Goethe Universität Frankfurt

Diplomarbeit

Wellenlängenbasiertes Beleuchtungsmodell im Shader

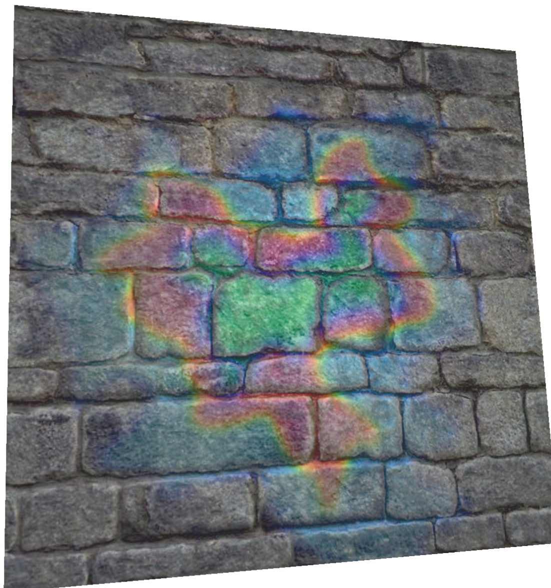
eingereicht bei

Prof. Dr.-Ing. Detlef Krömker,

Professur für Graphische Datenverarbeitung

von

Daniel Schiffner



Eingereicht am 9. Oktober 2008

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Frankfurt am Main, den 9. Oktober 2008

(Daniel Schiffner)

Zusammenfassung

In der Realität setzen sich Farben aus einzelnen Wellen zusammen, welche in Kombination mit zugehörigen Wellenlängen und Intensitäten bei Menschen den Sinneseindruck einer Farbe hervorrufen. Die Computergraphik definiert Farben mit dem RGB-Modell, in dem durch 3 Grundfarben (Rot, Grün, Blau) der darstellbare Farbbereich festgelegt wird. Ein Spektrum (genauer *Spectral Power Distribution*, SPD) ermöglicht eine variablere, physikalisch exaktere Darstellung von Farbe, kann aber nicht einfach mit dem RGB-Modell verwendet werden. Das von der *Commission Internationale de l'Éclairage* definierte XYZ-Farbmodell erlaubt es mit Wellenlängen zu rechnen, und bildet die Grundlage der Beleuchtungsrechnung mit Spektren.

Farben mittels Spektren zu ermitteln ist die Paradedisziplin von Raytracern, da der Berechnungsaufwand für Echtzeitanwendungen meist zu groß ist. Die neueste Graphikkarten-Generation kann große Datenmengen effizient parallel verarbeiten, und es wurden entsprechende Ansätze gesucht, wellenlängenbasiert zu rechnen. Das hier vorgestellte System erlaubt auf Grundlage von physikalischen Formeln einzelne Intensitäten zu beeinflussen, welche in Kombination mit den Tristimulus-Werten des Menschen in dem XYZ-Farbmodell abgebildet werden können. Diese XYZ-Koordinaten können anschließend in das RGB-Modell transformiert werden.

Im Gegensatz zu bestehenden Systemen wird direkt mit Spektren gearbeitet und diese nicht von einer RGB-Farbe abgeleitet, so dass für bestimmte Effekte eine höhere Genauigkeit entsteht. Durch die Verwendung einer SPD ist es möglich, Interferenzeffekte an dünnen Schichten und CDs in einem Polygon-Renderer zu visualisieren. In dieser Arbeit wird eine Berechnung von mehrlagigen dünnen Schichten mit komplexen Brechungsindizes präsentiert und ein LOD-System vorgestellt, welches es ermöglicht den Berechnungsaufwand frei zu skalieren.

Abstract

In reality, color is the sum of individual waves which lead to a sensation of color for humans. Waves are defined by their wavelength and the corresponding intensity. Computergraphics are based on the RGB-model, constructed via the 3 base colors red, green and blue, to specify a color. A Spectrum (more precisely a *Spectral Power Distribution*, short SPD) allows a more variable and physically based approach, but cannot easily be implemented in the RGB-model. Based on the XYZ-model from the *Commission Internationale de l'Éclairage*, a calculation with single wavelenghts is possible, and is the main system for all wavelength based illumination models.

Calculating colors via a SPD is mainly implemented with ray tracing renderer, because the costs are basically to high for real time applications. But the newest generation of graphic cards is able to process efficiently and parallel a high amount of data, so it seems possible to implement a wavelength based model on a polygon renderer. The developed system can manipulate single intensities based on physical formulas and can calculate the

corresponding color in XYZ-coordinates using the human tristimulus values. These values can be transformed into the RGB-model.

Unlike other systems, the SPD is not derived via a transformation of an RGB-color defined in the renderer. Using individual SPDs for calculation should allow a higher precision. This approach also allows to model interference effects of thin films or a CD on a polygon renderer. A formula for modeling a multi-layer system with complex indices of refraction and an LOD system is presented. The latter can decrase the amount of time spent on calculation at the cost of accuracy.

Danksagung

Ich danke allen, die mich während meines Studiums unterstützt und gefördert haben.

Vorallem danke ich meinen Eltern, die es möglich gemacht haben, meine Ziele und das Studium zu realisieren. Ich konnte mich in jedem Bezug auf ihre Unterstützung verlassen.

Auch den Korrekturlesern dieser Arbeit möchte ich besonderen Dank zukommen lassen, die diese Arbeit zu dem gemacht haben, was sie jetzt ist: Johannes, Ingo, Sven, Christian, Carsten.

Zuletzt möchte ich meinem Betreuer Sebastian Schäfer für seine tatkräftige Unterstützung und Hilfe danken, sowie Prof. Dr.-Ing. Detlef Krömker, der diese Arbeit ermöglicht hat.

Inhaltsverzeichnis

Zusammenfassung **ii**

1 Einleitung **2**

2 Grundlagen **4**

2.1 Farbmodelle 4

2.1.1 RGB-Modell 4

2.1.2 XYZ-Modell 6

2.1.3 Andere Modelle 10

2.2 Physikalische Grundlagen 11

2.2.1 Licht - Welle und Teilchen 11

2.2.2 Brechung von Licht 13

2.2.3 Interferenz 15

2.2.4 Diffraktion 17

2.2.5 Fresnel - Gleichungen 19

2.3 GPU und Shader 20

2.3.1 Renderpipeline 21

2.3.2 Entwicklung der Shader 23

3 State of the Art **29**

3.1 Beleuchtung 29

3.1.1 Lambert-Beleuchtungsmodell 30

3.1.2 Phong-Beleuchtungsmodell 30

3.1.3 Blinn-Beleuchtungsmodell 31

3.1.4 BRDF 33

3.2 Interferenz in der Graphik 33

3.2.1 Raytracer 34

3.2.2 Polygon-Renderer 35

4 Eigene Verfahren **42**

4.1 Shader Viewer 42

4.2 Konzeption des Beleuchtungsmodells 48

4.3 Implementation des Grundmodells 51

4.4 Implementation von Interferenzeffekten 60

4.5 Modellierung der Diffraktion an einer CD 74

4.6 Level Of Detail System 77

5 Aussichten und Diskussion **79**
5.1 Texturdefinition . . . 79
5.2 Weitere Ergänzungsmöglichkeiten . . . 82

A Herleitung des komplexen Reflexions- und Transmissionsfaktors **83**

B Spektrenverzeichnis **87**

C Literatur **89**

Quellcodes

1 CG Beispiel . . . 26
2 CG Techniques . . . 28
3 Pseudo Code für die grundlegende Wellenlängen Implementation . . . 51
4 Funktion zum Auslesen der CMF Werte . . . 52
5 Berechnung der XYZ-Koordinaten in festen Grenzen . . . 53
6 Fragment Shader zur einfachen wellenlängenbasierten Beleuchtungsrechnung . 53
7 Bestimmung der Intensität in einem Pixel nach Phong . . . 54
8 Die benutzten Funktionen für die Beleuchtung . . . 55
9 Auslesen einzelner Intensitäten aus einer Textur . . . 57
10 Kombiniertes Auslesen der Intensitäten aus einer Textur . . . 58
11 Beleuchtungsrechnung mit individuellen Intensitäten . . . 59
12 Vektorbasierte Berechnung des Brechungswinkels . . . 62
13 Einfache Berechnung von Interferenz . . . 63
14 Funktion zur Bestimmung des Reflexionsfaktors für beide Polarisierungen . . 64
15 Berechnung der Interferenz mit Fresnel . . . 66

Abbildungsverzeichnis

1 Einheitswürfel des RGB-Farbmodells . . . 4
2 CIE RGB Color Matching Functions . . . 6
3 CIE XYZ Color Matching Functions . . . 7
4 Visualisierung des Snellius'schen Brechungsgesetz . . . 14
5 Totalreflexion an optischen Materialien . . . 15
6 Beispiel von Interferenz . . . 16
7 Geometrie von Doppel- und Einzelspalt . . . 18

8 Amplitudenverhältnisse nach Fresnel-Gleichungen 19
9 Verschiedene standardmäßig verfügbare Polygone 21
10 Beleuchtung auf Vertex- und Pixel-Ebene 23
11 Die Renderpipeline 24
12 Vergleich zwischen Blinn- und Phong-Vektoren 32
13 Ergebnis des dünnen Schichten Effektes von nVidia 35
14 Regenbogen Textur 36
15 Ergebnisse der Diffraktion 37
16 Zylinder mit dünner Schicht 39
17 Standard-Ansicht des ShaderViewers 45
18 Aufteilungsvarianten für die Erstellung von Cubemaps 47
19 Speicherung der Spektrumdaten in einer Textur 48
20 In einer Textur gespeicherte CMF-Werte 50
21 Verschiedene Beleuchtungen des selben Objektes in dem Wellenlängenmodell 54
22 Auswirkungen auf die Farbwiedergabe mit dem Parameter **steps** 56
23 Vergleich zwischen Phong Beleuchtung im wellenlängenbasierten und im RGB-Modell 57
24 Beispiele von genutzten Spektren 58
25 Beleuchtung mit variablen Intensitäten 61
26 Interferenz mit der Weglängenmethode 64
27 Ergebnisse der Methode von Durikovic 65
28 Interferenz nach alternativer Berechnung 67
29 Interferenzmodelle nach Sun 68
30 Interferenz nach Fabry-Pérot 69
32 Interferenz in Kombination mit Bumpmapping 71
31 MultiLayer mit komplexen Brechungsindizes 72
33 Interferenz auf einer Wasseroberfläche 73
34 CD mit eigenem Shader 74
35 Strahlengang am Reflexionsgitter 75
36 CD mit Substratschicht 76
37 Ölschicht auf einer Wasserpfütze 80
38 Spektren 0-3 87
39 Spektren 4-7 88

Tabellenverzeichnis

1 Grundtypen von Sampler in CG 27
2 Vergleich der Frameraten durch Genauigkeit der CMF-Werte 50
3 Frameraten in FPS für die einfachen Modelle 55
4 Frameraten der Modelle mit variablen Intensitäten 60
5 Ergebnisse der einfachen dünnen Film-Modelle 66
6 Ergebnisse der erweiterten dünnen Film- Modelle 67
7 Ergebnisse des MultiLayer-Systems mit variabler Schichtzahl 71
8 Ergebnisse der einfachen dünnen Film-Modelle 73
9 Ergebnisse des LOD-Systems 78

1 Einleitung

In der Computergraphik war es von Beginn an eines der großen Ziele, die Realität im besten Maße abzubilden. Der große Anspruch bestand darin, die Farben aus der Natur auf einem Bildschirm oder Papier wiederzugeben. Auf Grund der Beschränkungen der Rechenkapazität der Computer ist es bis heute nicht möglich, beliebig genaue Modelle zu nutzen, so dass es zu Einschränkungen in der Qualität der Wiedergabe führt. Die Problematik der Farbwiedergabe konnte mit dem RGB-Modell gut behoben werden, da es an das Farbempfinden eines Menschen angelehnt ist. Dieses Farbmodell bildet bis heute die Grundlage der meisten Wiedergabegeräte, über Fernseher und Beamer bis hin zum Monitor. Jedoch ist es nicht ohne weiteres möglich auf Basis des gegebenen RGB-Farbmodell mit den drei Grundfarben (Rot, Grün und Blau) alle sichtbaren Effekte wiederzugeben.

Die Graphikhardware hat sich bis heute aber so stark in der Leistung gesteigert, dass eine genauere Darstellung mit den vorhandenen Mitteln realisierbar scheint, wie sie von den Raytracern, die eine andere Grundlage verwenden, schon heute durchgeführt wird. Somit liegt der Gedanke nicht fern, ein Modell zu entwickeln, mit dem es möglich sein sollte, Farben in Abhängigkeit von einem Wellenlängenspektrum zu bestimmen, das jedoch auf einem Polygon-Renderer genutzt werden kann. Diese Herangehensweise nutzt eine physikalische und nicht eine humane Grundlage, da sich Farben bzw. das Farbempfinden in der Natur aus einzelnen Wellenlängen zusammensetzt. Bei Menschen basiert die Farbwahrnehmung auf drei Rezeptoren, hingegen besitzen manche Tiere mehr Rezeptoren, wie z.B. Vögel, welche vier Grundfarben unterscheiden können [Bre07].

Ein Spektrum ist als eine Menge einzelner Wellenlängen definiert, denen jeweils eine Intensität zugeordnet ist. Menschen sind in der Lage Wellenlängen in dem Bereich von $\lambda = [360\text{nm}, 830\text{nm}]$ zu erkennen. Dieser wird daher als sichtbarer Bereich bezeichnet. Wellenlängen die kleiner sind ($\lambda < 360\text{nm}$) bezeichnet man als Ultraviolett-Strahlung, Wellenlängen die größer sind ($\lambda > 830\text{nm}$) nennt man Infrarot-Strahlung.

Ziel dieser Arbeit ist es ein Modell zu entwickeln, mit dem die Darstellungen von visuellen Effekten, wie z.B. *Interferenz*, *Dispersion* oder *Hologramme* am Computer nicht nur berechnet, sondern auch graphisch in Echtzeit ausgegeben werden können. Die bisherigen Beleuchtungsmodelle, wie z.B. Phong oder Blinn, sollen implementierbar bleiben, so dass dieses Modell als eine Erweiterung zu den bestehenden Beleuchtungssystemen verstanden werden kann.

Schon heute können Raytracer solche Berechnungen durchführen. Da aber diese ausschließlich auf der CPU[1] ausgeführt werden, fehlt es an der nötigen Performanz, oder es müssten

[1] Central Processing Unit

starke Einschränkungen an dem physikalischen Modell gemacht werden, um sie für Realtime-Anwendungen zu verwenden.

Für die Berechnung dieses Modells wird auf die spezialisierte Graphik-Hardware der aktuellen Graphikkarten Generation zurückgegriffen, die große Geschwindigkeitsvorteile gegenüber der normalen CPU im Bezug auf die Bilddarstellung bietet. Die Graphik-Hardware verfügt über eine GPU[2], die vektorbasiert alle Berechnungen durchführt, die nötig sind, um eine Szene (2D oder 3D) auf ein 2D-Ausgabegerät zu projezieren. Hierbei spricht man von der Renderpipeline. Die GPU ist dabei auf die Graphikausgabe spezialisiert, und kann viele Berechnungen gleichzeitig (parallel) ausführen. Denn bei der Ausgabe ändert sich die anzuzeigende Szene nicht mehr, wodurch eine deutliche Beschleunigung erzielt werden kann.

Die aktuellen Graphik APIs[3] bieten desweiteren eine Schnittstelle zu den sogenannten Shadern der Graphikkarten an. Diese sind frei programmierbar, und erlauben es - im Rahmen des Systems - beliebige Berechnungen auf der GPU durchzuführen. Für die Implementierung werden zur Zeit 3 Hochsprachen angeboten (HLSL, GLSL und CG), die alle in entsprechenden Assembler Code umgewandelt werden, so dass er von der GPU verarbeitet werden kann. Für die Implementierung wird das von nVidia angebotene CG benutzt, da es sowohl mit der Graphik API von Microsoft (Direct3D) und OpenGL verwendet werden kann, und daher eine universelle Modellierung erlaubt.

Im folgenden Kapitel 2 werden die Grundlagen erklärt, wozu die Farbsysteme, die zugrunde liegenden physikalischen Effekte und die Programmierung mit Shadern gehören. Ein Vergleich zu den aktuellen Modellen (Raytracer, andere Polygon-basierte Modelle) und die Erläuterung des eigenen Modells folgen in den Kapiteln 3 und 4. Zum Abschluss wird es einen Ausblick auf Verbesserungen und Erweiterungen geben (Kapitel 5).

[2]Graphics Processing Unit

[3]Sowohl DirectX als auch OpenGL

2 Grundlagen

Um zu verstehen, wie das entwickelte Modell genutzt wird, ist ein gewisses Verständnis über Farbsysteme und physikalische Grundlagen von Nöten, die im Folgenden erläutert werden. Am Ende des Kapitels wird auf die Struktur einer aktuellen Graphikkarte eingegangen und erläutert, was Shader sind, und wie sie genutzt werden können, um eigene Berechnungen durchzuführen.

2.1 Farbmodelle

Farbmodelle wurden schon lange vor der Computergraphik intensiv untersucht, und es wurde versucht ein möglichst allgemeines System zu finden. Schon Grassmann stellte 1853 fest, dass zwischen je 4 Farben eine eindeutige lineare Beziehung herrscht und das eine Farbe drei unabhängige Größen (Primärvalenzen) braucht, um eindeutig beschrieben zu werden. Primärvalenzen sind vom CIE ermittelte Farben, sie entsprechen den Grundfarben Rot, Grün und Blau. Daher folgerte er, dass Farbe eine 3-dimensionale Größe ist [Bre07]. Die Darstellung in einem Vektorraum ermöglicht es, die Rechengesetze von Vektoren zu nutzen, und Farben von einem Modell in ein anderes zu konvertieren. Somit ist das RGB-Modell ein grundlegendes System, jedoch gibt es unendlich viele andere Darstellungsformen für Farben.

2.1.1 RGB-Modell

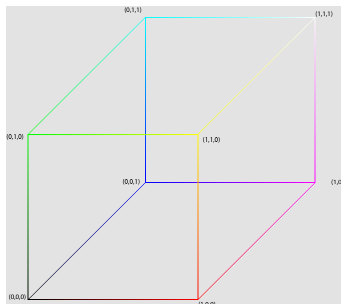


Abbildung 1: Einheitswürfel des RGB-Farbmodells

Nach dem RGB-Farbmodell können alle darstellbaren Farben in einem Einheitswürfel abgebildet werden. So befinden sich die Primärvalenzen auf den positiven Achsen des 3-dimensionalen Raums. Die einzelnen RGB-Farben setzen sich aus der additiven Farbmischung der Primärvalenzen zusammen. Da beim Menschen je ein Rezeptor (Zapfen) im Auge auf eine der Farben Rot,Grün und Blau anspricht, bildet das RGB-Modell eine naheliegende Grundlage.

Die Farbmischung kann dabei auf verschiedene Arten erfolgen. Die Farben können *a*) gemischt zum Auge gelangen, d.h. dass ein kombiniertes Spektrum am Auge auftritt und ein entsprechendes Farbempfinden auslöst, *b*) zeitlich verschoben beim Auge auftreffen, so dass sich bei entsprechend kleiner Verzögerung auf Grund der Wahrnehmung eine einzelne Farbe herausbildet, was z.B. bei einem Farbkreisel verwendet wird, oder *c*) auf Grund der räumlichen Verteilung ein einheitliches Farbempfinden auslösen. So wird letzteres bei der Darstellung im Monitor ausgenutzt, oder auch bei dem sogenannten *Dithering*, wo man eine größere Farbtiefe durch das Einstreuen von anderen Farben ermöglicht und somit harte Farbübergänge verhindern kann.

Als Hintergrundfarbe ist Schwarz definiert, und entspricht der Farbe, die man wahrnimmt, wenn kein Licht auf ein Objekt trifft. Diese befindet sich bei dem Einheitswürfel im Ursprung ($\vec{b} = (0, 0, 0)$). Weiß setzt sich aus der Summe der Primärvalenzen mit voller Intensität zusammen, und befindet sich somit auf der Diagonalen des Würfels am Punkt $\vec{w} = (1, 1, 1)$. Die Farben auf der Diagonalen selbst besitzen identische Werte für Rot, Grün und Blau, und sind als Grauwerte definiert. Eine Abbildung des Würfels ist in Abbildung 1 dargestellt.

Da es sich bei dem RGB-Modell um eine allgemeine Beschreibung handelt, kann nicht gesagt werden, um welche reale Farben es sich bei den einzelnen Primärvalenzen handelt. So ist die Farbwiedergabe bei den Monitoren von den einzelnen verwendeten Primärvalenzen abhängig, und die Ausgabe kann bei jedem Bildschirm variieren. Auch bietet das Modell keine intuitive Steuerung der Farbwerte, da die Wahrnehmung der Farben bei Menschen nicht linear ist, wie es von dem RGB-Modell vorgegeben wird [Bre07].

Die Vorzüge des RGB-Modells liegen in der einfachen Einbindung in die Hardware. Die einzelnen Farben (Rot, Grün und Blau) können durch bestimmte Phosphore oder Farbfilter einfach hergestellt werden und es existiert auch keine explizite Vorgabe für die einzelnen Primärvalenzen. Eine Unterteilung in 256 Schritte liefert eine gute Approximation und die Werte für die Farben plus einen zusätzlichen transparenten Wert können in 4 **Byte** (DWORD) gespeichert werden. Die Genauigkeit kann auch mit Speicherung als **Float** erhöht werden, was jedoch die Effizienz verringert, da eine Farbe mit **Float**-Werten aus 32 **Byte**[4] besteht, und somit die 8-fache Speichermenge benötigt wird.

Auch wenn die Ausgabe am Ende einer Beleuchtungsrechnung auf einen **Byte**-Wert geclippt werden sollte, so kann trotzdem in der Pipeline die Farbe mit einer höheren Genauigkeit verwendet werden.

