56881–308–2, S. 203–209

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

[4] Cook, Robert L. ; Porter, Thomas ; Carpenter, Loren: Distributed ray tracing. In: *SIGGRAPH Comput. Graph.* 18 (1984), Nr. 3, S. 137–145. http://dx.doi.org/http://doi.acm.org/10.1145/964965.808590. – DOI http://doi.acm.org/10.1145/964965.808590. – ISSN 0097–8930

[5] Durikovic, Roman ; Kimura, Ryou: GPU Rendering of the Thin Film on Paints with Full Spectrum. In: *IV '06: Proceedings of the conference on Information Visualization*, IEEE Computer Society, 2006, S. 751–756

[6] Foley, James D. ; Dam, Andries van ; Feiner, Steven K. ; Hughes, John F.: *Computer Graphics: Principles and Practice in C*. 2nd edition. Addison-Wesley Professional, 1995. – ISBN 0201848406

[7] Göddeke, Dominik: GPGPU–Basic Math Tutorial / FB Mathematik, Universität Dortmund. 2005. – Forschungsbericht. – Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300, http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu

[8] Huwe, Thomas: *Stochastischer GPU-Strahlenverfolger für GroIMP*. Brandenburgische Technische Unifersität Cottbus, 2008. – Bachelorarbeit

[9] Kajiya, James T.: The rendering equation. In: *SIGGRAPH Comput. Graph.* 20 (1986), Nr. 4, S. 143–150. http://dx.doi.org/http://doi.acm.org/10.1145/15886.15902. – DOI http://doi.acm.org/10.1145/15886.15902. – ISSN 0097–8930

[10] Kilgariff, Emmett ; Fernando, Randima: *The GeForce 6 series GPU architecture*. http://dx.doi.org/http://dx.doi.org/10.1145/1198555.1198767. Version: 2005

[11] Kniemeyer, Ole: *Design and implementation of a graph grammar based language for functional-structural plant modelling.* 03013 Cottbus, Brandenburg, Germany, Brandenburgische Technische Universität Cottbus, Diss., 2008

[12] Kurzweil, Peter: *Das Vieweg Formel-Lexikon: Basiswissen für Ingenieure, Naturwissenschaftler und Mediziner.* Vieweg+Teubner Verlag, 2002. – 594 S.

[13] NVIDIA: *NVIDIA CUDA Programming Guide 2.0.* 2008

[14] Oren, Michael ; Nayar, Shree K.: Generalization of Lambert's reflectance model. In: *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques.* New York, NY, USA : ACM, 1994. – ISBN 0–89791–667–0, S. 239–246

[15] Owens, John D. ; Luebke, David ; Govindaraju, Naga ; Harris, Mark ; Krüger, Jens ; Lefohn, Aaron E. ; Purcell, Timothy J.: A Survey of General-Purpose Computation on Graphics Hardware. In: *Computer Graphics Forum* 26 (2007), Nr. 1, 80–113. `http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x`

[16] Park, S. K. ; Miller, K. W.: Random number generators: good ones are hard to find. In: *Commun. ACM* 31 (1988), Nr. 10, S. 1192–1201. `http://dx.doi.org/http://doi.acm.org/10.1145/63039.63042`. – DOI http://doi.acm.org/10.1145/63039.63042. – ISSN 0001–0782

[17] Peres, Michael R.: *The Focal Encyclopedia of Photography: Digital Imaging, Theory and Applications, History, and Science.* Bd. 846. 4, illustrated, revised edition. Focal Press„ 2007. – ISBN: 0240807405, 9780240807409

[18] Pharr, Matt ; Humphreys, Greg: *Physically Based Rendering: From Theory to Implementation.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2004. – ISBN 012553180X

[19] Preetham, A. J. ; Shirley, Peter ; Smits, Brian: A practical analytic model for daylight. In: *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques.* New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1999. – ISBN 0–201–48560–5, S. 91–100

[20] Purcell, Timothy J.: *Ray tracing on a stream processor.* Stanford, CA, USA, Diss., 2004. – Adviser-Hanrahan, Patrick M.

[21] Raj, M.G. ; M.G, Raj: *Fundamentals of Optics.* Anmol Publications PVT. LTD., 2002. – 436 S.

[22] Rost, Randi J.: *OpenGL(R) Shading Language (2nd Edition).* Addison-Wesley Professional, 2005. – ISBN 0321334892

[23] Schiffner, Daniel: *Wellenlängenbasiertes Beleuchtungsmodell im Shader*, Goethe Universität Frankfurt, Diplomarbeit, 2008

[24] Schmittler, Jörg ; Pohl, Daniel ; Dahmen, Tim ; Vogelgesang, Christian ; Slusallek, Philipp: Realtime ray tracing for current and future games. In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA : ACM, 2005, S. 23

[25] Shreiner, Dave: *OpenGL(R) 1.4 Reference Manual (4th Edition)*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2004. – ISBN 032117383X

[26] Smits, Brian: An RGB-to-spectrum conversion for reflectances. In: *J. Graph. Tools* 4 (1999), Nr. 4, S. 11–22. – ISSN 1086–7651

[27] Sun, Y. ; Fracchia, F. D. ; Drew, M. S.: Rendering light dispersion with a composite spectral model. In: *International Conference on Color in Graphics and Image Processing - CGIP'2000*, 2000

[28] Sun, Yinlong ; Fracchia, F. D. ; Drew, Mark S. ; Calvert, Thomas W.: A spectrally based framework for realistic image synthesis. In: *The Visual Computer* 17 (2001), October, Nr. 7, S. 429–444

[29] Torrance, K. E. ; Sparrow, E. M.: Theory for Off-Specular Reflection From Roughened Surfaces. In: *J. Opt. Soc. Am.* 57 (1967), Nr. 9, S. 1105–1112

[30] Veach, Eric: *Robust Monte Carlo methods for light transport simulation*. Stanford, CA, USA, Diss., 1998. – Adviser-Guibas, Leonidas J.

[31] Walter, Bruce ; Marschner, Stephen R. ; Li, Hongsong ; Torrance, Kenneth E.: Microfacet Models for Refraction through Rough Surfaces. In: *Rendering Techniques 2007: 18th Eurographics Workshop on Rendering*, 2007, S. 195–206

[32] Whitted, Turner: An improved illumination model for shaded display. In: *Commun. ACM* 23 (1980), Nr. 6, S. 343–349. `http://dx.doi.org/http://doi.acm.org/10.1145/358876.358882`. – DOI http://doi.acm.org/10.1145/358876.358882. – ISSN 0001–0782

[33] Wright, Richard S. ; Lipchak, Benjamin: *OpenGL SuperBible (3rd Edition)*. Indianapolis, IN, USA : Sams, 2004. – ISBN 0672326019