[4] üblicherweise wird RGBA abgespeichert

2.1.2 XYZ-Modell

Das Ziel mit dem XYZ-Modell war es, ein Modell zu entwickeln, welches dem Farbempfinden des Menschen entspricht. In der sogenannten Colorimetrie wurde versucht, Farben durch wissenschaftliche Messungen zu beschreiben. Die CIE[5] hatte die ersten nennenswerten Ergebnisse geliefert, und eigene entsprechende Standards verabschiedet.

Hierfür wurde eine Versuchsreihe gestartet, die einen sogenannten Normalbetrachter beschreibt, der unter einem bestimmten Sehwinkel einzelne Farben wahrnimmt. Dabei wurden die Versuche in dem Standard $CIE1931$ mit einem Winkel von 2° durchgeführt, bei dem Standard von $CIE1961$ 10°. Die Aufgabe für die Probanden bestand in dem Versuch darin, mit 3 monochromatischen Lichtquellen eine auf einem Schirm vorgegebene Farbe zu mischen. Dabei hatten sie die Möglichkeit, die Intensitäten der einzelnen Lichtquellen per Regler einzustellen. Leider stellte man fest, dass nicht alle Farben auf diese Art und Weise herstellbar waren, dennoch wurden die einzelnen Farben in ein Modell eingebaut. Die definierten Farben entsprechen dabei [Bre07]:

$$\text{CIE Rot} := \text{Wellenlänge von } 700\text{nm}$$
$$\text{CIE Grün} := \text{Wellenlänge von } 546,1\text{nm}$$
$$\text{CIE Blau} := \text{Wellenlänge von } 435,8\text{nm}$$

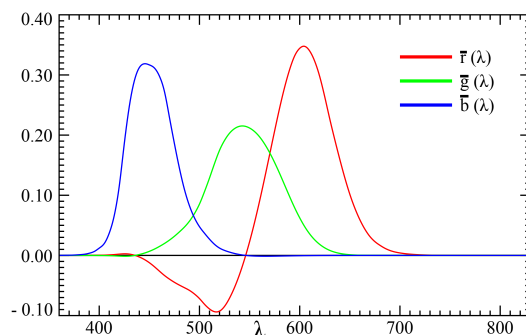


Abbildung 2: CIE RGB Color Matching Functions.
Quelle Wikipedia

Auch wenn nicht alle Farben darstellbar waren, konnte man dennoch mit den gewonnenen Daten eine statistische Herleitung für den sogenannten Normalbetrachter machen. Man erhielt die in Abbildung 2 gezeigten Kurven für die einzelnen Intensitäten der Lichtquellen, welche *Color Matching Functions* (CMF) genannt werden. Die Kurve nimmt für die Intensität der roten Lichtquelle im Bereich von ca. [450nm − 550nm] negative Werte an. Somit kann erklärt werden, warum nicht alle Farben mit den 3 Lichtquellen darstellbar sind. Des Weiteren wurde anstelle von spezifischen Intensitäten der einzelnen Primärvalenzen eine Gleichvertei-

[5] Commission International de l'Eclaireage

lung hergestellt, indem die Flächen unter den einzelnen Kurven auf gleiche Größe gebracht wurden. Die einzelnen Funktionen werden mit $\bar{r}, \bar{g}$ und $\bar{b}$ bezeichnet und beschreiben die einzelnen Intensitäten der entsprechenden Farben.

$$\int_0^\infty \bar{r}(\lambda)d\lambda = \int_0^\infty \bar{g}(\lambda)d\lambda = \int_0^\infty \bar{b}(\lambda)d\lambda$$

Zu beachten ist, dass es sich hier auch um ein RGB-Modell handelt. Ziel des CIE war es, ein Modell zu entwickeln, welches noch weitere spezielle Eigenschaften gegenüber dem RGB hat. Die einzelnen Funktionen der CMF aus dem XYZ-Farbmodell lauten danach $\bar{x}, \bar{y}$ und $\bar{z}$.

Es gibt eine lineare Umrechnung zwischen den beiden Modellen, die auf der Basis der Grassmann'schen Gesetze beruhen, da hier auch nur 3 Primärvalenzen benutzt werden. Die einzelnen Werte der CMF sollten nicht negativ sein, was zur Zeit der Definition des Modells (1931) vorteilhaft war, da es die Farbberechnung erheblich vereinfacht. Der Wert von $\bar{y}$ sollte der Hellempfindlichkeitskurve entsprechen, welches den spektralen Empfindlichkeitsgrad wiedergibt, den Testpersonen bei Tageslicht haben. Dieser wurde vom CIE bereits 1924 veröffentlicht [Ohn99]. Der Weißpunkt sollte in dem Modell dadurch zustande kommen, dass alle Werte in der Summe gleich 1 sind, also $\bar{x} = \bar{y} = \bar{z} = \frac{1}{3}$.

Die Umsetzung dieses Modells führte zu einer CMF, welche in Abbildung 3 dargestellt ist. Das XYZ-Farbmodell bildete die Grundlage für viele weitere Modelle, da es auf direkten Messungen der menschlichen Farbwahrnehmung beruht, und versucht, die einzelnen Rezeptoren des Auges nachzuempfinden. Das menschliche Auge besteht aus 3 verschiedenen Arten von Rezeptoren für Farben (den Zapfen): S,M und L-Rezeptoren. Dabei kommen die Namen aus dem Englischen, S steht für *short wavelenght receptor*, M für *medium wavelength receptor* und L für *long wavelength receptor*, und bezeichnen das Ansprechverhalten der einzelnen Rezeptoren. Man definiert daher auch die sogenannten Tristimulus-Werte, welche die einzelnen Einflüsse der Primärvalenzen widerspiegeln. So gibt es von jedem Farbmodell eine Methode, welche die Tristimulus-Werte bestimmt und somit zu einem resultierenden Farbempfinden führt. In dem XYZ-Farbmodell entsprechen XYZ den Tristimulus-Werten.

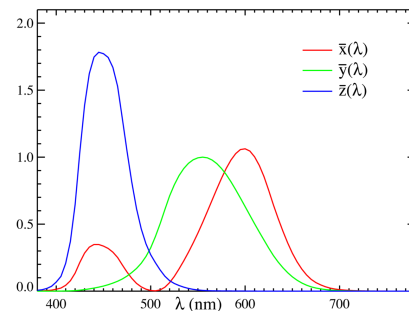


Abbildung 3: CIE XYZ Color Matching Functions. Quelle Wikipedia

Nachteil des Modells war bei der ersten Wertebestimmung, dass es nur eine sehr kleine Anzahl von Probanden gab, die zu den empirisch ermittelten CMFs geführt haben. Somit konnte erst durch spätere Korrekturen eine einwandfreie Definition der Werte sichergestellt

werden. In dieser Arbeit werden die Werte von 1964 verwendet [SS06].

Zur Berechnung der finalen XYZ-Werte muss das Integral der einzelnen CMFs gebildet werden. Verrechnet man also die CMF mit den einzelnen Intensitäten I einer *Spectral Power Distribution*, so erhält man die Tristimulus-Werte für die entsprechende resultierende Farbe im XYZ-Farbraum:

$$X = \int_{\lambda=0}^{\infty} I(\lambda)\bar{x}(\lambda)d\lambda \quad (1)$$

$$Y = \int_{\lambda=0}^{\infty} I(\lambda)\bar{y}(\lambda)d\lambda \quad (2)$$

$$Z = \int_{\lambda=0}^{\infty} I(\lambda)\bar{z}(\lambda)d\lambda \quad (3)$$

Die spezielle Konstruktion des XYZ-Farbraum ermöglicht es weiterhin, folgende Gleichungen aufzustellen, die die Farbe (engl.: *chromaticity*) ergeben:

$$x = \frac{X}{X+Y+Z} \quad (4)$$

$$y = \frac{Y}{X+Y+Z} \quad (5)$$

$$z = \frac{Z}{X+Y+Z} = 1 - x - y \quad (6)$$

Mit x, y und Y kann dann ein Farbraum definiert werden, da von diesen Werten ausgehend, die Tristimulus-Werte bestimmt werden können. Dadurch ist es möglich, den Farbraum abzubilden, was in der sogenannten CIE-Normfarbtafel (engl. *horse-shoe-* oder *chromaticity-diagram*) resultiert. Diese Tafel besitzt eine Menge von günstigen Eigenschaften. Es ist aber zu beachten, dass es sich nur um eine Abbildung von einem 3-dimensionalen Objekt auf eine 2-dimensionale Ebene handelt. Sie bildet das komplette Farbsprektrum ab, das ein durchschnittlicher Mensch sehen kann. Alle Farben, die auf einer Linie in der Tafel liegen, können durch beliebiges Mischen der beiden Endpunkte hergestellt werden. Auch ist erkennbar, dass die Tafel nicht durch 3 Punkte erstellt werden kann.

Der sichtbare Bereich wird Gamut genannt, und beschreibt, welche Farben durch ein Gerät dargestellt werden können. Die komplette Tafel entspricht dem Gamut der menschlichen Perzeption. Benutzt man 3 Primärvalenzen, so stecken sie einen Unterfarbraum ab, der durch beliebige Mischung dieser Primärvalenzen entstehen kann. Es ist evident, dass die Farbwiedergabe in direkter Abhängigkeit zum Gamut steht. Auch ist die Umrechnung vom XYZ- in den RGB-Farbraum durch den Gamut beschränkt.

Die Umrechnung von XYZ in RGB erfolgt durch eine lineare Transformationsmatrix. Die Tristimuluswerte müssen dabei entsprechend skaliert sein, damit die resultierende RGB-Farbe

auch korrekt ist. Normalerweise werden die Werte von dem Tristimulus nach dem Y Wert des hellsten anzuzeigenden Objektes skaliert, also $(X, Y, Z)_{\text{res}} = \frac{(X,Y,Z)}{Y_{\text{hellstes Objekt}}}$. Die Umrechnung zwischen den Farbräumen ist wie folgt definiert:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{7}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{bmatrix}^{-1} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{8}$$

Jedoch ist zu beachten, dass nicht alle Werte von XYZ in den RGB-Farbraum transformiert werden können, da RGB-Werte mit teilweise negativen Werten oder Werten größer als 1 entstehen. Die Farben mit negativen Werten sind nicht darstellbar, da es kein negatives Licht gibt. Bei diesen 2 Fällen wird normalerweise folgendermaßen vorgegangen: Eine Methode besteht darin, die resultierenden Farben durch den Wertebereich des RGB-Farbraumes zu begrenzen (*clipping*), indem Werte kleiner 0 auf 0 und Werte größer 1 auf 1 gesetzt werden. Dieses Verfahren ist im allgemeinen nicht korrekt, da es zu einer Verschiebung des Farbtons führen kann. Jedoch ist dies die einfachste Methode, um sicherzustellen, dass alle Farben angezeigt werden. Alternativ kann man die Farben auf den gültigen Wertebereich beschränken, in dem man eine Entsättigung (engl. *desaturation*) oder eine uniforme Skalierung für Werte ausserhalb des Gamuts verwendet [SFDC00].

Für die Bestimmung der Transformationsmatrizen ist es noch wichtig zu wissen, welchen Weißpunkt man definiert. Im XYZ-Modell gibt es mehrere Weißpunkte. Der „normale“ Weißpunkt ist: $w = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. Jedoch handelt es sich bei der Farbe einer natürlichen Leuchtquelle, wie z.B. der Sonne oder einer Glühbirne, nicht um ideales Weiß. Vielmehr ist die Verteilung der Intensitäten im Spektrum nicht gleichmäßig, so wie es bei dem „normalen“ Weißpunkt der Fall ist. Bestimmt man nun einen Gamut, so kann man mit Hilfe eines Weißpunktabgleiches eine Verschiebung erreichen, so dass für das zu kalibrierende Gerät am Ende auch ein Weiß erscheint, wenn alle 3 Primärvalenzen voll angesteuert werden.

Standardmäßig sind mehrere Weißpunkte definiert, welche auch als Normlicht (engl: *standard illuminant*) bezeichnet werden. Man unterscheidet dabei mehrere Arten:

- Illuminant A
  Grundlage dieses Normlichtes ist ein schwarzer Planckscher Körper, welcher mit einer Temperatur von 2856 Kelvin Licht emittiert. Es wurde die relative Strahlungsverteilung eines solchen Körpers ermittelt.

- Illuminant B und C

  Die Werte wurden aus der Kombination des Normlichtes A mit einem Filter erhalten. Sie sollten das Tageslicht simulieren, wobei B für die Sonne um die Mittagszeit und C für die eines durchschnittlichen Tages steht. Sie sind heute aber nicht mehr im Gebrauch, da die D Illuminants bessere Ergebnisse liefern.

- Illuminant D

  Durch eine allgemeine Berechnung können einzelne *Spectral Power Distributions* in Abhängigkeit von der Temperatur in Kelvin bestimmt werden. Grundlage war hierbei eine Analyse des normalen Tageslichtspektrums, welches von Judd et. al. durchgeführt wurde [JMW64].

- Illuminant E

  Das E steht hier für *equal-energy*, also gleiche Energie. Somit sind die Tristimulus Werte auch bekannt, da es sich um den normalen Weißpunkt handelt $(X, Y, Z) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

- Illuminant F

  Wird für Leuchstofflampen verwendet, wodurch die Spektralverteilung (SPD) von den einzelnen Leuchtstoffen angenähert wird.

In dieser Arbeit wird der Gamut vom sRGB-Farbraum verwendet, was dann zur folgenden Umrechnung von XYZ zum RGB-Farbraum führt [Ros05]:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{9}$$

Aus historischen Gründen ist eine Gamma-Korrektur noch heute in Verwendung. Alte Röhrenmonitore sind nicht in der Lage, eine lineare Ausgabe aus einer linearen Eingabe zu erzeugen. Um dennoch eine gleichmäßige Ansteuerung zu erhalten, wird das Signal entsprechend verändert und mit γ potenziert. Die Gamma-Korrektur ist bei der digitalen Bildübertragung nicht mehr von Nöten, wird aber dennoch genutzt, um die Kompatibilität mit Röhrenmonitoren zugewährleisten. Da dieser Effekt nur bei der Ausgabe auftritt, wird in dieser Arbeit auf die Literatur verwiesen [Poy03].

2.1.3 Andere Modelle

Die zwei vorgestellten Farbmodelle bilden, wie bereits angedeutet, nicht alle möglichen Systeme. Neben der additiven ist noch die subtraktive Farbmischung zu erwähnen. Dieses System wird bei gedruckten Materialien angewendet, und führt bei dem RGB-Modell zu dem CMYK

Farbmodell, welches die Grundlage von professionellen Druckern ist. Hierbei wird das einfallende Licht nicht addiert, sondern von einem (idealerweise) weißen Licht die entsprechenden Farben abgezogen. Die Hintergrundfarbe ist Weiß und die Summation aller Farben führt zu Schwarz. Da es beim Druck nicht sinnvoll ist, alle Farben gleichzeitig zu benutzen, also Cyan, Magenta und Gelb, wurde noch die zusätzliche Farbe Schwarz hinzugefügt, um den Verbrauch der einzelnen Farben zu verringen und den Kontrast zu erhöhen. Somit erhält man das CMYK Farbmodell, wobei das K für *Key* steht.

Es wurde versucht, die einzelnen Farbmodelle intuitiver zu gestalten, so dass einzelne Farben einfach bestimmt werden können und die Farbabstände gleichmäßig sind. Die Änderung der Farbe sollte also möglichst uniform über alle zu Grunde liegenden Werte verteilt sein. Dies war bei dem nativen XYZ auch nicht der Fall, sondern wurde erst durch Erweiterungen im Laufe der Jahre sichergestellt. Alternativen sind der HSV- (Hue, Saturation, Value) oder der HSL-Farbraum (Hue, Saturation, Lightness). Für Fernsehübertragungen wurden auch eigene Farbmodelle definiert, welche die Farbwiedergabe für diese Geräte optimiert, wie z.B. das YUV-Farbmodell. In der Literatur werden noch weitere Modelle genannt.

Das XYZ-Modell bietet für diese Arbeit sehr gute Voraussetzungen, da es mit einzelnen Wellenlängen, bzw. Spektren berechnet wird und man daraus die Farbe bestimmt. Dieses Farbmodell lässt des weiteren eine universelle Einbindung zu, da keine spezielle Konfiguration an dem Ausgabegerät vorgenommen werden muss, bzw. diese vom Endnutzer getätigt werden kann. Die Konfiguration ist einfach durch eine Anpassung der Transformationsmatrix des Gamuts zu realisieren. Es sind Daten für die CMF-Werte in 1nm Schritten vorhanden, so dass physikalische Effekte mit einer entsprechenden Genauigkeit berechnet werden können.

2.2 Physikalische Grundlagen

Die Berechnungen in dem zu erstellenden Modell sollen es ermöglichen in einer SPD einzelne Wellenlängen nach den Gesetzen der Physik zu verändern, so dass am Ende eine neue SPD entsteht. Diese kann dann mittels den CMFs in XYZ-Koordinaten umgerechnet werden, welche dann zur Ausgabe in eine RGB-Farbe transformiert werden können.

2.2.1 Licht - Welle und Teilchen

In der Optik werden physikalische Gesetzmäßigkeiten des Lichts erklärt. Die Untersuchungen in diesem Bereich führten aber recht schnell zu einigen schwerwiegenden Problemen. Effekte wie Interferenz, Beugung oder Brechung können sehr gut durch die Vorstellung erklärt werden, dass Licht eine Welle ist, und sich über die Zeit ausbreitet. Der lichtelektrische Effekt, der von Einstein beschrieben wurde, oder der Compton-Effekt konnten mit diesem Modell allerdings

nicht erklärt werden. Bei Licht musste es sich um Teilchen handeln, den Photonen. Dies führte dazu, dass Licht sowohl Welle als auch Teilchen sein kann, und es je nach Modell anders beschrieben wird. Bis heute ist noch keine einfache Modellvorstellung vorhanden, die den Welle-Teilchen-Dualismus löst. Für die Erklärung von Beugung, Brechung oder Interferenz reicht die Vorstellung, dass Licht eine Welle ist aus.

In der Wellenoptik wird Licht als eine elektromagnetische Welle aufgefasst, und kann durch die Wellengleichung beschrieben werden. Es handelt sich dabei um eine Transversalwelle, d.h. die Welle schwingt senkrecht zur Ausbreitungsrichtung. Somit kann die Welle durch Wellenlänge, Phase und Amplitude beschrieben werden. Die Ausbreitung wird dabei durch eine Anfangsamplitude $\vec{E}_0$, der Wellenlänge λ, einen Wellenvektor $\vec{k}$, der die Richtung der Ausbreitung bestimmt, und einen Ortsvektor $\vec{x}$ beschrieben. So kann das elektrische Feld zum Zeitpunkt t wie folgt bestimmt werden:

$$\vec{E} = \vec{E}_0 f(\vec{k} \cdot \vec{x} - ct) \tag{10}$$

Die Frequenz f ist dabei durch die Lichtgeschwindigkeit c und der Wellenlänge λ des Lichtes bestimmt:

$$f = \frac{\lambda}{c}$$

Die verschiedenen Farben entstehen, indem sich mehrere monochromatische Wellen überlagern und sich eine neue Welle entwickelt. So kann eine einzelnene Welle mehrere Wellenlängen besitzen. Die einzelnen Komponenten sind dann wiederum durch die Wellengleichung berechenbar. Die Intensität einer Welle bestimmt sich aus der Amplitude, die über die Zeit gemittelt wird, deshalb können einzelne Komponenten unterschiedliche Intensitäten besitzen. Sie beschreiben dadurch ein Spektrum.

Obwohl die Lichtwelle senkrecht zur Ausbreitungsrichtung schwingt, besitzt sie immer noch 2 Freiheitsgrade. Durch Benutzung von Polarisationsfiltern kann ein Freiheitsgrad entfernt, oder das Licht komplett absorbiert werden. Die 2 Dimensionen der Polarisation werden mit s (im folgenden perpendicular) und p (im folgenden parallel) bezeichnet, und finden vor allem bei den Fresnel-Gleichungen Anwendung, da die Reflexion abhängig von dieser ist. Unpolarisiertes Licht besitzt keine besondere Ausprägung in Bezug auf die beiden Achsen, und die Gesamtamplitude ergibt sich durch Mitteln der beiden Werte.

Zur Beschreibung von Effekten des Lichtes spielt die Phase der Lichtwelle eine entscheidende Rolle, da sie bei der Überlagerung sowohl zu einer Verstärkung, als auch zu einer Auslöschung führen kann. Das Modell nutzt dabei aus, dass Wellen sich überlagern und die

resultierende Welle aus der Summe der einzelnen Wellen entsteht. Dies wird Superposition genannt.

2.2.2 Brechung von Licht

Trifft eine Welle von einem transparenten Medium auf ein anderes transparentes, so ändert die Welle die Richtung. Diesen Effekt bezeichnet man als Brechung. Sie findet nur statt, wenn sich bei dem Übergang der Medien die Geschwindigkeit der Welle verändert, also die Lichtgeschwindkeit in ihnen unterschiedlich ist, denn die Lichtgeschwindigkeit ist abhängig von der Dichte des Materials. Zur Vermeidung von Verwirrungen wird auch von der Phasengeschwindigkeit gesprochen, die beschreibt, mit welcher Geschwindigkeit sich die Phase einer Welle ausbreitet:

$$c_{\text{Medium}} = v_{\text{Phase}} = \lambda \cdot f \tag{11}$$

Das Gesetz von Snell erklärt, wie die Richtung der Welle bestimmt werden kann. Es gibt aber keinen Aufschluss darüber, ob Teile der Welle reflektiert, absorbiert oder transmittiert werden. Für die Berechnung dieser Teile werden die Fresnel-Gleichungen benutzt (Siehe Kapitel 2.2.5).

Die Herleitung des Snellius'schen Gesetzes lässt sich am einfachsten durch das Fermat'sche Prinzip beschreiben. Dies besagt, dass eine Lichtwelle immer den schnellsten (nicht kürzesten) Weg nimmt, um den Zielpunkt zu erreichen. Ist die Richtung des Lichtes im ersten Medium bekannt, so kann man daraus die Richtung des Lichtes im 2.ten Medium errechnen. Mit Hilfe von Pythagoras und dem Fermat'schen Prinzip erhält man das Gesetz von Snell:

$$\frac{\sin \theta_i}{\sin \theta_o} = \frac{n_2}{n_1} \tag{12}$$

θ_i und θ_o bezeichnen hierbei den einfallenden Winkel, respektive den ausfallenden Winkel, zum Lot des 2.ten Mediums. Die Abbildung 4 zeigt den Strahlengang der einzelnen Wellen. n_1 und n_2 sind die von den Medien abhängige Brechungsindizes (auch Brechzahlen ,engl: *Index of Refraction*, kurz *IOR*), welche sich aus dem Quotienten der Phasengeschwindigkeit des Lichts im Vakuum c_0 und der Phasengeschwindigkeit des Lichts im Medium c_{Medium} zusammensetzen. Somit erhält man folgende Definiton:

$$n_{\text{Medium}} = \frac{c_{\text{Vakuum}}}{c_{\text{Medium}}} \tag{13}$$

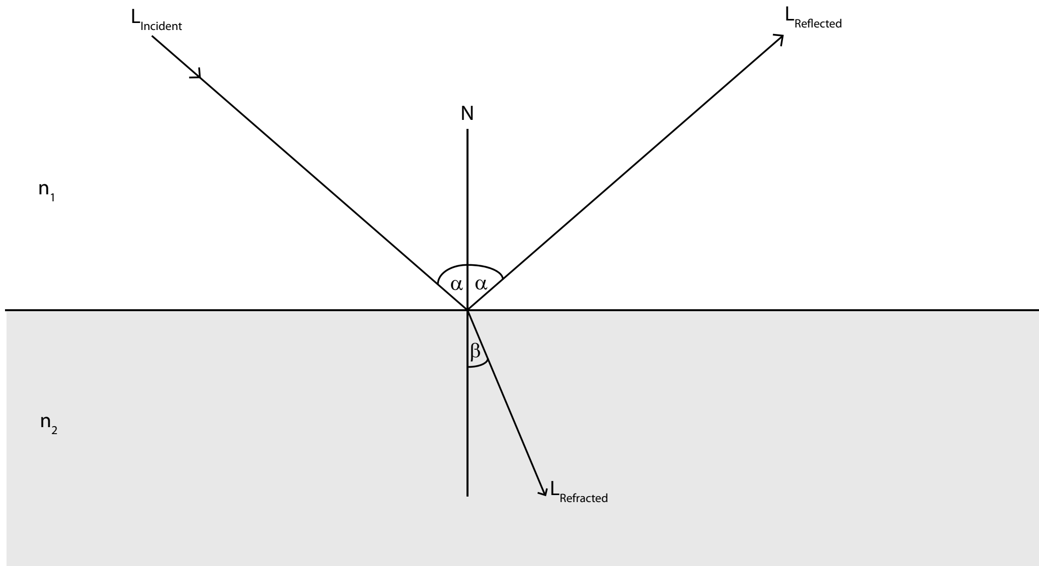


Abbildung 4: Visualisierung des Snellius'schen Brechungsgesetz

Diese Formel gilt dabei nur für isotrope Medien bei denen die Ausbreitungsrichtung der Welle keine Änderungen einer ihrer Eigenschaften zur Folge hat. Bei dem Übergang einer Welle in ein anderes Medium, wird der Strahl gebrochen. Ist der Brechungsindex des ersten Mediums kleiner als der des zweiten ($n_1 < n_2$), so wird die Welle zum Lot der Oberfläche des zweiten Mediums hin gebrochen. Im anderen Fall wird die Welle vom Lot weg gebrochen.

Der Brechungsindex ist dimensionslos, und kann auch als komplexe Zahl aufgefasst werden. Dies ist für die Beschreibung von nichttransparenten Medien vorteilhaft, da in der Definition ein entsprechender Faktor κ (Extinktionskoeffizient) festgelegt werden kann, der dem Imaginärteil des komplexen Brechungsindexes entspricht. Die Definition lautet dann wie folgt:

$$\hat{n}(\lambda) = n(\lambda) + i\kappa(\lambda) \tag{14}$$

Der Extinktionskoeffizient beschreibt dabei, wie Wellen in dem entsprechenden Medium abgeschwächt werden. Die beiden Faktoren, Brechungsindex und Extinktionskoeffizient, können wellenlängenabhängig sein. Benutzt man komplexe Brechungsindizes mit einem großem Imaginärteil, gilt das Snellius'sche Brechungsgesetz nicht mehr, da eine Dämpfung nicht beachtet wird. Verwendet man nur kleine Extinktionskoeffizienten, so bleibt das Gesetz in guter Näherung erhalten.

Wenn die Lichtwelle von einem optisch dichten Medium (großer Brechungsindex n) in ein optisch dünnes Medium (kleiner Brechungsindex n) übergeht, so kommt es ab einem bestimmten Einfallswinkel zu einer Totalreflexion, da die Welle von dem Lot der Oberfläche

weggebrochen wird. In der Berechnung wird der Winkel größer als 90° und der einfallende Strahl dringt nicht mehr in das andere Medium ein. Der Fall, dass der Winkel genau 90° beträgt, kann mittels folgender Formel bestimmt werden:

$$\theta_{\text{total}} = \arcsin\left(\frac{n_2}{n_1}\right) \tag{15}$$

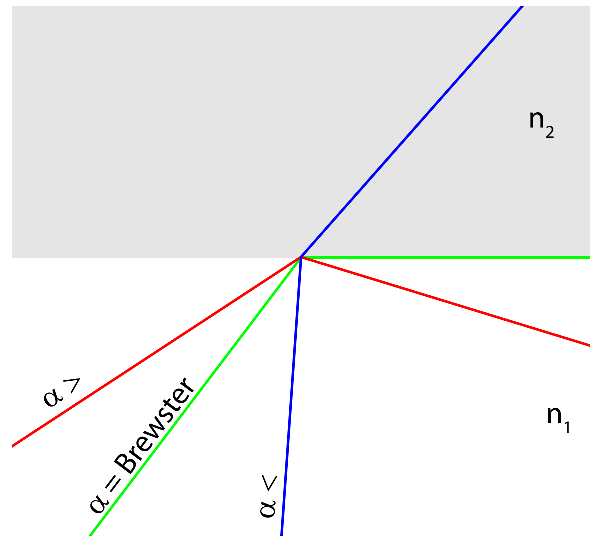


Abbildung 5: Totalreflexion beim Übergang von optisch dichtem zu optisch dünnem Material

Die Totalreflexion ist somit von den Brechungsindizes der beiden Medien bestimmt und variiert damit. Sind die Brechungsindizes der beiden Medien identisch, so sieht man leicht, dass sich der Winkel der einfallenden und der Winkel der transmittierenden Welle nicht unterscheiden, da keine Brechung stattfindet. Für die Totalreflexion ergibt sich: (15): $\theta_{\text{total}} = 90°$. Eine Totalreflexion kann somit nur dann stattfinden, wenn der Einfallswinkel $\theta_i = 90°$ beträgt.

Für die Berechnung der Weglängen in einem Medium ist der Brechungsindex n als Streckungsfaktor zu verstehen, um einen äquivalenten Wert für die Weglänge im Vakuum zu erhalten, da er der Quotient der beiden Phasengeschwindigkeiten ist. Der zurückgelegte Weg in einem Medium mit Brechungsindex n entspricht dem Weg im Vakuum wie folgt:

$$l_{\text{Vakuum}} = n \cdot l_{\text{Medium}} \tag{16}$$

2.2.3 Interferenz

Überlagert man zwei Wellen, so ergibt sich nach dem Superpositionsprinzip eine neue Welle, in

dem man die einzelnen Amplituden der beiden Wellen addiert. Das Resultat einer Interferenz kann man Abbildung 6 entnehmen. Es gibt 3 verschiedene Fälle, wie durch Überlagerung eine neue Welle entstehen kann:

1. Konstruktive Interferenz:
   Beide Wellen befinden sich in gleicher Phase und haben die gleiche Wellenlänge. Durch die Summation der einzelnen Amplituden verstärken sich beide Wellen zu gleichen Teilen und die resultierende Welle besitzt als Intensität die Summe der Intensitäten der einzelnen Wellen.

2. Destruktive Interferenz:
   Beide Wellen haben die gleiche Wellenlänge, sind aber um $\frac{\pi}{2}$ phasenverschoben, und ergeben in der Summe 0. Es kommt zu einer Auslöschung und es entsteht eine Welle mit einer Intensität von 0.

3. Mischung der Wellen:
   Die Wellen haben gleiche Wellenlänge, aber eine Phasenverschiebung von $\Phi \neq 0$ und $\Phi \neq \frac{\pi}{2}$. Bei der Überlagerung kommt es zu einer Mischwelle die beide einzelnen Wellen kombiniert. Es kann zu einer teilweise destruktiven oder konstruktiven Interferenz kommen.

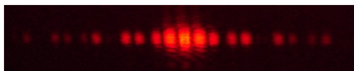


Abbildung 6: Beispiel von Interferenz. Quelle: Uni-München

Damit es zu der Interferenz kommen kann, muss zwischen den Wellen eine konstante Beziehung mit der Phase bestehen, sonst sind die einzelnen Effekte nicht mehr wahrnehmbar. Ist diese konstante Beziehung gegeben, so nennt man das Licht „kohärent“ ansonsten „inkohärent“. Ist das Licht kohärent, so kann es zu einem stabilen beobachtbaren Interferenzmuster kommen. Zu beachten ist, dass Kohärenz keine Eigenschaft einer Lichtquelle ist, sondern sich nur auf die einzelnen Lichtwellen bezieht. Man unterscheidet zwischen zwei Kohärenzbedingungen, die in den jeweiligen Fällen erfüllt sein müssen, damit Licht interferieren kann.

- Zeitliche Kohärenz
- Räumliche Kohärenz

Soll eine Lichtwelle mit einer zeitlich verschobenen Kopie interferieren, so muss die zeitliche Kohärenzbedingung erfüllt sein. Die Zeit, in der die Korrelation der beiden Wellenzüge möglichst groß ist, beschreibt die Interferenzfähigkeit. Dieses Intervall gibt an, wie groß die

maximale Zeitverzögerung sein darf, damit das Licht noch interferenzfähig ist. Man bezeichnet diese Zeitverzögerung als Kohärenzzeit und die Kohärenzlänge ist die Strecke, welche die Welle in der Kohärenzzeit zurücklegt.

Handelt es sich nicht um eine punktförmige Lichtquelle, so kann auch der räumliche Abstand der Wellenfronten in einer Kohärenzbedingung beschrieben werden. Man spricht dann von der räumlichen Kohärenz. Sie beschreibt, wie weit die beiden Wellenfronten voneinander getrennt sein dürfen damit es zur Interferenz kommt.

Eine Lichtquelle, die kohärentes Licht erzeugt, ist z.B. ein Laser. Räumlich inkohärentes Licht kann durch einen Einfachspalt (eine Blende mit einem genügend kleinen Spalt) kohärent gemacht werden, entsprechend dem Huygen'schen Prinzip.(Siehe Kapitel 2.2.4).

Die Kohärenzlänge von natürlichem Licht beträgt etwa 3m, resultierend aus einer Kohärenzzeit von etwa 10ns. Diese Werte basieren auf einem vereinfachten Modell, wobei ein Atom von einem angeregten Zustand in einen weniger angeregten Zustand übergeht und die freiwerdende Energie in Form von Photonen (Lichtteilchen) emittiert wird. Dieser Übergang hat eine durchschnittliche Dauer von ca. 10ns. Da die Photonen aus der Schwingung eines Atoms stammen sind sie kohärent.

Bei einem Laser ergibt sich eine große Kohärenzlänge, da diese auf Grund ihrer Konstruktion sehr monochromatisch sind. Dadurch bieten sich Laser für Interferenzversuche an, denn sie erzeugen ein stabiles Interferenzmuster.

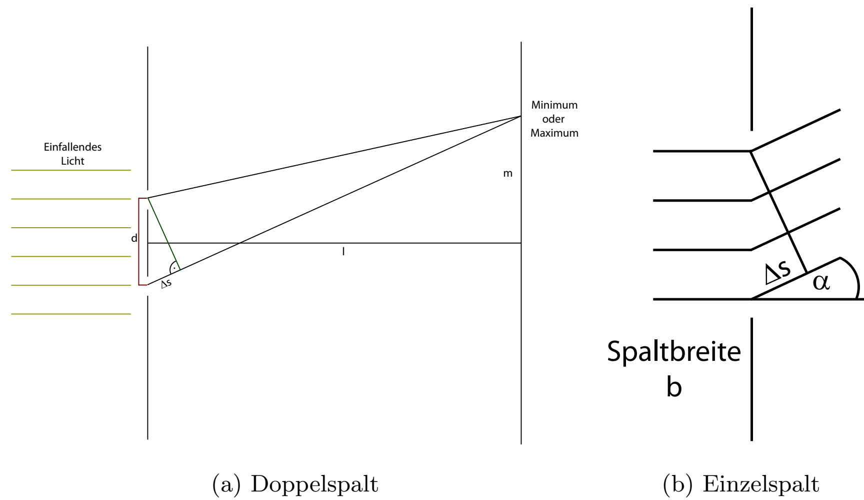
2.2.4 Diffraktion

Die Diffraktion (Beugung) geht auf das Prinzip von Huygens zurück, bei dem jeder Punkt einer Wellenfront Quelle einer neuen Elementarwelle sein kann. Die Wellenfront ist dabei die Fläche, die durch Punkte gleicher Phase von mehreren Wellen definiert wird. Trifft nun eine Wellenfront auf einen Spalt (Einfachspalt, Doppelspalt oder Gitter) entsteht dort eine neue Front, die sich kugelförmig (im 3-dimensionalen Raum) bzw. kreisförmig (2-dimensional) ausbreitet. Da diese Wellenfronten eine räumliche Kohärenz besitzen, können sie interferieren.

Durch die Geometrie eines Doppelspaltes erhält man unter Annahme der Kleinwinkelnäherung:

$$m \approx \Delta s \cdot \frac{l}{d}$$

Wobei m der Abstand von der Mitte der Konstruktion auf dem Bildschirm, Δs der Gangunterschied der Wellen zueinander, l der Abstand vom Bildschirm zum Spalt und d der Abstand der Spaltmitten zueinander ist (Siehe Abbildung 7a). Es kommt dabei zu einer konstruktiven Interferenz, wenn der Gangunterschied Δs ein Vielfaches der Wellenlänge λ ist.



(a) Doppelspalt (b) Einzelspalt

Abbildung 7: Geometrie von Doppel- und Einzelspalt

Man erhält ein Maximum wenn:

$$M_{\text{Maximum Ordnung i}} = i \cdot \lambda \cdot \frac{l}{d} \quad i \in \mathcal{N}_0 \tag{17}$$

Die Orte der destruktiven Interferenz können durch Einsetzen für i durch $\frac{(2i+1)}{2}$ erhalten werden. Die Intensitätsverteilung bei einem Doppelspalt in Abhängigkeit des Winkels α zwischen Normale des Doppelspaltes und des Vektors zwischen Beobachtungspunkt und Doppelspaltmitte sowie Spaltbreite b berechnet sich durch folgende Formel:

$$I(\alpha) = I_0 \left(\frac{\sin\left(\frac{\pi}{\lambda} b \sin\alpha\right)}{\frac{\pi}{\lambda} b \sin\alpha} \right)^2 \cos^2 \frac{\pi}{\lambda} d \sin\alpha \tag{18}$$

Ist es bei der Interferenz am Doppelspalt noch einfach vorstellbar, dass sich zwei kugelförmige Wellenfronten überlagern und auf Grund ihres Weglängenunterschiedes (und der daraus resultierenden Phasenverschiebung) interferieren, so ist es beim Einzelspalt nicht offensichtlich. Betrachtet man die gebeugte Wellenfront zweigeteilt, so kann man sehen, dass einzelne Wellenfronten sich auslöschen können.

Die Bedingung für die Berechnung der Minima ist nun - wie folgt - gegeben.

$$\sin\alpha_i = i\frac{\lambda}{b} \quad i = 1, 2, 3, \ldots \; i < \frac{l}{\lambda} \tag{19}$$

Die Konstruktion und Definition des Dreiecks mit Winkel α ist in 7b abgebildet.

2.2.5 Fresnel - Gleichungen

Wie schon in Kapitel 2.2.2 erwähnt, können der reflektierte und transmittierte Anteil einer Welle mit Hilfe der Fresnel-Gleichungen berechnet werden. Eine Welle wird bei dem Übergang zu einem Medium mit einem anderen Brechungsindex nicht nur transmittiert, sondern auch zum Teil reflektiert. Die Größe der Reflexions- und Transmissionsfaktoren hängt sowohl von den beiden Brechungsindizes, als auch von der Polarisation der einfallenden Lichtwelle ab.

Die Fresnel-Gleichungen für die jeweils parallel und perpendicular polarisierten Welle lauten wie folgt:

$$r_{\text{Parallel}}(\theta_i, \theta_t, n_1, n_2) = \frac{n_2 \cos(\theta_i) - n_1 \cos(\theta_t)}{n_2 \cos(\theta_i) + n_1 \cos(\theta_t)} \tag{20}$$

$$t_{\text{Parallel}}(\theta_i, \theta_t, n_1, n_2) = \frac{2n_1 \cos(\theta_i)}{n_2 \cos(\theta_i) + n_1 \cos(\theta_t)} \tag{21}$$

$$r_{\text{Perpendicular}}(\theta_i, \theta_t, n_1, n_2) = \frac{n_1 \cos(\theta_i) - n_2 \cos(\theta_t)}{n_1 \cos(\theta_i) + n_2 \cos(\theta_t)} \tag{22}$$

$$t_{\text{Perpendicular}}(\theta_i, \theta_t, n_1, n_2) = \frac{2n_1 \cos(\theta_i)}{n_1 \cos(\theta_i) + n_2 \cos(\theta_t)} \tag{23}$$

Definition der Variablen:

- θ_i: Winkel des einfallenden Lichtes zur Normalen der Oberfläche
- θ_t: Winkel des transmittierten Lichtes zur Normalen der Oberfläche
- n_1: Brechungsindex der umgebenden Schicht
- n_2: Brechungsindex des Materials, an dem das Licht reflektiert wird

Im folgenden werden x_{Parallel} mit $x_{\|}$ und $x_{\text{Perpendicular}}$ mit $x_{\perp}$ bezeichnet.

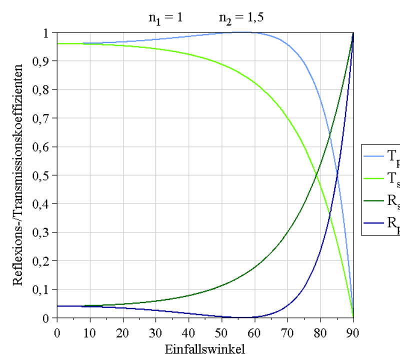


Abbildung 8: Amplitudenverhältnisse nach Fresnel-Gleichungen. Quelle: Wikipedia

Trägt man die resultierenden Werte dieser Gleichungen auf, so erhält man eine Kurve der Amplitudenverhältnisse (Abbildung 8. Man erkennt dabei, dass $r_{\|}$ einen Nulldurchgang besitzt. Dieser tritt bei dem *Brewsterwinkel* α_B auf, an dem der reflektierte Strahl nur perpendicular polarisiert ist. Setzt man die Formel für $r_{\|}$ gleich 0, so erhält man den Brewsterwinkel:

$$r_{\|} = \frac{n_2 \cos(\theta_i) - n_1 \cos(\theta_t)}{n_2 \cos(\theta_i) + n_1 \cos(\theta_t)} = \frac{\tan(\theta_i - \theta_t)}{\tan(\theta_i + \theta_t)}$$

Für $r_{\|}$ ist dies erfüllt, wenn $\theta_i + \theta_t = 90°$ gilt. Setzt man diese Bedingung nun in das Snellius'sche Gesetz ein, erhält man

$$\frac{n_2}{n_1} = \frac{\sin \theta_i}{\sin \theta_i + 90°} = \frac{\sin \theta_i}{\cos \theta_i} = \tan \theta_i$$

Also ist

$$\alpha_B = \arctan \frac{n_2}{n_1}. \tag{24}$$

Da die Brechungsindizes auch komplex sein können, wird die Berechnung des Reflexions- und Transmissionskoeffizienten entsprechend weitergeführt. Die resultierende Amplitude der Welle für Reflexion und Transmission ergibt sich aus:

$$R_i = |r_i|^2 \tag{25}$$

$$T_i = \left| \frac{n_2 \cos \theta_t}{n_1 \cos \theta_i} \right| |t_i|^2 \tag{26}$$

Für den Betrag dieser Variablen gilt:

$$|r_i| = \sqrt{\mathrm{Re}(r_i)^2 + \mathrm{Im}(r_i)^2} \quad \text{(entsprechend für } t_i\text{)}.$$

Da normales Licht im Grunde unpolarisiert ist, können die einzelnen Amplituden gemittelt, und die resultierenden Werte für den Reflexions- oder Transmissionskoeffizienten verwendet werden. Es gilt also:

$$R_{\mathrm{res}} = \frac{R_{\|} + R_{\perp}}{2} \tag{27}$$

$$T_{\mathrm{res}} = \frac{T_{\|} + T_{\perp}}{2} \tag{28}$$

2.3 GPU und Shader

Aktuelle Graphikkarten sind in der Lage eine große Menge von Daten in kürzester Zeit zu verarbeiten. Im Gegensatz dazu stehen die Raytracer, welche die Berechnung des Sichtbildes

durch die Verfolgung von Sichtstrahlen durchführen (daher auch der Name). Dies kann nicht von der Graphikkarte berechnet werden, sondern muss von der CPU übernommen werden. Deren Rechenleistung ist zwar auch in den letzten Jahren enorm gestiegen, doch bleibt keine Zeit für andere Berechnungen, wie sie in modernen Spielen, dem Antriebsrad der Graphikkarten-Industrie, benötigt werden.

2.3.1 Renderpipeline

In der Computergraphik müssen Objekte abgespeichert und eindeutig identifiziert werden. Ebenso muss es eine Vorschrift geben, um aus den gegebenen Daten ein Bild zu erstellen. Man definiert dafür ein entsprechendes 3-dimensionales Koordinatensystem, welches sich je nach Implementierung unterscheiden kann. In *Direct3D* handelt es sich um ein linkshändiges System, d.h. die positive z-Achse zeigt vom Betrachter weg, bei *OpenGL* handelt es sich um ein rechtshändiges System, in dem die positive z-Achse auf den Betrachter zeigt. Beide Systeme sind ohne Probleme ineinander umrechenbar.

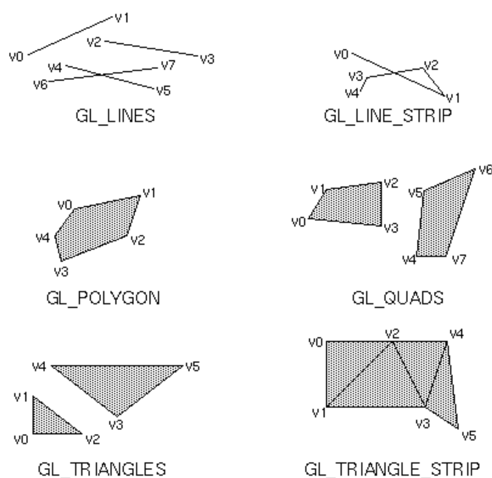


Abbildung 9: Verschiedene standardmäßig verfügbare Polygone. Aus [SWND05]

Bei den Polygon-Renderern werden Objekte durch Polygone definiert, deren Eckpunkte (im folgenden Vertices) Koordinaten im 3-dimensionalen Raum zugewiesen sind. Dabei wird in der Regel das Objekt um den Ursprung eines Koordinatensystems definiert, und man bezeichnet dieses als Modellraum (engl. *Object-Space*). Die Vertices können neben Position auch weitere Informationen, wie z.B. Normale, Farbe oder Tangente, für spätere Berechnungen enthalten. Die Reihenfolge der erzeugten Vertices bestimmt die Orientierung des aufgespannten Polygons. In *OpenGL* werden die Punkte gegen den Uhrzeigersinn definiert. Von den einzelnen APIs werden neben einfachen Dreiecken auch verschiedene komplexere Polygonzüge zur Verfügung gestellt, wie z.B *Triangle Fan, Triangle Strip, Quad* oder auch *Lines*. Dies vereinfacht die Beschreibung von Objekten, ein Beispiel wird in Abbildung 9 gezeigt.

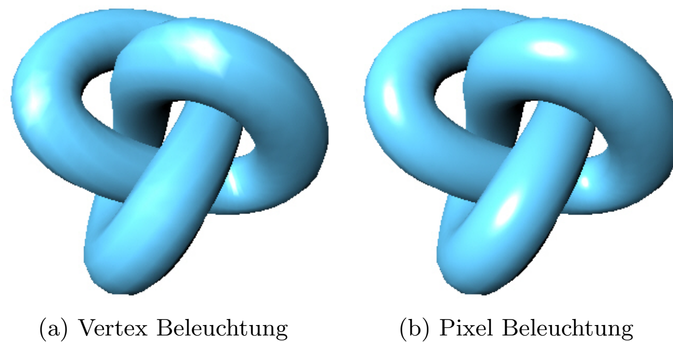
Um Objekte in einer Welt, auch Szene genannt, zu platzieren, müssen diese in ein entsprechendes Koordinatensystem übertragen werden. Die lineare Algebra bietet die mathematische Grundlage, um Koordinatentransformationen durch eine Matrixmultiplikation auszudrücken. Jedes Objekt besitzt eine eigene Transformationsmatrix, die *ModelToWorld*-Matrix oder auch kurz *World*-Matrix genannt wird. Die Berechnung wird in homogenen Koordinaten durch-

geführt, wodurch die Translation (Verschiebung) durch einen einfachen Vektor repräsentiert werden kann. Um ein Objekt mehrmals in einer Szene abzubilden, muss es nicht zwangsläufig mehrmals definiert werden, sondern wird einfach mit einer anderen *World*-Matrix verrechnet. Dies erspart unnötige Daten im Speicher, wodurch die Anwendung beschleunigt wird.

In 3-D Anwendungen wird eine Kamera definiert. Diese Kamera übernimmt die Rolle eines Beobachters, und wird in der Szene entsprechend platziert. Sie kann zur Laufzeit eines Programmes verändert werden, falls die Sicht variabel sein soll. Um die Berechnungen für die sichtbaren Objekte zu vereinfachen, wird die Kamera in den Ursprung gedreht. Die Blickrichtung ist in *OpenGL* entlang der negativen z-Achse. Desweiteren benötigt die Kamera einen *Up*-Vektor, der „Oben“ definiert. Ihre Drehung bewirkt eine Rotation der gesamten Szene. Es wird eine *Viewing*-Transformation durchgeführt. Mit Hilfe der *View*-Matrix werden die einzelnen Vertices der Objekte in den *View-Space* transformiert. Die Matrix kann mit Ursprung, Sichtpunkt und Up-Vektor der Kamera berechnet werden. Da die Transformation von *Object-Space* in *View-Space* eine häufig benutzte Transformation ist, wird eine entsprechende Matrix in *OpenGL* zur Verfügung gestellt, die *ModelView*-Matrix. Diese kann durch einfache Multiplikation (linksseitig) der einzelnen Matrizen berechnet werden. Die einzelnen Transformationsmatrizen werden in *OpenGL* nicht explizit angeboten, und müssen ggf. per Hand berechnet werden.

Die letzte Transformation muss die Objekte auf die Ausgabegröße des Bildschirms beschränken und eine Projektion durchführen. Man spricht deshalb auch von der *Projection*-Matrix. Grundlegend gibt es zwei Projektionen: *a*) Die orthogonale Projektion, in der die Seitenverhältnisse beibehalten werden, weiter entfernte Objekte erscheinen genauso groß, wie Objekte, die sich sehr nahe am Betrachter befinden. Diese wird bevorzugt für technische Abbildungen genutzt, da am Ausgabebild die Größenverhältnisse stimmen. *b*) Die perspektivische Projektion, die der menschlichen Perzeption nachempfunden ist. Objekte, die einen größeren Abstand zu dem Betrachter haben erscheinen kleiner, als Objekte, die sich näher befinden. Für die Projektion wird ein *View*-Volume oder auch *Clipping*-Volume definiert, welches die äusseren Grenzen der Projektion festlegt. Objekte, die innerhalb dieses Raumes liegen, werden angezeigt. Objekte ausserhalb werden in den nächsten Schritten nicht weiter beachtet. Damit wird der Berechnungsaufwand weiter minimiert. Zuletzt wird das Volume auf die Ausgabegröße gebracht, und eventuelle Streckungen oder Stauchungen durchgeführt.

Der nächste Schritt ist die Rasterisierung, in dem die einzelnen Objekte, abhängig von ihrer Farbe, gezeichnet werden. Ab diesem Punkt handelt es sich nicht mehr um eine 3-dimensionale Darstellung, sondern um eine 2-dimensionale Projektion. Die Rasterung erfolgt durch verschiedene Algorithmen, die wichtigsten sind hierbei der *Scan-Line*-Algorithmus und die Rasterung von Dreiecken mit baryzentrischen Koordinaten. Auf die Beschreibung der



(a) Vertex Beleuchtung (b) Pixel Beleuchtung

Abbildung 10: Beleuchtung auf Vertex- und Pixel-Ebene. Aus [Krö08]

Algorithmen wird hier verzichtet. Als Ausgabe erhält man Pixel[6], die das Bild ergeben. Pixel können mehr Informationen als eine Farbe enthalten und werden in OpenGL Fragments genannt. Fehlende Punkte werden interpoliert.

So ergibt sich ein 2-dimensionales Bild der Szene in Abhängigkeit von den Objekten, ihrer Position und der Kamera. Die Farbe der Objekte wird zum Teil schon während der Rasterisierung bestimmt. Mit Beleuchtungsberechnungen in der Pixel-Ebene können einzelne Ungenauigkeiten ausgeglichen, und eine einfachere Beschreibung der Objekte verwendet werden. Wenn man beispielsweise ein Spotlight generiert, so kann dieses in der Vertexebene nicht richtig angezeigt werden, wenn es nicht genau auf einem Vertex liegt. Die Abbildung 10 zeigt den möglichen Unterschied. Auf Pixel-Ebene wird auch die Visibilitätsrechnung durchgeführt, die entscheidet, ob ein Pixel angezeigt wird.

Man erkennt, dass keiner der beschriebenen Schritte ohne den vorherigen durchgeführt werden kann. Jedoch kann jede einzelne Berechnung (sowohl für Vertices als auch für Pixel) unabhängig ausgeführt werden, d.h. es existiert ein hoher Grad an Parallelsierung. Aus der Sequentialität und der Parallelsierung hat sich der Begriff der *Renderpipeline* geprägt. Die Abbildung 11 zeigt die einzelnen Zustände in der Pipeline und auch die Positionen der zugehprigen Shader (Kapitel 2.3.2).

2.3.2 Entwicklung der Shader

Die Entwicklung der heutigen *Graphic Processing Units* (kurz GPU) geht auf die Zeit zurück, in der noch keinerlei Hardwarebeschleunigung für die Graphikausgabe existierte. Alle Graphikberechnungen wurden von der CPU durchgeführt, die, zusätzlich zur Anzeige, auch die Bereitstellung der Daten verarbeiten musste. Die Graphikkarte diente als Datenspeicher und

[6]Pixel: von Picture Element

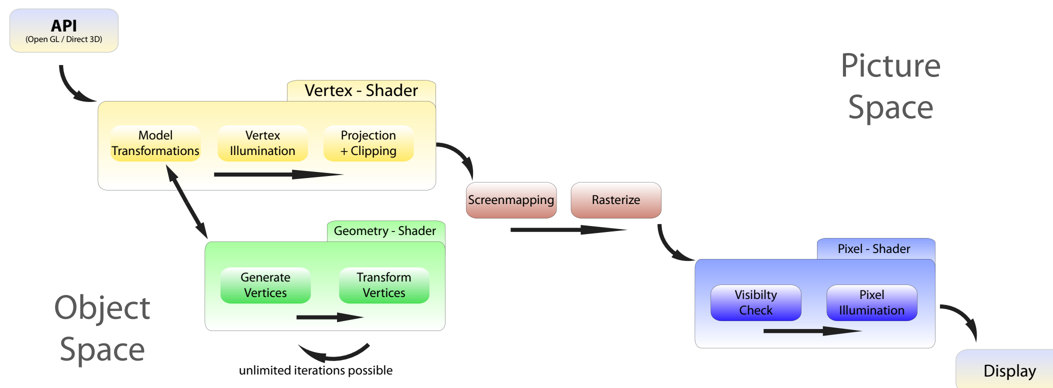


Abbildung 11: Die Renderpipeline

Kommunikationsschnittstelle. Es gab einzelne Hersteller wie SGI oder Evans & Sutherland, die die Grundlage moderner GPUs bildeten, und einige Berechnungen auf ihren Karten durchführen konnten. Die hohen Preise verhinderten einen Durchbruch, und sie wurden vor allem in speziellen Workstations eingesetzt.

Die erste Generation der GPUs konnte die Anzeige von vorberechneten Dreiecken übernehmen, so dass die Rasterisierung auf die Graphikkarte ausgelagert werden konnte. Zusätzlich waren bis zu 2 Texturen auf Objekte anwendbar. Die Operationen waren allerdings stark eingeschränkt. Es gab nur wenige Instruktionen Texturen mit einer Farbe zu verrechnen, um die Farbe eines gerasterten Pixels zu bestimmen. Man zählt zu der ersten Generation von GPUs nVidia's TNT2, ATI's Rage und 3dfx's Voodoo3 [FK03].

Die Auslagerung von Transformations- und Beleuchtungsrechnung (engl: *Transformation and Lighting*, kurz T&L) führte zur nächsten Generation, die die CPU zum Großteil entlastete. Dadurch konnte ein höherer Datendurchsatz erzielt werden, weil die GPUs speziell für die Verarbeitung von vektorbasierten Daten entwickelt wurden. Der CPU stand mehr Leistung für andere Berechnung zur Verfügung und somit konnten komplexere Szenerien entworfen werden. Die Anzahl der einzelnen Instruktionen der GPUs wurde erhöht und weitere Texturformate eingeführt. Im Unterschied zur heutigen Generation von GPUs ist die zweite Generation nicht frei programmierbar [FK03].

Die dritte Generation erlaubt es, auf Vertex-Ebene eine eigene Programmierung anzugeben. Man ist nicht mehr an eine fest verdrahtete Reihenfolge gebunden, die Daten können beliebig verarbeitet werden. Die Programmierung der Graphikkarte kann in Assembler erfolgen und ist somit von der Zielhardware abhängig. Die Berechnungen in der Pixel-Ebene ist in einem höheren Maße konfigurierbar als noch in der zweiten Generation. Mit *Direct3D 8* wurde das `Shader Model 1.0` definiert. Die Berechnungen in der Vertex-Ebene werden durch den Vertex-Shader und die auf Pixel-Ebene durch den Pixel-Shader beschrieben.

Mit Einführung von *DirectX 9* wurden die Hochsprachen in die Graphikkartenprogrammierung eingeführt. Die vierte Generation ist in Vertex- und Pixel-Ebene frei programmierbar. Dies ermöglicht es, komplexe Graphikberechnungen von der CPU direkt auf die Graphikkarte auszulagern. Dies führte zur Definition von `Shader Model 2` und `3`, welche zusätzliche Instruktionen auf der Graphikkarte festlegten.

Mit `Shader Model 4.0` wurde ein zusätzlicher Shader eingeführt, der *Geometry Shader*. In der Pipeline befindet er sich hinter dem Vertex-Shader und ist in der Lage, neue Objekte zu erstellen. Dies kann für Displacementmapping[7]oder dem Erzeugen von genaueren Geometrien genutzt werden. Es kann durch diesen Shader auch ein LOD[8]-System eingeführt werden, solange die nächsten Detailstufe durch eine Rechenvorschrift oder Textur beschrieben werden kann. Auch wurde im `Shader Model 4.0` die Anzahl der Instruktionen pro Shader erhöht und Erweiterungen zur Steigerung der Geschwindigkeit eingeführt.

Der hohe Geschwindigkeitsvorteil der modernen GPUs liegt in ihrer Architektur. Es handelt sich um Streamprozessoren, die Datenströme verarbeiten können und nicht auf einen wahlfreien Zugriff auf die Daten angewiesen sind. Desweiteren handelt es sich um SIMD Prozessoren, die man auch als Vektorprozessoren bezeichnet. SIMD steht für „Single Instruction, Multiple Data“, und bedeutet, dass mit einer Anweisung viele Daten manipuliert werden können. Normale x86 Prozessoren besitzen zum Teil auch SIMD-Erweiterungen, um die Datenausgabe zu beschleunigen.

Die Bezeichnung *Shader* kommt von dem ursprünglichen Einsatzgebiet. Sie sollten die Beleuchtungsrechnung durchführen und sind auf Grund der weitreichenden Erweiterungen inzwischen weniger eingeschränkt. Shader sind Programme, die Berechnungen für das Rendering durchführen. Diese können durch entsprechende Methoden auch für allgemeine Berechnungen genutzt werden. Von nVidia wird dafür z.B. *Cuda* (= *Compute Unified Device Architecture*) angeboten, welche eine Schnittstelle für die sogenannte GPGPU[9]-Programmierung bietet. So eignen sich vor allem Berechnungen, die stark parallelisierbar sind, für eine Implementation auf der GPU. Die Berechnung eines Shaders muss nicht zwangläufig auf einer Graphikkarte stattfinden. Die *RenderMan*-Software benutzt bis heute Software-Shader, die unabhängig von einer Graphikkarte funktionieren.

Für die Programmierung wurden Hochsprachen entwickelt. Von *Direct3D* kam `HLSL` (= *High Level Shading Language*) und von *OpenGL* `GLSL` (= *GL Shading Language*). Beide Hochsprachen werden von den einzelnen APIs in entsprechenden Assemblercode umgewandelt und an die Graphikkarte übertragen, die dann die Ausführung der Programme übernimmt. Ob-

[7]Displacementmapping: Verlagern von Vertices in Abhängigkeit von einer Textur oder Rechenvorschrift
[8]Level Of Detail
[9]GPGPU = *General Purpose Graphics Processing Unit*

gleich sie in der Definition sehr ähnlich sind können sie nur mit der entsprechenden API verwendet werden. nVidia's `CG` entspricht einer alternativen Implementierung von HLSL, da hier Microsoft und nVidia zusammengearbeitet haben. Die Nomenklatur der 3 Sprachen unterscheidet sich in Bezug auf die Berechnungen in der Pixel-Ebene. Bei Microsofts HLSL wird der Shader als *Pixel-Shader* bezeichnet, hingegen verwenden `GLSL` und `CG` die Bezeichnung *Fragment-Shader*. Fragment- und Pixel-Shader bezeichnen die gleiche Berechnungsebene [FK03].

`CG` steht für *C for Graphics* und lässt sich in beide APIs einbinden und bietet damit eine universelle Schnittstelle für die Graphikkartenprogrammierung. Als Grundlage für die Hochsprache diente die Programmiersprache `C` und wird mittels eines Compilers in Maschinencode umgewandelt. Die Entwicklung von `CG` ist noch nicht abgeschlossen und besitzt nicht alle Eigenschaften von `C`. So sind z.B. noch keine Pointer implementiert. Ein Beispel für einen Quellcode ist in Code 1 gegeben.

```
1 float3 simple_fragment(float3 color : COLOR) : COLOR
2 {
3     half3 newColor = half3(color);
4     newColor *= 0.5;
5     return (float3) newColor;
6 }
```

Code 1: CG Beispiel

`CG` erweitert die vorhanden Variablentypen um spezielle Typen, die sich auf Grund der verwendeten Architektur und des anzuwendenden Systems (die Graphikpipeline) nützlich sind (wie z.B. der **half**-Typ). Vektoren können, genauso wie Matrizen, definiert werden und die viel verwendeten Vektorrechnungen, wie Normalisieren oder Skalarprodukt, sind durch eigene Befehle in der Hochsprache verfügbar.

Der **half**-Typ z.B. zeichnet sich dadurch aus, dass er die halbe Präzision des **float**-Typs aber auch nur die Hälfte des Speicherplatzes benötigt. Für die meisten Berechungen ist dieser Typ ausreichend und kann eine Beschleunigung erzielen, da der Speicherplatz der Shader von der Graphikkarte dynamisch zugewiesen wird.

Semantiken (in dem Beispiel `COLOR`) verweisen auf spezielle Register. Beim Aufruf der Funktion **`simple_fragment`** wird der Wert für die Variable **color** von der Farbe des aktuellen Fragmentes bestimmt. Das Programm verändert die Farbe des aktuellen Fragmentes durch den Rückgabewert. Es gibt viele vordefinierte Semantiken, die wichtigsten sind: `COLOR, NORMAL, POSITION, TANGENT, BINORMAL, TEXCOORD0-6`. Die Semantiken `TEXCOORD0-6` bezeichnen beliebige Texturkoordinaten. Diese Register können für die Speicherung von beliebigen Daten genutzt werden.

Zur Definition von Variablen, besonders Texturen und den zugehörigen Samplern, werden *Annotations* genutzt. Durch diese können zusätzliche Daten für eine Variable gespeichert werden. So kann mittels der Annotations ein Wertebereich einer Variablen definiert werden oder für Texturen z.B. der Pfad an dem die Textur zu finden ist. Sie definieren mit `<` und `>` einen Bereich, in dem die zusätzlichen Daten angegeben werden. Hierbei können **Strings**,**Boolean** oder **Float/Int** Variablen verwendet werden.

Für Shader gibt es in `CG` eigenene Datentypen zur Verarbeitung von Texturdaten, die **Sampler**. Diese beschreiben ein externes Objekt, von welchem einzelne Samples gelesen werden können, wie z.B. von einer Textur [FK03]. Es gibt 5 verschiedene grundlegene Samplertypen, die in Tabelle 1 aufgezählt sind.

Sampler Typen	
sampler1D	1-dimensionale Texturen (Funktionen)
sampler2D	2-dimensionale Texturen (Bilder, Normalmaps, etc.)
sampler3D	3-dimensionale Texturen (3D Funktionen)
samplerCUBE	Cube Maps
samplerRECT	beliebige Texturen (beliebige Daten)

Tabelle 1: Grundtypen von Sampler in CG. Aus [FK03]

Ein Vertex- und Fragment-Shader (und Geometry-Shader) können in einer *Technique* zusammengefasst werden. Techniques weisen der Graphikkarte die jeweiligen Programme zu, die zur Laufzeit ausgeführt werden. Eine Technique definiert einen oder mehrere Renderpasses (kurz Pass), in der Vertex- und Fragment-Shader zusammen mit eventuellen API Anweisungen und den Profilen[10] zusammengefasst werden. In einem Multipass-Verfahren können Berechnungen durchgeführt werden, die in einem Durchgang nicht zu berechnen sind, wie z.B. einen Weichzeichner-Effekt. Auf Grund der Parallelisierung ist es nicht möglich, während eines Pass auf andere Pixel zuzugreifen und seine Daten auszuwerten, da keine Synchronisation zwischen den einzelnen Shadern stattfindet. Die Daten eines Multipass-Verfahrens können in sogenannte MRT[11] zwischengespeichert werden, auf die im nächsten Renderpass zugegriffen werden kann. Es ist evident, dass Multipass- langsamer sind als Singlepass-Verfahren, da eine Szene mehrmals gerendert werden muss.

Die Anzahl der Techniques pro `CG` Effect File ist nicht beschränkt. Ein Beispiel einer Technique ist in Code 2 zu sehen.

[10] Definiert das zu verwendende Shader Model
[11] Mutli Render Targets

```
technique TechniqueExample
{
    pass
    {
        //Disable Lighting
        LightingEnable = false;
        //Load Programs
        VertexProgram = compile vp40 vertex_program();
        FragmentProgram = compile fp20 fragment_program(someData);
    }
}
```

Code 2: CG Techniques

Die Übergabe von globalen Variablen kann entweder durch Definition im globalen Bereich der Datei geschehen oder durch die Übergabe einer *uniformen* Variablen. Der **uniform** Bezeichner legt dabei fest, dass die Variable nicht durch das angegebene Programm definiert ist, sondern durch einen anderen Kontext bestimmt wird. Dies könnte durch die API geschehen oder durch andere Variablen in einer `CG` Datei. Variablen, die an eine Semantik gebunden sind, sind vom internen Typ **varying**, da ihre Werte von Vertex zu Vertex bzw. Fragment zu Fragment variieren. Konstante Werte können mittels des **const** Bezeichners gesetzt werden. Diese können dann wie in `C` nicht mehr verändert werden.

Die Möglichkeit **Structures** zu erstellen wird bevorzugt bei der Ein- und Ausgabe von Vertex-Shadern genutzt. Alternativ kann die Ausgabe mit **out** oder **inout** Qualifiern definiert werden. Variablen, die so erzeugt wurden, werden mittels *call-by-result* übergeben, d.h., die aufgerufene Funktion legt deren Wert fest. Der **inout** Qualifier erlaubt zusätzlich die Übergabe eines Wertes an die Funktion. Hierbei ist die Ausgabe nicht nur auf eine Variable beschränkt, es können beliebig viele erzeugt werden. Zusätzlich kann die Funktion einen Rückgabewert besitzen. Die Benutzung von Rückgabewerten oder **out** Qualifiern hat keinen Einfluss auf die Funktionalität der Programme [FK03].

Bei der Entwicklung kann eine Modularisierung durchgeführt werden, da `CG` die Compileroption `#include` zur Verfügung stellt. Wie bei `C` wird die angegebene Datei an die Stelle hineinkopiert, bevor der gesamte Code an den Compiler gesendet wird.

Obwohl HLSL und GLSL einen ähnlichen Aufbau in der Programmierung besitzen und auch an die Programmiersprache `C` angelehnt sind, wurden sie auf Grund ihrer Bindung zu einer spezifischen API nicht ausgewählt, um eine Implementierung sowohl in `DirectX` als auch `OpenGL` zu ermöglichen. Weiterführend wird diese Hochsprache direkt von einem Graphikkartenhersteller [nVidia] angeboten, so dass Optimierungen der Basisfunktionen effektiv durchgeführt werden können. Für die Implementierung von `CG` wurde `C++` mit `.NET` verwendet.

3 State of the Art

Die Grundlage aktueller Polygon-Renderer bildet das RGB-Modell. Die Simulation von einzelnen Beleuchtungen und ihre Berechnungen können einfach durchgeführt werden, da sie auch im RGB-Modell zutreffen. Nicht alle Effekte sind korrekt darstellbar. Durch Vorberechnung und Speicherung der Daten in Texturen oder Vereinfachungen im zu grundeliegenden Modell, können sie jedoch in ausreichender Qualität approximiert werden. Die Ergebnisse variieren je nach Vereinfachung in ihrer Qualität, bieten aber meistens eine ausgewogene Verteilung von Realismus und Geschwindigkeit.

Da es bei Raytracern einfacher ist, ein physikalisches Beleuchtungsmodell zu Grunde zu legen, gab es erste Implementierungen schon sehr früh. Die in den Raytracern verwendeten Methoden lassen sich auch auf eine GPU zu übertragen. Aufbauend auf dieser Tatsache wird das eigene Verfahren umgesetzt.

3.1 Beleuchtung

Eine große Herausforderung für 3D-Renderer ist bis heute, eine glaubwürdige Beleuchtung einer Szenerie zu erzeugen. Die Grundlagen dafür werden dabei von der Physik vorgegeben. Diese genau zu modellieren ist ein sehr schwieriges Unterfangen. In modernen Rendersytemen werden sehr gute Ergebnisse erzielt, jedoch handelt es sich meist um Offline-Renderer, also Nicht-Echtzeit-Anwendungen. In Echtzeit-Anwendungen werden einfachere Modelle gewählt, die versuchen, die optischen Effekte in Formeln auszudrücken. Man kann dabei 2 grundlegende Beleuchtungsmodelle unterscheiden. Zum einen ein globales Beleuchtungsmodell, in dem sich Objekte gegenseitig beeinflussen können, und die Beleuchtung der Gesamtszenerie durch alle Objekte und Lichtquellen entsteht. Zum anderen ein lokales Beleuchtungsmodell, in dem nur Lichtquellen zur Beleuchtung der Szenerie beitragen.

Bei den meisten Echtzeitanwendungen handelt sich um lokale Beleuchtungsmodelle. Durch Vorberechnungen sind auch globale Modelle realisierbar. Diese sind jedoch statisch und können nicht auf Veränderungen in der Szenerie reagieren. Das lokale Beleuchtungsmodell vereinfacht die Berechnung und berücksichtigt folgende 4 Faktoren in der Beleuchtungsgleichung:

$$\text{intensity}_{\text{total}} = \text{intensity}_{\text{specular}} + \text{intensity}_{\text{diffuse}} + \text{intensity}_{\text{ambient}} + \text{intensity}_{\text{emission}} \tag{29}$$

Die einzelnen Komponenten ergeben sich aus einer Modellvereinfachung, bei der sich jeder Punkt aus den einzelnen Intensitäten eines spielgelnden, diffusen, ambienten und emittierenden Anteil zusammensetzt. Jede einzelne Intensität kann dabei durch ein individuelles Modell dargestellt werden.

3.1.1 Lambert-Beleuchtungsmodell

Für diffuse (nicht reflektierende) Objekte wird mit dem *Lambertschen Cosinusgesetz* eine gute Beschreibung geliefert. Ideal diffuse Objekte werfen das Licht nur abhängig vom Einfallswinkel zurück, die Strahlungstärke sinkt mit der Größe des Winkels. Dieses Modell ist nachweisbar sehr nahe an der Realität und kann durch folgende Formel berechnet werden:

$$\text{intensity}_{\text{diffuse}} = \max((\vec{N} \odot \vec{L}), 0) \cdot k_{\text{diffuse}} \cdot \text{lightcolor}_{\text{diffuse}} \cdot \text{objectcolor}_{\text{diffuse}} \quad (30)$$

Das Skalarprodukt von Normale $\vec{N}$ und Vektor zum Licht $\vec{L}$ ergibt dabei den Cosinus des Winkels zwischen den beiden Vektoren, die normalisiert sein müssen. k_{diffuse} ist ein Skalierungsfaktor, der die einzelnen Teile der Beleuchtung untereinander angleicht. Die beiden Farben werden komponentenweise multipliziert und ergeben die resultierende diffuse Farbe des Objektes.

3.1.2 Phong-Beleuchtungsmodell

Der spiegelnde Anteil der Beleuchtung kann nach Phong[12] durch die Berechnung eines reflektierten Vektors $\vec{R}$ geschehen. Diese Modellvorstellung liefert eine gute Beschreibung für die Glanzlichter von Objekten, ist physikalisch jedoch nicht ganz korrekt. Hierbei wird mehr Licht reflektiert als eingestrahlt wird, was gegen den Energieerhaltungssatz der Physik verstößt. Die Berechnung ist sehr einfach durchführbar, und die Formel wird in (31) gezeigt.

$$\text{intensity}_{\text{specular}} = (\vec{R} \odot \vec{V})^{m_{\text{shininess}}} \cdot k_{\text{specular}} \cdot \text{lightcolor}_{\text{specular}} \cdot \text{objectcolor}_{\text{specular}} \quad (31)$$

Das Skalarprodukt berechnet den Cosinus zwischen dem an der Normalen reflektierten Lichtvektor $\vec{R}$ (siehe Gleichung (32)) und dem Vektor zur Kamera $\vec{V}$. Der spiegelnde Anteil ist nur dann zu sehen, falls das Reflexionsgesetz der Physik (Einfallswinkel gleich Ausfallswinkel) erfüllt ist und wird durch den Cosinus des reflektierten Vektors wiedergegeben. Der Exponent $m_{\text{shininess}}$ beeinflußt die Streuung der Strahlen, also wie glänzend die Oberfläche des Objektes ist. k_{specular} bildet wieder einen Skalierungsfaktor, der den spiegelnden Anteil in der Beleuchtungsgleichung bestimmt.

$$\vec{R} = 2 \cdot (\vec{N} \odot \vec{L})\vec{N} - \vec{L} \quad \text{mit } |\vec{N}| = 1, |\vec{L}| = 1 \quad (32)$$

Durch $\text{objectcolor}_{\text{specular}}$ kann bestimmt werden, um welches Material es sich bei dem zu beleuchtenden Objekt handelt. Hierbei gilt, dass Objekte mit einer transparenten reflektierenden Schicht (größtenteils nichtleitende Materialien) die Farbe des Lichtes und Metalle die

[12] Bui-Tuong Phong, wobei Phong eigentlich der Vorname ist!

Farbe des Objektes reflektieren, da bei Metallen das Licht in die Gitterstruktur eindringt und die Atome in Schwingung versetzt.

Der *ambient* Anteil ist der Versuch eine allgemeine Hintergrundstrahlung in einem einfachen Term zusammenzufassen. Dies ist der Anteil, der durch gegenseitige Beleuchtung, Reflektionen von Objekten, etc. entsteht. Man kann den ambienten Term als „Raumgrundhelligkeit“ [Krö08] zusammenfassen. Er stellt eine Approxmation der indirekten Beleuchtung dar, da bei lokalen Beleuchtungsmodellen nur der Anteil von den vorhandenen Lichtquellen verrechnet werden kann. Der ambiente Anteil ist ungerichtet und wird zu allen Intensitäten addiert. Man erhält daraus den in Formel (33) gegebenen Faktor, der sich durch die komponentenweise Multiplikation der Farben ergibt.

$$\text{intensity}_{\text{ambient}} = \text{lightcolor}_{\text{ambient}} \cdot \text{objectcolor}_{\text{ambient}} \tag{33}$$

Der letzte Faktor in der Beleuchtungsgleichung ist der sogenannte emittierende Term. Dieser entspricht der Farbe, die ein Objekt selbst ausstrahlt, wobei es sich hierbei nicht um eine Lichtquelle handeln muss. Dies ist ein Lichtanteil, der immer zu dem Objekt dazuaddiert wird.

Die Beleuchtungsgleichung kann erweitert werden, so dass diffus und spiegelnder Anteil abhängig von der Entfernung des Betrachers sind. Man kann diese Rechnung auch an Spotlights anpassen, so dass die Szenerie nur innerhalb eines Lichkegels beleuchtet wird.

Ein Problem dieser Gleichung ist, dass die Summe größer als 1 sein kann, die Werte für die Farbe aber innerhalb des Bereichs von $[0, 1]$ liegen müssen. Man kann zwar die Werte in diesen Bereich clampen, oder eine Skalierung durchführen, doch resultiert dies meist in einer Farbverschiebung. Daher sollte - durch passende Wahl der Skalierungsfaktoren - darauf geachtet werden, dass die Summe nicht größer als 1 wird. Dies erleichtert auch die Fehlersuche bei falscher Farbausgabe.

3.1.3 Blinn-Beleuchtungsmodell

Da die Berechnung zur Zeit der Einführung des Phong-Beleuchtungsmodell zu aufwändig war, wurde versucht, eine Vereinfachung für die Berechnung zu finden. Hierbei hatte Blinn die Idee, den reflektierten Vektor durch den *Halfway*-Vektor auszudrücken. Dieser lässt sich, im Vergleich zu der aufwändigen Berechnung bei der Reflektion (1 Addition, 2 Multiplikationen, 1 Skalarprodukt), durch Addition und anschließende Normalisierung bestimmen (1 Addition + 1 Normalisierung). Wird die Normalisierung effektiv implementiert, können hier große Geschwindigkeitssteigerungen erzielt werden, da die Beleuchtung pixelweise berechnet wird.

Die Beleuchtungsgleichung bleibt identisch, nur anstelle des Skalarproduktes aus reflek-

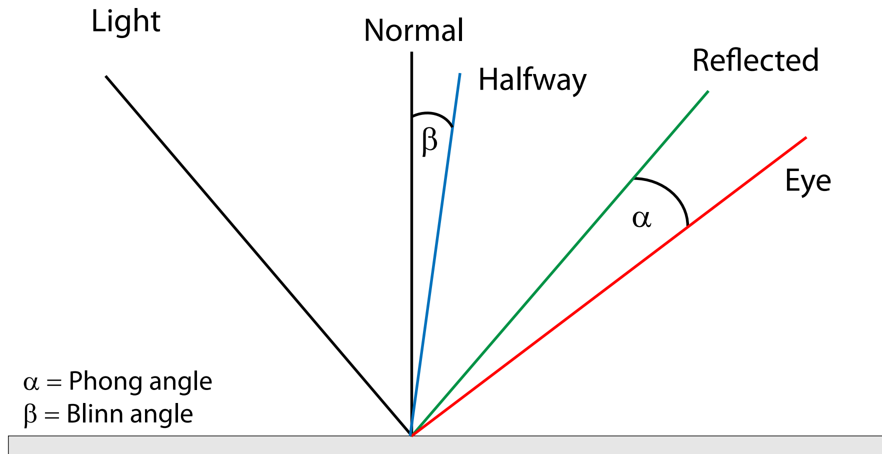


Abbildung 12: Vergleich zwischen Blinn- und Phong-Vektoren

tiertem Lichtvektor und Kameravektor wird das von Normale und Halfway-Vektor berechnet. Das Ergebnis ist sehr ähnlich und es ergibt sich die Gleichung (34) für den spiegelnden Anteil.

$$\text{intensity}_{\text{specular}} = (\vec{N} \odot \vec{H})^{m_{\text{shininess}}} \cdot k_{\text{specular}} \cdot \text{lightcolor}_{\text{specular}} \cdot \text{objectcolor}_{\text{specular}} \tag{34}$$

Ein Vergleich der Vektoren zwischen den beiden Ansätzen von Blinn und Phong ist in Abbildung 12 dargestellt. Man erkennt, dass die Winkel zwischen den entsprechenden Vektoren (Blinn: Normale und Halfway, Phong: Reflektion und Kamera) sehr ähnlich sind.

Die Modelle von Blinn und Phong basieren zum Großteil auf empirischen Daten und haben keine physikalische Grundlage. Doch erhält man mit ihnen eine glaubwürdige Beleuchtung von Objekten. Im Laufe der Zeit wurden noch andere lokale Beleuchtungsmodelle eingeführt. Von Bedeutung ist das Torrance-Sparrow Modell, in dem die Oberfläche facettiert wird, und die einzelnen Facetten als Spiegel betrachtet werden. Dadurch ist es möglich, einzelne lokale Effekte, wie Brechungsindex oder Dichte mit einzubeziehen. Der Aufwand bei der Berechnung ist aber so hoch, dass dieses Beleuchtungsmodell in Echtzeitanwendungen selten verwendet wird. Wie schon bei Blinn wird der spiegelnde Anteil variiert. Dazu werden noch Fresnel-Reflektanz und eine Dichtefunktion eingeführt. Auch gibt es einen Geometrie-Faktor, der eigenen Schattenwurf berücksichtigt. Dadurch kann eine genauere Berechnung der lokalen Beleuchtung durchgeführt werden.

Schlick [Sch94] optimierte die Berechnungen von Torrance-Sparrow und erreichte durch Approximierungen physikalisch korrektere Ergebnisse als mit dem Phongmodell. Er definierte dafür eine besondere BRDF (Siehe Kapitel 3.1.4), welche die nötigen Anforderungen erfüllte.

3.1.4 BRDF

Eine BRDF (*Bidirectional Reflectance Distribution Function*) beschreibt allgemein das Reflexionsverhalten einer Oberfläche. So können alle bisher beschriebenen lokalen Beleuchtungsmodelle einfach durch eine BRDF beschrieben werden. Abhängig von Licht- und Kameraposition gibt eine BRDF an, wie sich Intensität und Farbe des Lichtes bei dem Beobachter zusammensetzen. Es ist auch möglich, eine BRDF durch Messung zu bestimmen, so dass gemessene Daten für die Wiedergabe der Einstrahlung genutzt werden können. Die Menge der gesammelten Daten ist enorm groß und findet daher in Echtzeitanwendungen selten einen Einsatz.

Der Vorteil dieser Funktionen besteht in der einfachen Modellierung anisotropischer Materialien (Licht durchdringt die Materialen unterschiedlich, abhängig von der Richtung). Auch liefert es eine einfache Steuerung, da nur 4 Parameter benötigt werden: Winkel des einfallenden Lichtes zur Normalen und Winkel der Kamera zur Normalen, wobei die Winkel in Kugelkoordinaten angegeben werden. Sie können aus den Positionen und der Normalen der Oberfläche berechnet werden. Zusätzlich kann eine BRDF noch eine Dimension für die Wellenlänge λ besitzen und die Koordinaten des getroffenen Punktes der Oberfläche können in das Ergebnis einfließen. Betrachtet man die 4-dimensionale Version, so ergibt sich folgende Funktion:

$$f_r(\omega_i, \omega_r) = \frac{dL_r(\omega_r)}{L_i(\omega_i)\cos(\theta_i)d\omega_i} \tag{35}$$

Dabei ist θ_i der Winkel zwischen ω_i und der Normalen der Oberfläche. L_r bezeichnet die spezifische Intensität (engl: *radiance*) des emittierten Lichtes. Der Nenner gibt die Intensität des einfallenden Lichtes an.

Die lokalen Beleuchtungsmodelle sind sowohl in Raytracer als auch Polygon-Renderer implementierbar. Die einzelnen Farben in den Modellen können entweder einzelne Wellenlängen oder eine RGB-Farbe sein. Die globalen Beleuchtungsmodelle bilden das Aushängeschild der Raytracer, da hier einzelne Strahlen durch die ganze Szenerie gesendet werden, und auf Grund ihrer Intensität diese erhellen. Das Verfahren kann nicht einfach auf einen Polygon-Renderer übertragen werden, auch wenn es mögliche Implementierungsansätze gibt [NPG05].

3.2 Interferenz in der Graphik

Für die Darstellung von dynamischen Interferenzeffekten benötigt man eine wellenlängenabhängige Repräsentation des Beleuchtungsmodells, da Interferenz durch die Auslöschung von einzelnen Wellenlängen entsteht (Kapitel 2.2.3). Diese Repräsentation kann jedoch so stark vereinfacht werden, dass sie auf GPUs mit `Shader Model 3.0` lauffähig ist. Eine statische

Repräsentation kann gewählt werden, wenn im Voraus bekannt ist, welche Farben bei der Interferenz auftreten.

3.2.1 Raytracer

Bei Raytracern ist eine Implementierung relativ einfach umsetzbar, da die Beleuchtungsrechnung anstelle von RGB nur auf jede Wellenlänge erweitert werden muss. Einschränkungen sind hierbei nur durch Approximationen in der Berechnung gegeben, welche auf Grund von Effizienz gemacht werden müssen.

Eine dünne Schicht könnte im Raytracer so modelliert werden, dass die Strahlen sich nach der Fresnel-Reflektanz aufteilen, und sich danach unabhängig voneinander durch die Szene propagieren. Fallen die Strahlen nun auf eine Lichtquelle, so kann der Weg zurückverfolgt, und nach Kriterien für Interferenz untersucht werden. Sind diese gegeben, so löschen sich einzelne Wellenlängen aus. Da die Berechnung für jede Wellenlänge unabhängig durchgeführt wird, werden Brechung und Diffraktion korrekt behandelt.

Raytracer können vollkommen auf einen Rasterisierungsprozess verzichten, wodurch eine genauere Abtastung der Objekte stattfindet und eine Quantisierung nicht zwangsläufig notwendig ist. Objekte werden in der Regel mit einer impliziten Formel angegeben, durch die eine Schnittpunktberechnung erfolgen kann.

Die Effizienz hängt hierbei linear von der Anzahl der Objekte ab, und kann durch einzelne Optimierungen (Boundingboxes, *spatial Coherence*, etc.) gesteigert werden. Es besteht die Möglichkeit, einen hybriden Ansatz für den ersten Schnitttest zu nutzen, so dass die Graphikkarte das erste getroffene Objekt anzeigt (Rasterisierung), und von diesem ausgehend die nächsten Strahlen errechnet werden. Die meiste Leistung wird bei der Rekursion der Strahlberechnung, wegen Aufteilung der einzelnen Strahlen (in der Regel 2: reflektiert und transmittiert), und der Schnittpunktberechnung jedes Strahls mit den möglichen Objekten benötigt. Ein Raytracer ermöglicht es hingegen durch seine Funktionsweise, einfach Schatten zu generieren, ohne dass zusätzliche Berechnungen durchgeführt werden müssen. Es ergeben sich weiche Schattenübergänge, welche durch größere Lichtquellen (z.B. eine Leuchtstoff-Lampe) erzeugt werden. Für Polygon-Renderer sind Methoden entwickelt worden, die dieses Verhalten durch aufwendige Berechnungen simulieren.

Da die Ausgabe des Bildes, wie auch beim Polygon-Renderer, statisch für eine Ausgabe bleibt, ist hier ein großer Grad an Parallelisierung vorhanden. In Zukunft werden die GPUs wohl durch freie Programmierung, unabhängig von der Renderpipeline, es ermöglichen, Raytracing durchzuführen (Stichwort: Intel Larrabee).

3.2.2 Polygon-Renderer

Interferenzberechnung ohne Zugriff auf einzelne Strahlen scheint im ersten Augenblick quasi unmöglich. Daher wird zuerst nur eine statische Situation betrachtet. Hierbei ist die Interferenzberechnung als Momentaufnahme anzusehen, wodurch einige Vereinfachungen und Beschleunigungen in den Berechnungen gemacht werden können.

Abbildung 13: Ergebnis des dünnen Schichten Effektes von nVidia. Quelle: FX Composer Gallery

nVidias Beispiel für die Berechnung von dünnen Schichten benutzt eine Textur, um die nötigen Farben zu ermitteln. Dabei wird im Vertex-Shader eine Sichttiefe berechnet, die mit der vorgegebenen Tiefe der dünnen Schicht einen TextureLookup-Vektor[13] erzeugt.

Die Textur enthält die vorberechnete destruktive Interferenzfarbe, die sich für jede Tiefe ergibt. Durch den TextureLookup-Vektor wird nur noch bestimmt, welche Farbe ausgewählt wird. Dieser Wert wird dann genutzt, um den spiegelnden Anteil der Beleuchtungsrechnung zu verändern, wohingegen der diffuse Anteil unberührt bleibt. Ein Ergebnis dieses Effektes ist in Abbildung 13 zu sehen.

Dies ist ein übliches Verfahren in der Computergraphik. Einschränkungen sind jedoch durch die fest vorgegebenen Brechungsindizes und Materialeigenschaften gegeben. Es wäre dennoch denkbar, eine größere Textur zu definieren, welche die vorberechneten Interferenzen bereithält.

Die Brechung, welche die Irideszenz (Regenbogen) auf einer CD erzeugt, ist mit einem Polygon-Renderer durch einige Approximationen darstellbar. Dabei liefert das Shader-Programm aus dem ersten Band der Reihe GPU Gems [Fer04] eine mögliche Realisierung. Dort wird eine einfache Methode beschrieben, aus einer Textur, welche das komplette sichtbare Spektrum abdeckt (Siehe Abbildung 14), für eine Wellenlänge die zugehörige Farbe zu bestimmen. Mit Hilfe einer Diffraktionsgleichung (36) wird die resultierende Farbe bestimmt.

$$|u|d < n < 2|u|d \quad (36)$$

[13] TextureLookup-Vektor: Koordinaten in der Textur, deren Stelle ausgelesen soll

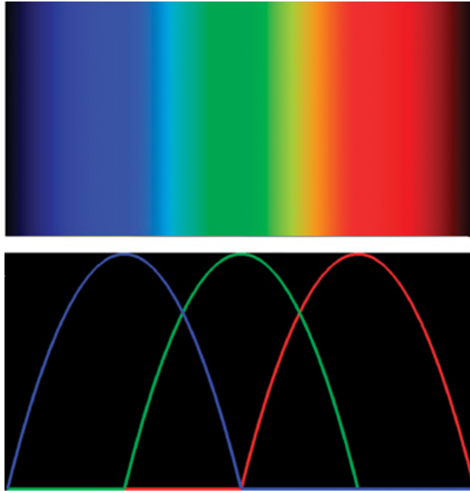


Abbildung 14: Regenbogen Textur. Aus [Fer04]

Dabei ist $u = \sin\theta_i - \sin\theta_o$, und d der Abstand des Gitters der CD. θ_i ist wieder der Winkel zwischen Licht und Normale, θ_o der Winkel zwischen Normale und Betrachter. Das Kriterium für eine Auslöschung der einzelnen Wellenlängen ergibt sich aus der Weglängendifferenz. Für die Bestimmung der resultierenden Farbe werden die einzelnen Wellenlängenfarben aufaddiert, für die sich eine positive Interferenz ergibt. Diese Vereinfachung ist korrekt, da sich im negativen Fall unendlich viele Strahlen mit verschiedenen Weglängenunterschieden überlagern, und somit immer - vereinfacht - die Bedingung für destruktive Interferenz besteht.

Der Sonderfall für $u = 0$ wird in dem Shader explizit abgefangen, da dieser eine direkte Reflexion wiederspiegelt, und nach der Formel nicht bestimmt werden kann.

Die Autoren benutzen die Funktion `blend`, um eine Regenbogentextur indirekt zu erstellen. Die Tristimulus-Werte werden dabei durch 3 Parabeln approximiert, deren Werte nicht kleiner als 0 werden können. Die verwendete Funktion lautet:

$$bump(x) = |x| > 1 \; ? \; 0 \; : \; 1 - x^2 \tag{37}$$

Mittels einer *Conditional expression* wird hierbei die `if`-Abfrage ersetzt. Benutzt man diese Blendfunktion für die Erstellung der einzelnen RGB-Komponenten der Farbe, erhält man folgende Ausdrücke:

$$R(\lambda) = bump(C \cdot (y - 0.75)) \tag{38}$$

$$G(\lambda) = bump(C \cdot (y - 0.50)) \tag{39}$$

$$B(\lambda) = bump(C \cdot (y - 0.25)) \tag{40}$$

Dabei bildet y die einzelnen Werte in den Bereich $[0.5, 1.0]$ ab. Der Faktor C beeinflusst die Erscheinung der Regenbogentextur um einen konstanten Wert, die Autoren haben für ihre Implementierung 4 gewählt. Ein eventueller Speedup könnte hierbei durch eine Textur erzielt werden, da die Berechnung im Voraus getätigt werden kann, und der y-Wert nur die Position der gewählten Wellenlänge wiederspiegelt [Fer04].

Der Fall für $u = 0$ wird hier durch eine Beleuchtung, die nach Ward [War92] definiert

Abbildung 15: Ergebnisse der Diffraktion nach Stam. Aus [Fer04]

wurde, um anisotroptische Materialien zu beschreiben, abgehandelt. Die Formel für die Berechnung lautet nach den Autoren wie folgt [Fer04]:

$$anis = \text{highlight color} \cdot \exp\left(-\left(\frac{ru}{w}\right)^2\right) \tag{41}$$

r bestimmt die Rauheit der Oberfläche, und w ist der Anteil des *halfway*-Vektors in Normalenrichtung, welche durch den Cosinus bestimmt werden kann. Benutzt man die angegebene Formeln für die Berechnung, können gute Ergebnisse erzielt werden. Als Einschränkung ist hier zu nennen, dass immer 8 Wellenlängen für die Berechnung der Interferenzfarbe genutzt werden, und dieser Wert wohl durch Versuche erzielt worden ist. Die Berechnung der Interferenz wird auch nur auf Vertex-Basis durchgeführt, was unter Umständen zu Ungenauigkeiten führen kann. In den gezeigten Beispielen in Abbildung 15 ist eine zusätzliche Berechnung eingeführt, die die Spiegelung nach den Fresnelkriterien an einer dünnen Schicht berechnet. Dadurch werden realistischere Ergebnisse erzielt, jedoch wird nicht angegeben, wie diese Berechnung durchgeführt wird. Die Ergebnisse zeigen unter bestimmten Winkeln gute Ergebnisse, teilweise wirken aber die einzelnen Farben zu stark betont.

Eine explizite Berechnung der Interferenz an einer dünnen Schicht wird von Durikovic [DK06] in einer wellenlängenbasierten Methode beschrieben. Durch die Nutzung des XYZ-Farbraumes kann mit einzelnen, positiven Intensitätswerten für jede Wellenlänge gerechnet werden. Zur Bestimmung der Intensität einer Wellenlänge werden die Fresnel-Gleichungen (2.2.5) verwendet. Somit können beliebige dünne Schichtenmodelle erstellt und simuliert werden.

Nutzt man die Fresnel-Gleichungen, so kann für die einzelnen Lichstrahlen die Reflexion an jeder Schicht berechnet werden. Die einzelnen Reflexionsfaktoren kann man dann durch Multiplikation zu dem resultierenden Faktor zusammenfassen. Da das Licht innerhalb einer dünnen Schicht mehrmals gebrochen und reflektiert werden kann, und jedes mal die Energien sich nach den Fresnel-Gleichungen aufteilen, wird die Intensität immer geringer. Deshalb können Strahlen ab einer gewissen Ordnung ignoriert werden, da ihr Anteil an der resultierenden Farbe gering ist.

Für die Bestimmung der Intensitäten nutzt [DK06] eine geometrische Reihe, wobei die Nomenklatur der einzelnen Reflexions- und Transmissionsfaktoren wie folgt festgelegt ist: $r_{i,j}$ ist der Reflexionsfaktor von Schicht i zu Schicht j. Da Licht eine Welle ist, kann es durch die Wellengleichung charakterisiert werden und für die Intensität der Reflexion ergibt sich:

$$I_r = r_{0,1} \cdot \Phi_j \exp i(\delta - \omega t) \tag{42}$$

Diese Gleichung ist eine komplexwertige Variante der in Kapitel 2.2 angegeben Formel für die Bestimmung der Welle. Führt man diese Reihe nun fort, so erhält man die von [DK06] verwendete Formel:

$$\Phi_r = \left\{ r_{0,1} + \frac{(t_{0,1} \cdot t_{1,0}) \cdot r_{1,2} \exp i\gamma}{1 - (r_{1,2} \cdot r_{1,0}) \exp i\gamma} \right\} \cdot \Phi_i \tag{43}$$

Die reflektierte Intensität ergibt sich also aus dem Produkt von Reflexionsfaktor und einfallendem Licht. Licht wird hier nicht absorbiert, da es sich um transparente Materialen handelt. Diese Gleichung kann nun umgeformt werden, so dass man den Reflextionsfaktor $r_{0,2}$ erhält, der die Amplitude des reflektierten Lichtes von der untersten Schicht angibt. Da das Licht auch das Basismaterial durchdringen kann ist der Transmissionskoeffizient wichtig und kann in einem System ohne Absorption durch die Energieerhaltung berechnet werden.

Die einzelnen Koeffizienten müssen im Grunde komplex berechnet werden, da die Brechungsindizes jedoch reellwertig sind, ist der Imaginärteil immer 0 und die Berechnung kann entsprechend vereinfacht werden. Man erhält somit:

$$R_{0,2} = |r_{0,2}|^2 = |\frac{-r_{1,0} + r_{1,2} \exp i\gamma}{1 - (r_{1,2} \cdot r_{1,0}) \exp i\gamma}|^2 \tag{44}$$

Man kann dabei $\exp i\gamma$ durch $\cos(\gamma) + i \sin(\gamma)$ ersetzen, wobei der Sinus in der Rechnung dann wegfällt, da bei der Multiplikation mit dem reellwertigen Reflexionsfaktor $r_{1,2}$ bzw. dem Produkt $(r_{1,2} \cdot r_{1,0})$ dieser 0 wird. Der Transmissionkoeffizient ergibt sich aus dem Reflexionskoeffizient einfach durch:

$$T_{0,2} = 1 - R_{0,2}$$

Die Koeffizienten können nun schließlich in die Lichtberechnung eingesetzt werden, und man erhält die von [DK06] verwendete Gleichung für die Lichtberechnung:

$$I_r(\lambda) = I_i(\lambda) R_{0,2}(\lambda) + I_t(\lambda) T_{0,2}(\lambda) \tag{45}$$

Die Implementierung dieses Verfahrens resultiert in sehr guten Ergebnissen. Dabei werden, um mehr Geschwindigkeit zu erreichen, die einzelnen Reflektionskoeffizienten vorberechnet

und in einer Textur gespeichert. Damit ist das System theoretisch zwar frei berechenbar, Änderungen während der Laufzeit sind aber nicht möglich. Auch basiert das System auf einer Unabhängigkeit der Wellenlängen. Das trifft bei Phosphoreszenz / Fluoreszenz aber nicht zu [DK06]. Zu beachten ist auch, dass die einzelnen Koeffizienten abhängig von ihrer Polarisation unterschiedlich sind, und somit getrennt berechnet werden müssen.

Durikovic beschreibt auch eine Methode, die Umgebung für die Beleuchtung zu benutzen. Für die Realisierung werden mehrere Vereinfachungen verwendet, da z.B. eine RGB-Farbe durch unendlich viele verschiedene Spektren dargestellt werden kann. Die durch Interferenz und Illumination erhaltenen Resultate sind in Abbildung 16 zu sehen.

Abbildung 16: Zylinder mit dünnen Schicht (Dicke: $100 - 400$nm). Aus [DK06]

Dieses Verfahren erlaubt es auch nicht, Interferenzen für komplexe Brechungsindizes zu berechnen. Sun et. al. [SW08] veröffentlichten dafür ein Verfahren, welches auf der Rendersoftware *RenderMan* basiert. Dabei werden die Farben nach einem eigenen Verfahren [Sun06] in Spektren aufgeteilt, so dass in dem RGB-Modell basierten System mit einzelnen Wellenlängen gearbeitet werden kann. Die Berechnung der entsprechenden Spektren wird dabei durch eine Fourier-Transformation durchgeführt, welche in [Sun06] beschrieben wird.

Die Berechnung eines einzelnen dünnen Schichtensystems erfolgt im Grunde nach der Gleichung, wie sie von Durikovic bestimmt wurde. Somit ergibt sich für einen dünnen Film folgender Reflexionskoeffizient:

$$R_{\text{film}}(\lambda) = \frac{r_{0,1}^2 + |r_{1,2}|^2 + 2 \cdot r_{0,1}[\text{Re}(r_{1,2}) \cdot \cos\delta + \text{Im}(r_{1,2}) \sin\delta]}{1 + r_{0,1}^2 \cdot |r_{1,2}|^2 + 2 \cdot r_{0,1} \cdot [\text{Re}(r_{1,2}) \cdot \cos\delta + \text{Im}(r_{1,2}) \sin\delta]} \tag{46}$$

Mit $\delta = \frac{4\pi n_1 d \cos\theta}{\lambda}$. Vergleicht man nun Gleichung (46) mit (44) erkennt man, dass es sich bei ersteren um die ausgewertete Form handelt. Die Vereinfachung, den Sinus zu ignorieren, ist folglich für transparente (und somit reelle Brechungsindizes) Objekte korrekt. Somit erhält man die ausgewertete Formel für transparente Schichten:

$$R_{\text{film}}(\lambda) = \frac{r_{0,1}^2 + r_{1,2}^2 + 2 \cdot r_{0,1} \cdot r_{1,2} \cos\delta}{1 + r_{0,1}^2 \cdot r_{1,2}^2 + 2 \cdot r_{0,1} \cdot r_{1,2} \cos\delta} \tag{47}$$

Unter Verwendung der Formel (46) kann man nun auch für freie (schwebende) dünne Filme eine Gleichung herleiten, wie sie z.B. bei Seifenblasen vorkommen. Man erhält nach [SW08] folgende Formel:

$$R_{\text{Free Film}} = \frac{2r_{0,1}^2(1-\cos\delta)}{1+r_{0,1}^4-2r_{0,1}^2\cos(delta)} \tag{48}$$

Diese Formel kann nach Sun [SW08] noch weiter vereinfacht werden und man erhält:

$$R_{\text{Free Film}} = \frac{2r^2(1-\cos\delta)}{1+r^4-2r^2\cos\delta} \tag{49}$$

Für beide Formeln (48),(49) wurde ausgenutzt, dass der Reflexionsfaktor in diesem besonderen Fall richtungsunabhängig ist. Da somit $r = -r$ gilt, und es nur einen einzigen Reflexionsfaktor gibt, kann die Berechnung entsprechend effizient durchgeführt werden.

Durch das Modell von Sun ist es möglich, auch komplexe Brechungsindizes zu verwenden, wobei der Aufwand für die Berechnung für die Fresnel-Gleichungen stark ansteigen. Da eine nichttransparente Schicht nur als Basisschicht vorkommen kann, ist der zusätzliche Berechnungsaufwand überschaubar. Die Berechnung für opaque Materialien folgt der Formel (46). Die Herleitung der resultierenden Formel wird im Appendix A durchgeführt. Das Ergebnis weicht von dem von [SW08] angegebenen ab:

$$r_{\parallel} = \frac{(n_2^2+\kappa_2^2)\cos^2\theta_1 - n_1^2(u^2+v^2) + i(2n_1\cos\theta_1(\kappa_2 u - n_2 v))}{(n_2\cos\theta_1+n_1 u)^2+(\kappa_2\cos\theta_1+n_1 v)^2} \tag{50}$$

$$r_{\perp} = \frac{n_1^2\cos^2\theta_1 - (n_2^2+\kappa^2)(u^2+v^2) + i(-2n_1\cos\theta_1(n_2 v+\kappa_2 u))}{(n_1\cos\theta_1+n_2 u-\kappa_2 v)^2+(n_2 v+\kappa_2 u)^2} \tag{51}$$

Als Erweiterung für dieses Modell können auch wellenlängenabhängige Brechungsindizes und Extinktionskoeffizienten implementiert werden. Dies wird bei Sun et. al. berücksichtigt, und jeweils mit einer zusätzlichen Textur realisiert, in der die entsprechenden Daten gespeichert werden.

Trotz vieler Vereinfachungen, z.B. werden nur 6 Stützpunkte für die Tristimuluswerte verwendet, wird für die Berechnung eines Bildes der Größe 800×600 im Schnitt 1 Sekunde benötigt [SW08]. Dies ist trotz der hohen Genauigkeit noch kein zufriedenstellendes Ergebnis. Diese Methode ist auch nicht in der Lage mehrere dünne Schichten zu bearbeiten.

Ein entsprechender Ansatz für ein mehrschichtiges System wurde von Hirayama et. al. [HKYM01] entwickelt und wird in den eigenen Verfahren (Kapitel 4) vorgestellt.

Die Implementierung ist bei dem System von Sun et. al. in *RenderMan* durchgeführt worden. Somit ist fraglich, inwiefern sie in allgemeinen Shadern zum Einsatz kommen kann, da die *RenderMan*-Software eine eigene Shader-Schnittstelle besitzt. Jedoch zeigt dieses System, dass auch komplexe Berechnungen auf einer GPU durchgeführt werden können.

Durchscheinende Effekte und Sichtstrahlen sind aber so nicht realisierbar, da hier die grundlegende Eigenschaft des Polygon-Renderers die notwendigen Berechnungen verhindert.

So müsste die Transparenz von Objekten explizit in die Beleuchtung miteinfließen, welche zum Zeitpunkt der Rasterisierung abgeschlossen sein muss. Raytracer besitzen diese Limitierung nicht, so dass diese z.B. Prismen korrekt berechnen und angezeigen. Allerdings ist es mit diesem Modell möglich einzelne Interferenzeffekte darzustellen, welche direkt auf der Oberfläche des Objektes erscheinen.

4 Eigene Verfahren

Die Entwicklung des eigenen Modells greift den Implementationsgedanken von [DK06] auf. da dieser keine Optimierung für die einzelnen CMF Werte vornimmt, kann hier eine Iteration über alle vorhandenen Wellenlängen für die Berechnung genutzt werden, um die resultierende Farbe zu bestimmen. Als Shader-Programmiersprache wurde `CG` von nVidia gewählt, die Graphik-API ist `OpenGL`. Für die Anzeige der entwickelten Shader wurde ein eigener Viewer in `C++` mit `.NET` geschrieben. Der Aufbau des *ShaderViewers* wird hier nur kurz erklärt, denn die Entwicklung des eigenen Beleuchtungsmodells bildet den Schwerpunkt dieser Arbeit.

4.1 Shader Viewer

Obwohl die bekannten Entwicklungsumgebungen *RenderMonkey* von AMD und *FX Composer* von nVidia eine große Funktionalität für die Entwicklung und Ausführung eines Shaders bieten, sind die vorhandenen Limitierungen gravierend. So ist der *RenderMonkey* nur bis `Shader Model 3.0` implementiert und kann daher nicht die volle Leistung der neusten GPU-Generation ausschöpfen. Dadurch entstehen starke Einschränkungen in der Programmierung, welche die Implementation eines allgemeinen Modells unmöglich machen. In diesem `Shader Model` ist z.B. der neue Geometry-Shader nicht enthalten, auch können Texturen nicht über einen **Integer** abgefragt werden. Dafür wird eine vollkommene Unterstützung für **Integer** benötigt, die erst in `Shader Model 4.0` eingeführt wurde. Vorher wurden **Integer** als - auf einen ganzzahligen Bereich - geclippte **float**-Variablen aufgefasst. In älteren Shader-Modellen ist der TextureLookup nur durch eine **float** Variable unterstützt. Dabei werden die Koordinaten im Bereich von $[0, 1]$ ausgewertet und auf die Größe des resultierenden Bildes gemappt. Da es dabei zu Rundungsfehlern kommen kann, bzw. man nicht immer genau auf einem Texel[14] landet, muss man entscheiden, wie die Daten ausgelesen werden. Dabei kann im 1-dimensionalen Fall entweder eine Interpolation zwischen den zwei betroffenen Texeln stattfinden (linear), oder es wird der Texel gewählt, der am nächsten zu dem gesuchten Punkt liegt. Die lineare Interpolation bietet bei Bildern gute Ergebnisse, ist aber beim Auslesen von kritischen Daten, wie z.B. CMF-Werten, eher unvorteilhaft.

FX Composer bietet zwar die Unterstützung für das neuste `Shader-Model 4.0`, ist aber auf Grund von mangelnder Benutzerfreundlichkeit und Instabilität nicht für größere Projekte geeignet. Nativ können in dem Programm nur Objekt und Kamera gedreht aber nicht sehr intuitiv gesteuert werden. Jedoch bietet der *FX Composer* eine große Unterstützung für Texturen und deren Einbindung und besitzt ein gutes Interface für die Bearbeitung der Variablen, die in dem Shader genutzt werden. Dadurch lässt sich der angewendete Shader gut

[14] Texel: Texture Element

zur Laufzeit verändern und die Auswirkung der Variablen sehr einfach beobachten.

Sowohl *FX Composer* als auch *RenderMonkey* bieten eine solche Benutzeroberfläche, jedoch zeigen sie unterschiedliche Herangehensweisen. Für die Entwicklung des eigenen Programms, das unabhängig von einer Entwicklungsumgebung Shader anzeigen soll, werden grundlegende Ideen von den beiden oben genannten Programmen adaptiert. So ist ein wesentlicher Bestandteil die *inplace*[15] Bearbeitungsmöglichkeit der vorhandenen Variablen. Sind in einem Shader-Programm mehrere `Techniques` vorhanden, so sollten diese direkt auswählbar sein und auf das ausgewählte Objekt angewendet werden können.

Unter dieser Vorgabe wurde mit `OpenGL` eine `Viewer`-Klasse geschrieben, die die Verarbeitung für `CG`-Shader-Programme einbindet. Die Klasse wurde dabei weitestgehend von einer API abstrahiert, um auch eine Implementation in `Direct3D` oder anderen Graphik APIs zu ermöglichen. Dafür wird das interne Datenformat in das von `OpenGL` umgewandelt. Die Anfragen von `CG` werden ebenfalls durch diese Klasse gesteuert.

Die größte Problematik entstand bei der Einbindung der Texturen, da diese erst in der API eingebunden sein müssen, bevor sie von `CG` geladen werden können. Um dieses Problem zu umschiffen, wurden entsprechende Interface-Klassen geschrieben, die die nötigste Funktionalität im Voraus definieren und universell für jede API eingesetzt werden können.

Da sowohl Licht, Objekt als auch Kamera unabhängig von einander gedreht werden sollten, wurden entsprechende Klassen mit Rotations- und Positionsdaten gespeichert, welche zur Laufzeit des Rendervorgangs ausgewertet werden. Man erhält dadurch eine starke Kapselung. Die einzelnen Objekte können unabhängig initialisiert und genutzt werden. So wäre es z.B. denkbar, auch mehrere Lichtobjekte zu implementieren, um in einer Szene mehrere Leuchtquellen verwenden zu können.

Für die Implementation von mehreren Objekten wurde ein entsprechendes Interface geschrieben, welches die unterliegenden Objekte verwaltet. Jedem Objekt ist dabei eine individuelle Position, Technique des Shaders, Geometrie, etc. zugewiesen. Diese können in der Szene unabhängig von einander verändert werden. Auch ist es möglich, mehrere Objekte zu gruppieren und diese gleich zu transformieren.

Da in `OpenGL` nativ keine *ModelToWorld* und *View*-Matrix vorhanden ist und von `CG` eigene Matrizen definiert werden können, wurde eine entsprechende Matrix- und zusätzlich eine Vektor-Klasse geschrieben. Diese enthalten die grundlegenden Funktionalitäten für die Matrix- bzw. Vektorrechnung, so dass aus den gegebenen Daten eine *ModelToWorld*- bzw. *View*-Matrix erstellt werden kann. Für die Implementierung wurden nur 4×4 Matrizen und 4-elementige Vektoren verwendet. Unter Benutzung von Quaternionen[16] für die Rotation, zur

[15] Direkte Manipulation der Daten

[16] Quaternionen sind eine Erweiterung für den 3-dimensionalen Raum und erlauben eine einfache Beschrei-

Vereinfachung der Rotation und der Vermeidung des sogenannten *Gimbal-Locks*[17] können die Matrizen wie folgt bestimmt werden: Jedem Objekt ist eine Position und Rotation zugewiesen. Die resultierende Rotation wird durch Multiplikation von Quaternionen erstellt. Aus diesem Quaternion erstellt man eine Rotationsmatrix, die wie folgt definiert ist (für normalisierte Quaternionen):

$$M_{\text{rot}} = \begin{pmatrix} (w^2+x^2-y^2-z^2) & (2xy+2wz) & (2xz-2wy) & 0 \\ (2xy-2wz) & (w^2-x^2+y^2-z^2) & (2yz+2wx) & 0 \\ (2xz+2wy) & (2yz-2wx) & (w^2-x^2-y^2+z^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{52}$$

Dabei sind x, y, z, w die Komponenten des Quaternions.

Aus den Positionsdaten, welche der Translation des Objektes entsprechen, kann eine Translationsmatrix M_{trans} erstellt werden, wobei diese einer Identitätsmatrix und die Komponenten der letzten Spalte dem Vektor $v^T = (x, y, z, 1)$ entsprechen. Objekte sollen dabei um die eigene Achse gedreht werden, weshalb zuerst die Rotationsmatrix angewendet wird, welche danach mit der Translationsmatrix multipliziert wird. Die *ModelToWorld*-Matrix ist somit wie folgt definiert:

$$M_{\text{world}} = M_{\text{rot}} \cdot M_{\text{trans}} \tag{53}$$

Die *View*-Matrix wird durch das Kamera-Objekt definiert. Zur Berechnung der Matrix werden die Rotationen in Quaternionen umgerechnet und eine Rotationsmatrix M_{rot} erstellt. Die Translation der Kamera wird in 2 Matrizen aufgeteilt, da eine Translation entlang der Z-Achse unabhängig von der Rotation sein soll und also einem Zoom entspricht. Man erhält somit die *View*-Matrix wie folgt:

$$M_{\text{view}} = M_{\text{transZ}} \cdot M_{\text{rot}} \cdot M_{\text{transXY}} \tag{54}$$

Die globalen Parameter des Shader-Programms werden in einer entsprechenden Klasse verarbeitet und die Werte bei jeder Änderung aktualisiert. Für die Ausgabe auf dem Bildschirm werden die Daten zusätzlich in Variablen gespeichert, so dass sie von einem Programm ausgelesen werden können.

Der *ShaderViewer* ist in zwei Teile aufgeteilt, der Anzeige und dem Befehlsbereich. In der Anzeige wird der aktuelle Shader auf die definierten Objekte angewendet. Die Objekte können mit der Maus manipuliert und die Größe des Fensters dynamisch angepasst oder über

bung der Rotation eines Objektes.

[17] Bei freier Rotation kann es zu dem Verlust eines Freiheitsgrades kommen.

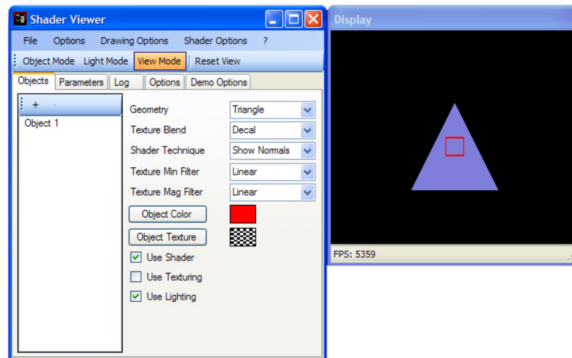


Abbildung 17: Standard-Ansicht des ShaderViewers

ein Kontextmenu eingestellt werden. Ein FPS-Zähler zur Messung der Performanz ist genauso integriert wie die Möglichkeit, Bilder zu speichern (im **.png**-Format), die Farbe des Pixels an der Mausposition auszulesen oder den Renderprozess zu unterbrechen. Ein Vollbildschirmmodus ist implementiert. Für Screenshots[18] wird die Ausgabe des Bildes auf einen Framebuffer umgeleitet und dieser in einer Datei gespeichert. Die Auflösung der Screenshots kann seperat angegeben werden. Framebuffer werden auch beim Auslesen der Farbe des Pixels an der akutellen Mausposition genutzt. Dadurch können die Werte mit einer höheren Genauigkeit bestimmt werden und müssen nicht auf 8 **bit** beschränkt sein. Dies ist hilfreich, falls man evaluierte Daten auf eine genügende Genauigkeit in der Berechnung des Shaders überprüfen möchte. Eventuelle Rechenfehler können deutlicher erkannt werden. Zusätzlich werden bei der Ausgabe der Farbinformationen die Daten in verschiedene Basistypen umgewandelt, so dass auch die am Bildschirm erscheinende 32 **bit** Farbe und der dazugehörige **integer**-Wert ausgegeben werden. Eine Umrechnung für Normale wurde auch implementiert. Werden die Normalen in den positiven Bereich gemappt, so können sie durch die Umkehrung der Rechnung wieder ausgewertet werden und man erhält die Richtung der Normalen. Eine graphische Anzeige wäre zusätzlich denkbar. Das Mapping der Normalen in den positiven Wertebereich geschieht nach Gleichung (55).

$$\vec{n}_{\text{mapped}} = \text{normalize}(\vec{n}) * 0.5 + 0.5 \tag{55}$$

Ebenfalls können mittels der Tastatur Manipulationen am Renderprozess, den Objekten oder dem System ausgeführt werden. Als gutes Feature hat sich das Zurücksetzen der einzelnen Objekte, des Lichtes und der Kamera, sowie die direkte Auswahl von Kamera und Licht durch die Tastatur herausgestellt.

Der Befehlsbereich stellt eine GUI für die Bearbeitung der Variablen, der Texturen und

[18] Das Ausgabebild wird in einer Datei gespeichert

Objekte zur Verfügung. Die Parameter des Shaders werden dynamisch aktualisiert und in einem entsprechenden Bereich angezeigt. Wird die Eingabe durch die vorhandene GUI verändert, so wird der Renderprozess unterbrochen und nur bei Veränderung der Variablen aktualisiert. Schließt man die Eingabe, so wird mit dem eventuell neuen Wert der Renderprozess weitergeführt.

Als Standardeffekt für den *ShaderViewer* wurden *Debug Shader* geschrieben, die Daten von der Szenerie auswerten. Implementiert wurden folgende Shader:

- Anzeige der Objektnormalen
- Anzeige der Textur-Koordinaten des Objektes
- Anzeige des Vektors zur Kamera
- Anzeige des Vektors zum Licht

Diese wurden statisch in den *ShaderViewer* implementiert, so dass sie bei allen eingesetzten Techniques verwendet werden können, um direkte Informationen über das vorhandene Objekt zu erhalten. Die Verwaltung der einzelnen Techniques wird dabei durch einen Effekt innerhalb von `CG` durchgeführt, so dass eventuelle Optimierungen in dem SDK von `CG` direkt übernommen werden.

Zum Vergleich der Framerate mit einem direkten System, kann die Shaderbenutzung objektweise deaktiviert werden. Dies ermöglicht auch einen direkten Vergleich zwischen der Ausgabe von `CG` und der direkten Ausgabe von `OpenGL`. Zu beachten ist hierbei, dass `CG` von `OpenGL` genutzt wird, um die Shader der Graphikkarte anzusteuern, so dass die Ausgabe immer mittels `OpenGL` geschieht.

Der *ShaderViewer* unterstützt fast alle geläufigen Texturformate, wozu **BMP**-, **JPEG**-, **PNG**- und **TGA**-Bilder gehören. Eine grundlegende Implementierung für das **DDS** Format ist vorhanden, jedoch wird nur der erste Typ dieses Formats unterstützt. Für das Einlesen von **float**-basierten Daten wurde ein eigener Datentyp **FTXT** definiert. Hierbei wird die resultierende Texturgröße definiert, danach folgen die Daten zeilenweise. Jede Zeile enthält die Daten für die vier Farbkanäle RGBA. Diese Werte werden dabei mit einem Semicolon abgetrennt. Sind nicht genügend Daten vorhanden, wird das Einlesen abgebrochen und die Daten verworfen. Dieses Format wird für das Einlesen exakterer CMF-Werte in den Shader genutzt.

Damit Texturen zur Laufzeit des Programms in einem Shader-Effekt[19] effektiv geändert werden können, ist ein *TextureViewer* vorhanden, welcher die Textur-Daten aus der entsprechenden Sammlung von Texturen lädt und diese in visueller Form zur Verfügung stellt. Durch

[19]Ein Shader-Effekt bezeichnet die Sammlung mehrer Programme und Techniques

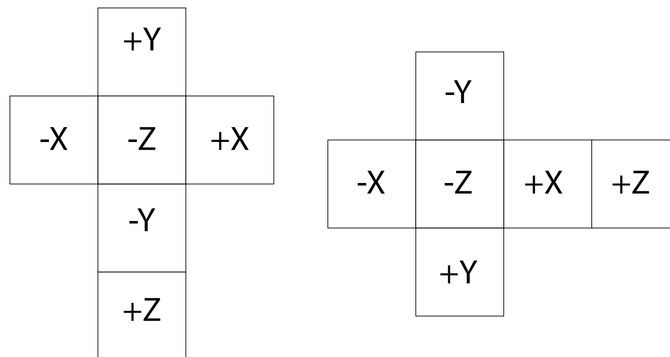


Abbildung 18: Aufteilungsvarianten für die Erstellung von Cubemaps

einfaches Auswählen können die Texturen zugewiesen werden. Dieses Vorgehen zur Texturauswahl ist auch in den bekannteren Editoren üblich und wurde dementsprechend adaptiert. Das Laden neuer Texturen wird mittels eines Dialoges in dem *TextureViewer* gesteuert, welcher nach dem Auswählen der Datei die entsprechenden Elemente in der Sammlung der Texturen anlegt, so dass die geladene Textur anschliessend zur Auswahl steht.

Texturen werden in `CG` mit verschiedenen Formaten unterstützt, welche im **Sampler** geladen werden. Die unterstützten Formate sind in Tabelle 1 auf Seite 27 angegeben. In der aktuellen Version des *Shader Viewer* werden 1D-, 2D-Texturen und CubeMaps unterstützt. 1D-Texturen werden intern als 2D-Texturen mit der Höhe von 1 Texel verwaltet. CubeMaps setzen sich aus 6 individuellen 2D-Texturen zusammen, welche die einzelnen Seiten beschreiben und eine **Environment Map** bilden. Diese können entweder in einer **DDS**, wozu bei jeder Seite der CubeMap zusätzlich die einzelnen Mipmap-Stufen gespeichert werden, oder als zusammengesetztes Bild der einzelnen Seiten gespeichert sein. Für die Verarbeitung des zusammengesetzten Bildes gibt es zwei verschiedene Vorgehensweisen, die Bilddaten zu extrahieren:

Ist das Bild höher als es breit ist, so setzen sich die Seiten des Würfels durch die in Abbildung 18a gezeigte Reihenfolge zusammen. Im anderen Fall wird die Reihenfolge in Abbildung 18b gewählt. Sollten die Seiten gleiche Größe aufweisen, wird der erste Fall gewählt. Zu beachten ist, dass die einzelnen Bildteile gleich groß sein müssen, da sonst eine Verschiebung auftritt. Die Aufteilung des Bildes wird durch einfache Teilung des Bildes in entsprechende Rechtecke erzielt.

Aus den extrahierten Daten werden zugleich die entsprechenden Texturen für eine SkyBox gespeichert. Bei **DDS** Texturen müssen die Daten decodiert werden, da sie komprimiert gespeichert werden. Eine SkyBox spiegelt den Hintergrund der Szenerie wieder und kann entweder eine Box oder eine Sphäre sein. Zu beachten ist, dass bei der Erstellung der SkyBox

keinerlei Translation Einfluss nehmen kann, da sonst die Kamera ausserhalb der SkyBox landen könnte. SkyBoxes werden benutzt, um einer Szenerie zusätzliche Realität zu verleihen und einen Horizont zu erzeugen. Ergebnisse wirken glaubwürdiger, wenn ein entsprechendes Cubemapping mit der zugehörigen Skybox durchgeführt wurde.

Da der *ShaderViewer* unabhängig von jeglicher Entwicklungsplattform ist, kann jeder Editor für die Erstellung der Shader-Effekte benutzt werden. Dies garantiert eine flexible Anpassung an das genutzte System für die Entwicklung des Shaders. Auf Basis des *ShaderViewers* wurde auch die Entwicklung des eigenen Beleuchtungsmodells durchgeführt.

4.2 Konzeption des Beleuchtungsmodells

Mit diesem System soll eine wellenlängenbasierte Beleuchtungsrechnung auf Basis eines Polygon-Renderers durchgeführt werden. Dafür muss eine passende Repräsentation gefunden werden, damit die Daten effektiv verarbeitet und in einem Shader-Programm ausgeführt werden können.

Für die Darstellung einzelner Wellenlängen dienen Spektren, die auf den sichtbaren Bereich der menschlichen Perzeption beschränkt sind. Dieser umfasst etwa den Bereich von 360nm bis 830nm, für den auch CMF-Werte des XYZ-Farbmodells vorhanden sind [SS06]. Es empfielt sich, die Reichweite etwas zu erhöhen, da es Effekte gibt, die einzelne Wellenlängen verschieben, wie es z.B. bei der Fluoreszenz und Phosphoreszenz vorkommt. Computersysteme arbeiten auf einer binären Basis und eine effektive Speicherung legt Daten in diesem Format ab. Man erhält damit als nächste geeignete Größe für die Daten $(830 - 360 = 470)$ einen Wert von 512. Man kann ausserdem noch genügend Wellenlängen ausserhalb des sichtbaren Bereiches speichern. Das Intervall wurde auf $[324 - 836]$ festgelegt, wobei eine Verschiebung zu den kürzeren Wellenlängen gewählt wurde. Damit können auch in Zukunft ultraviolette Lampen implementiert und simuliert werden. Die Speicherung der Daten kann natürlich auch auf einer anderen Basis erfolgen.

Die Speicherung der Datenmenge ist zu groß, um sie lokal in einem Shader speichern zu können. Daher wird eine Repräsentation als Textur gewählt. Um eine Farbe mittels eines Farbvektors (bestehend aus 3 **byte**-Werten) allokieren zu können, wurde zunächst eine 3-dimensionale Textur entwickelt, welche die einzelnen Spektren als **byte**-Werte speichert. Eine Abbildung der Textur ist in 19 zu sehen. So ist man in

Abbildung 19: Speicherung der Spektrumdaten in einer Textur

der Lage theoretisch 2^{32} verschiedene Spektren mit einem Farbvektor zu adressieren. Diese Repräsentation ist gut geeignet, da dadurch jedem Objekt mittels einer Farbe oder einem `VertexAttribute`[20] ein eindeutiges Spektrum zugewiesen ist. Diese konzeptionelle Idee lässt sich in der aktuellen Implementation einfach umsetzen und wurde auf Grund der noch geringen Datenmenge an Spektren verkleinert. Diese Einschränkung hat im Bezug auf die Erweiterungen der in dieser Arbeit vorgestellten Methode weitere Vorteile, welche in Kapitel 5 erläutert werden. Zu einfachen Debugzwecken wird die Auswahl des Spektrums noch durch eine globale Variable bestimmt, kann aber jederzeit in die Farbe des Objektes codiert werden.

Speichert man in jedem Farbkanal der Textur nun eine spezifische Intensität für eine Wellenlänge, erhält man als resultierende Breite der Textur 128 Texel, da pro Texel 4 Kanäle zur Verfügung stehen. Durch Verwendung eines Offsets kann nach einem abgeschlossenem Spektrum ein weiteres Spektrum anghängt werden. Da die Größe mit 512 Elementen entsprechend gewählt worden ist, schließt jede Textur auch im Alpha-Kanal ab ($512 \mod 4 = 0$), d.h. das Auslesen eines Texels kann nicht zu einer Überschneidung des nächsten Spektrums führen. Alternativ ist es möglich, die Intensitäten auf der kompletten Breite der Textur abzuspeichern. Die zusätzlichen Kanäle können für weitere Spektren oder für andere Daten genutzt werden (Siehe Kaptiel 5).

Jede Wellenlänge ist durch einen entsprechenden TextureLookup auslesbar und eindeutig durch die Position bestimmt, wenn man von einem statischen Wellenlängenbereich ausgeht. Auch bleibt die Möglichkeit erhalten, diese Daten für den 3-dimensionalen Raum zu erweitern, da 3-dimensionale Texturen aus mehreren 2-dimensionalen bestehen. Somit bildet die 2-dimensionale Variante eine Vereinfachung zum ursprünglichen Modell.

Für die Berechnung der resultierenden XYZ-Werte, welche dann in das RGB-Farbmodell transformiert werden können, werden die einzelnen CMF-Werte für jede Wellenlänge in der Berechnung benötigt. Da es sich dabei um drei Funktionen handelt, empfiehlt sich eine Speicherung in einer Textur wie folgt:

$$R_{\text{texel}}(\lambda) = \bar{x}(\lambda)$$
$$G_{\text{texel}}(\lambda) = \bar{y}(\lambda)$$
$$B_{\text{texel}}(\lambda) = \bar{z}(\lambda)$$

Für die Erstellung einer entsprechenden Textur, wurde ein Konvertierungsprogramm geschrieben (`CIEXYZ2Texture`), welches die Rohdaten aus [SS06] in einer RGB-Textur speichert, wie in Abbildung 20 dargestellt.

[20]Zusätzliche Attribute eines Vertex können dadurch definiert werden, wie z.B. Tangente oder Binormale

Abbildung 20: In einer Textur gespeicherte *Color Matching Function*-Werte

Unterschiedliche Daten zum Auslesen der *Color Matching Functions* Werte		
byte-Textur	**float**-Textur	Array
744	407	581

Tabelle 2: Vergleich der Frameraten durch Genauigkeit der CMF-Werte

Da für die Speicherung in einem Bildformat nur die **byte**-Repräsentation zur Verfügung steht, werden die CMF-Werte durch das Maximum dividiert. Das Ergebnis zwischen 0 und 1 wird anschliessend in den **byte**-Bereich zwischen 0 und 255 transformiert. Dies kann in dem Programm global oder lokal geschehen, wobei durch eine lokale Ausführung eine größere Genauigkeit entsteht, jedoch müssen dann die Werte individuell zurückgerechnet werden. Bei der globalen Skalierung wird durch den Peak von $\bar{z}$ dividiert, da dieser die Blauwahrnehmung wiedergibt und die größte Intensität besitzt. Für eine genauere Auswertung wurde das **FTXT** Format definiert, welches in Kapitel 4.1 erläutert wird. Dadurch ist es möglich, **float**-Werte direkt in einer Textur zu speichern, zur Erhöhung der Genauigkeit, jedoch die Geschwindigkeit verringert. Ein Vergleich der Frameraten wird in Tabelle 2 aufgeführt, wobei der Wellenlängenbereich [450nm − 650nm] gewählt wurde und das Objekt in Ausgangsgröße, bei einer Bildgröße von 800×600, angezeigt wird.

Die Implementierung hängt hierbei stark von den aktuellen Profilen der neueren Graphikkarten-Generation ab, welche es erlauben, ungefilterte Zugriffe auf einzelne Texel durchzuführen. Dadurch ist es möglich, die Daten einer spezifischen Wellenlänge direkt auszulesen.

Verrechnet man die einzelnen Wellenlängen mit ihrer zugewiesenen Intensität und dem Tristimulus-Wert, so erhält man einen XYZ-Farbwert. Dieser Farbwert muss nun in den RGB-Farbraum konvertiert werden, da die Ausgabe von `OpenGL` und des Bildschirmes nur RGB-Farben wiedergibt. Natürlich wäre es auch möglich, XYZ-Werte auszugeben, nur würden diese nicht der korrekten Farbe entsprechen, da die Wiedergabe von einem Monitor durch RGB-Werte definiert ist. Für die Transformation wird eine lineare Transformationsmatrix genutzt, so dass die XYZ-Werte in RGB-Werte umgerechnet werden. Dabei kann es passieren, dass Farben sich nicht in dem Gamut befinden und müssen entweder geclippt, oder skaliert werden. Im folgenden werden alle Werte auf den Wertebereich von $[0, 1]$ geclippt. Dies verhindert eine Farbverschiebung, wobei aber einzelne Farben eventuell nicht korrekt angezeigt werden.

Die Berechnung wird dabei pixelbasiert ablaufen, d.h. es muss ein `Fragment`-Shader entworfen werden, da für die Benutzung der bestehenden Beleuchtungmodelle und für eine höhere

Genauigkeit die Berechnung im Fragment-Bereich durchgeführt werden sollte. Auch kann die Anzahl der Texturzugriffe im Vertex-Shader stark eingeschränkt sein. Eine Berechnung von z.B. Interferenzeffekten auf einem Objekt mit grober Tessellierung kann nur im Fragment-Shader mit genügender Genauigkeit erfolgen. Man stelle sich hierzu ein Viereck vor, das den ganzen Bildschirm ausfüllt und nur durch die Eckpunkte definiert wird. Ein Viereck wird normalerweise in zwei Dreiecke zerlegt und die Eckpunkte werden dann entsprechend beleuchtet. Möchte man nun eine Interferenz in der Mitte dieses Viereckes erzeugen, so wird es, auf Grund der weit entfernten Eckpunkte, zu keinem brauchbaren Resultat kommen, da die Farbe eines Fragmentes aus den Farben der anliegenden Vertices interpoliert wird.

Nutzt man die vorhandenen Grundlagen, so kann ein einfaches Modell implementiert werden, welches mit einem einfachen *Pass-Thru*-Vertex-Shader und einem entsprechenden Fragment-Shader funktioniert. Hierbei werden einzelne Funktionen für das Auslesen der Tristimulus-Werte, die Transformation in den RGB-Farbraum mittels der linearen Transformationsmatrix und die Bestimmung der einzelnen Intensitäten für jede Wellenlänge benötigt. Mit diesem Basismodell können Effekte, welche die Intensitäten einzelner Wellenlängen berücksichtigt, wie Interferenz, Diffraktion oder eine Verschiebung, eingebunden werden. Diese greifen dabei auf das vorhandene Grundsystem zurück, um eine resultierende Farbe zu ermitteln. Ein Pseudocode für die Ausführung ist in Quellcode 3 wiedergegeben.

```
1 float3 xyzValues;
2 for( all Wavelengths lambda)
3 {
4     float intensity = getIntensity(lambda);
5     xyzValues += intensity * CMF(lambda);
6 }
7 float3 finalColor = ConvertToRGB(xyzValues);
8 return finalColor;
```

Code 3: Pseudo Code für die grundlegende Wellenlängen Implementation

4.3 Implementation des Grundmodells

Die erste Implementation des Grundmodells sollte eine Beleuchtungsrechnung in einem möglichst einfachen Rahmen bieten. Dafür wurde zunächst die Einschränkung vorgenommen, die Intensitäten der Lichtquelle über das gesamte Spektrum als 1 zu definieren. Dies bedeutet, dass nur die Objektfarbe, welche ebenfalls durch Wellenlängen bestimmt sein muss, die resultierende Farbe bestimmt. Die einzelnen Intensitäten aus Licht und Objekt können durch Multiplikation zu der resultierenden Intensität zusammengefasst werden und man erhält für die Bestimmung der Intensitäten die Gleichung (56). In diesem vereinfachten Fall erhält man

das Objektspektrum als Resultat. Da die Multiplikation kommutativ ist, könnte es sich ebenfalls um ein „farbiges“ Licht und ein weißes Objekt handeln.

$$\text{intensity}_{\text{final}} = \text{intensity}_{\text{light}} \cdot \text{intensity}_{\text{object}} \tag{56}$$

Als weitere Simplifikation wurde die Intensität in einem Bereich auf einen konstanten Wert größer 0 gesetzt, ausserhalb dieses Bereichs auf 0. Somit sind die Intensitäten nur in diesem Bereich für die Beleuchtungsrechnung und die Bestimmung der XYZ-Werte relevant.

Das Auslesen der Daten aus der CMF-Textur geschieht abhängig von der Wellenlänge. Eine entsprechende Funktion, welche eine konstante Verschiebung der λ-Werte vornimmt, wurde implementiert und ist in Code 4 zu sehen.

```
1 float4 getCMF(int lambda)
2 {
3     return tex2Dfetch(XYZ, int4((lambda-360),0,0,0));
4 }
```

Code 4: Funktion zum Auslesen der CMF Werte

Bei der Funktion `getCMF()` wurde auf das Abfangen von Eingaben ausserhalb des gültigen Bereiches abgesehen, da ein großer Geschwindigkeitsverlust daraus resultiert. Da diese Funktion für jedes Pixel und innerhalb dieses Pixels im Worstcase 512 mal aufgerufen wird, liefern schon kleinere Optimierungen einen großen Effekt auf die Geschwindigkeit eines Programmes. Dementsprechend wurden die Funktionsaufrufe so geschrieben, dass keine falschen Daten geliefert werden. Für den Fall, dass ein Wert ausserhalb des definierten Bereiches aufgerufen wird, nutzt die `tex2Dfetch` aus, dass der genutzte Sampler für die CMF-Werte an den Rändern durch **ClampToEdge** definiert ist. Dabei ist die Textur nur in ihrem Bereich bestimmt, Werte ausserhalb des Bereiches nehmen den Wert des letzten gültigen Texels an. Im Gegensatz zu **Clamp** wird hier nicht mit der Randfarbe der Textur interpoliert, was vor allem bei dem Zusammensetzen mehrerer Texturen von Nutzen ist.

Für die Addition der einzelnen XYZ-Werte wird eine **half3**-Variable definiert, welche die einzelnen Werte von CMF mit den resultierenden Intensitäten multipliziert. In jeder Iteration, also für jede Wellenlänge, werden die einzelnen XYZ-Werte aufaddiert, so dass am Ende der Iterationen das Integral über die einzelnen Wellenlängen gebildet wurde. Skaliert man nun die resultierenden XYZ-Werte mit einem konstanten Faktor k, so erhält man normierte Werte, welche in den RGB-Farbraum transformiert werden können.

Dieser Skalierungsfaktor k wurde von [DK06] adaptiert und resultiert aus dem Integral über die $\bar{y}$-Kurve der CMF-Werte. Zu beachten ist, dass die Werte aus der Textur skaliert sind und sich dadurch ein anderer Wert ergibt, als wenn man das Integral mit den originalen Werten

bildet. Dieser Skalierungsfaktor ist konstant und wird dementsprechend global definiert, so dass man ihn durch den *ShaderViewer* editieren kann. Er wird im folgenden für die skalierte Textur auf

$$k = 0.0176$$

gesetzt. Da die Berechnung der Intensitäten nur für den Bereich interessant ist, in dem diese ungleich 0 sind, kann die Schleife entsprechend eingeschränkt und somit eine Beschleunigung des Verfahrens erzielt werden, falls nicht das gesamte Spektrum eine Intensität besitzen soll.

Für die Berechnung der XYZ-Werte wird die in Code 5 gegebene Funktion genutzt. Die Funktion `calculateXYZValuesInBounds` führt dabei keine Skalierung der Werte durch und berechnet die Koordinaten mit Hilfe der übergebenen Intensität.

```
1  half3 calculateXYZValueInBounds(half illumination)
2  {
3      half3 xyzValues = half3(0,0,0);
4
5      for(int i=lower;i<=upper;i=i+steps)
6      {
7          xyzValues += illumination*getCMF(i);
8      }
9
10     return xyzValues;
11 }
```

Code 5: Berechnung der XYZ-Koordinaten in festen Grenzen

Der global definierte Wert **steps** gibt an, wieviele Lambdas übersprungen werden sollen. Dies ist eine simple Methode für ein LOD-System, versagt aber, falls komplexere Berechungen durchgeführt werden müssen.

Durch die Funktion `calulateXYZValuesInBounds` ist es möglich, die normalen Beleuchtungsmodelle zu implementieren, da diese sowohl für komplette Farben, als auch für einzelne Intensitäten gelten, denn die Berechnung an einem Pixel bleibt konstant. Man erhält somit einen allgemeinen Fragment-Shader (Code 6), welcher die Farbberechnung wellenlängenbasiert durchführt.

```
1  float3 fragmentSimpleIntensity() : COLOR
2  {
3      half3 xyzValues = calculateXYZValueInBounds(1.0);
4      xyzValues *= k*steps;
5
6      return XYZToRGB(scaling*xyzValues);
7  }
```

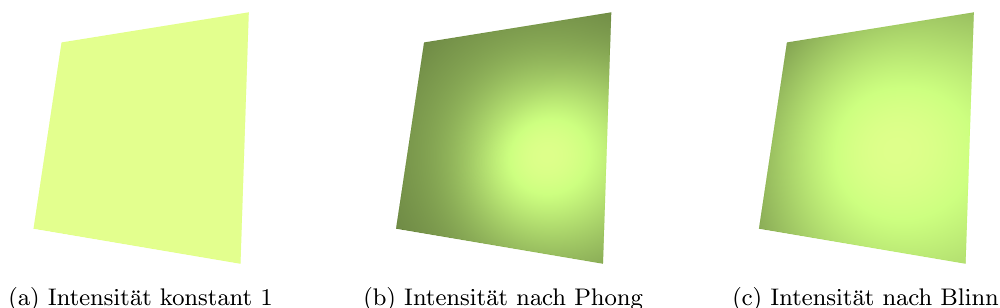
Code 6: Fragment Shader zur einfachen wellenlängenbasierten Beleuchtungsrechnung

Die resultierenden XYZ-Werte müssen ggf. mit dem **steps** Parameter multipliziert werden, damit die Fläche zwischen 2 Stützpunkten korrekt wird. Berechnet man nun die Beleuchtung nach Phong bzw. Phong-Blinn, oder einem anderem Modell, so kann das Resultat für die Berechnung der einzelnen Intensitäten genutzt werden. Die Berechnungen wurden dafür jeweils in entsprechenden Funktionen zusammengefasst - als Beispiel dient das Phong-Beleuchtungsmodell in Quellcode 7. Die benutzten Funktionen für die Berechnung sind Code 8 zu entnehmen. Die Ausgabe der einzelnen Funktionen in Kombination mit einem *Pass-Thru*-Vertex-Shader sind in Abbildungen 21a bis 21c dargestellt. Der *Pass-Thru*-Vertex-Shader bestimmt dabei die Positionen der einzelnen Vertices und transformiert die Vertex-Attribute passend. Ebenfalls werden die Reflexionsvektoren, sowie Vektor zur Kamera (**eyevec**) und Vektor zum Licht bestimmt, welche im Fragment-Shader entsprechend interpoliert werden. Als Wellenlängen-Bereich für die Beleuchtung wurde [450nm – 650nm] gewählt.

```
1 half calculatePhong(float3 normal, float3 lightvec,
2                     float3 reflectionvec, float3 eyevec)
3 {
4     half lambert = calculateLambert(normal,lightvec);
5     half spec = calculateSpecular(reflectionvec,eyevec);
6
7     return lambert+spec;
8 }
```

Code 7: Bestimmung der Intensität in einem Pixel nach Phong

Die einzelnenen Komponenten des Beleuchtungsmodells werden durch individuelle Funktionen berechnet, so dass eine gute Kapselung erreicht wird. Eventuelle Optimierungen im Quellcode können so sofort auf alle Funktionen Einfluss nehmen.



(a) Intensität konstant 1 (b) Intensität nach Phong (c) Intensität nach Blinn

Abbildung 21: Verschiedene Beleuchtungen des selben Objektes in dem Wellenlängenmodell. Der Exponent bei Blinn und Phong ist gleich, $\lambda \in [450\text{nm} - 650\text{nm}]$, Lichtspektrum 38a

In Abbildung 22 ist die Auswirkung des Parameters **steps** zu sehen. In den kleinen Bereichen ist kaum ein Unterschied festzustellen. Der Fehler der Farbe steigt mit zunehmender

```
half calculateLambert(float3 normal, float3 lightvec)
{
    half dotnl;

    normal = normalize(normal);
    lightvec = normalize(lightvec);

    dotnl = dot(normal,lightvec);
    return (dotnl <= 0) ? 0 : kd*dotnl;
}
half calculateSpecular(float3 reflectionvec, float3 eyevec)
{
    half dotrv;

    reflectionvec = normalize(reflectionvec);
    eyevec = normalize(-eyevec);

    dotrv = dot(reflectionvec,eyevec);
    return (dotrv <= 0) ? 0 : ks*pow(dotrv,power);
}
```

Code 8: Die benutzten Funktionen für die Beleuchtung

Frameraten der einzelnen einfachen Verfahren				
Bildgröße	Unoptimiert, Intensität **const.**	Intensität **const.**	Phong	Blinn
400×300	1527	2587	1910	1840
640×480	772	1035	1163	1141
800×600	549	745	736	814
1024×768	353	486	508	477

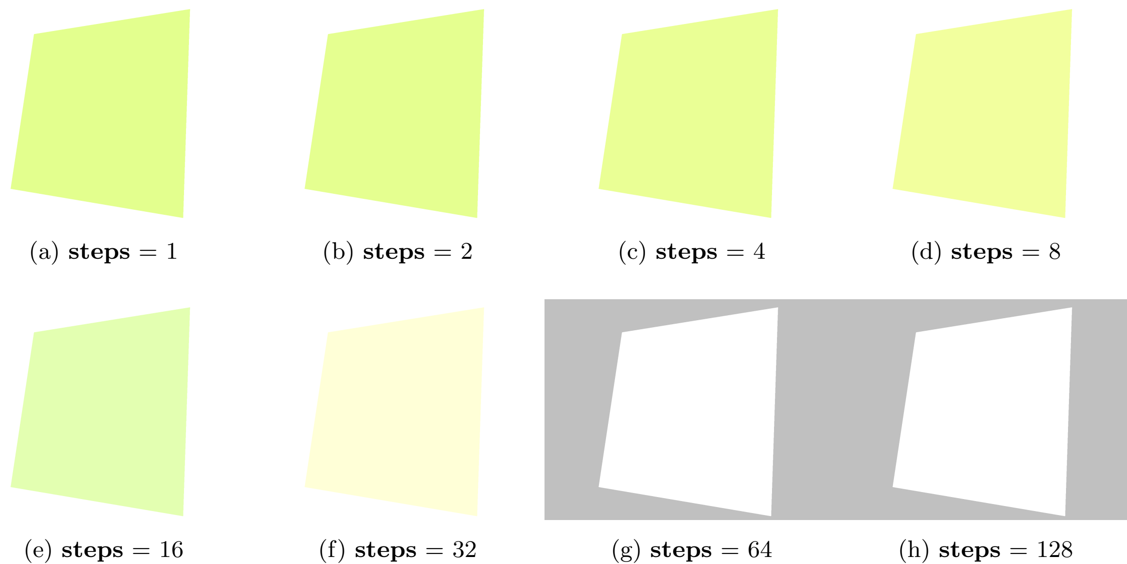
Tabelle 3: Frameraten in FPS für die einfachen Modelle

Größe der Schrittweite sehr schnell an und führt zu unbrauchbaren Ergebnissen. In den Beispielen wurden nur Werte zur Basis 2 gewählt, die Größe des Parameters ist jedoch nicht auf solche Werte beschränkt.

Die einfachen Methoden zeigen schon, dass moderne Graphikkarten selbst eine große Anzahl von Berechnungen effizient durchführen können. Es ist davon auszugehen, dass der Compiler von `CG` die Schleifen „ausrollen“ (engl: *unroll*) wird, also die einzelnen Befehle sequentiell in Code umgewandelt und ausgeführt werden. Eine Messung der Frameraten (in *Frames Per Second*, kurz FPS) ist in Tabelle 3 aufgeführt. Das Objekt wird dabei unmodifiziert[21] betrachtet. Der Wellenlängenbereich ist $\lambda \in [360\text{nm} - 830\text{nm}]$[22].

[21] in der Ausgangsposition der Szene

[22] Entspricht dem kompletten CMF-Wertebereich



(a) **steps** = 1 (b) **steps** = 2 (c) **steps** = 4 (d) **steps** = 8

(e) **steps** = 16 (f) **steps** = 32 (g) **steps** = 64 (h) **steps** = 128

Abbildung 22: Auswirkungen auf die Farbwiedergabe mit dem Parameter **steps**. Wellenlängen im Bereich [450nm – 650nm], Lichtspektrum 38a. Die letzten 2 Bilder (**steps** = 64 und **steps** = 128) wurden Grau hinterlegt, damit das Objekt sichtbar bleibt.

In Abbildung 23 wurde versucht, innerhalb des RGB-Modells die berechnete Farbe nachzustellen. Große Probleme entstanden dabei durch die Highlight-Farbe, welche nur in Näherung gefunden wurde. Ändert sich eine einzelne Intensität in dem Wellenlängenmodell, oder wird der Bereich verschoben, so sind die Ergebnisse unbrauchbar und die RGB-Farbe muss umständlich wiederhergestellt werden.

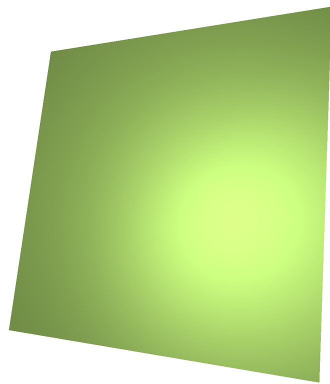
Im nächsten Schritt wurden variable Intensitäten für Licht und Objekt eingeführt. Mit Hilfe des *SpectralLibrarian* wurden einzelne Spektren definiert. Die Exportfunktion des Programms schreibt eine Textur, die nach dem definierten 2-dimensionalen Format (Siehe Kapitel 4.2) aufgebaut ist. Da einzelne Intensitäten auf die RGBA-Farbkanäle der Textur verteilt sind, musste eine entsprechende Ausleseroutine entwickelt werden.

Dabei gibt es 2 verschiedene Implementationen:

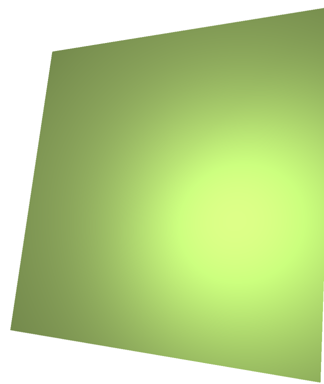
1. Auslesen erfolgt kanalweise, jedes Texel wird 4 mal bearbeitet. Vorteil ist, dass die vorhandene Routine für die Beleuchtungsrechnung nur minimal angepasst werden muss (Einfügen des Funktionsaufrufes zum Auslesen der Intensität).

2. Auslesen eines **float4** Wertes, welcher alle 4 Kanäle, und damit 4 aufeinanderfolgende Intensitäten beinhaltet. Vorteil ist, dass weniger Texturzugriffe benötigt werden.

Die Implementation der Funktion zum Auslesen einzelner Wellenlängen ist in Code 9



(a) Wellenlängen Modell



(b) RGB-Modell

Abbildung 23: Vergleich zwischen Phong Beleuchtung im wellenlängenbasierten und im RGB-Modell. Wellenlängen im Bereich [450nm – 650nm]. Lichtspektrum: 38a

gegeben. Die Funktion für Objekte besitzt die gleiche Grundlage, nur der **sampler** zum Auslesen muss angepasst werden.

```
float getLightIntensity(int lambda)
{
    int diff,texpos;
    float4 tmp;

    if(lambda < 324 || lambda > 836) return 0;

    texpos = lambda-324;
    diff = (texpos)/4;

    tmp = tex2Dfetch(LightColl,
        int4(diff+128*LightTexture.x,LightTexture.y,0,0));

    texpos = texpos % 4;
    if(texpos == 0) return tmp.r;
    else if(texpos == 1) return tmp.g;
    else if(texpos == 2) return tmp.b;
    else return tmp.a;
}
```

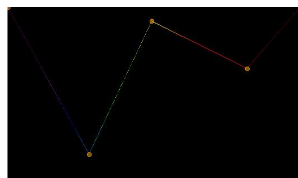
Code 9: Auslesen einzelner Intensitäten aus einer Textur

Man sieht sofort, dass eine große Menge von Daten nicht benutzt werden, da diese nicht zurückgegeben werden. Der alternative Ansatz zum Auslesen der Daten garantiert, dass keine unnötigen Daten eingelesen werden. Jedoch muss die Hauptiterationsschleife anders geschrieben werden, damit 4 Wellenlängen bearbeitet werden können. Hierfür wurde eine Funktion `getCMFCombined` zum Auslesen von 4 CMF-Werten programmiert, die 4-mal die Funktion `getCMF` aufruft und die Werte als 4×4 Matrix zurückgibt. Matrizen können in `CG` als Arrays von Vektoren aufgefasst werden, so dass die einzelnen Zeilen einfach durch einen entsprechenden Index addressiert werden können.

Die kombinierte Version vereinfacht dadurch die Funktion zum Auslesen der Intensitäten. Diese ist in Code 10 beispielhaft für die Lichtintensitäten aufgeführt, die sich nur durch den **sampler** von der Variante für die Objekte unterscheidet.

```
1 float4 getLightIntensityCombined(int xPos)
2 {
3     return tex2Dfetch(LightColl,
4         int4(xPos+128*LightTexture.x, LightTexture.y,0,0));
5 }
```

Code 10: Kombiniertes Auslesen der Intensitäten aus einer Textur



(a)

(b)

Abbildung 24: (a) Spektrum 39b, (b) Ausschnitt der benutzten Textur der Intensitäten

Bei der Erstellung der Schleife, welche alle Wellenlängen λ abdeckt, wurde auf eine native Darstellung verzichtet. Der Shift, welcher standardmäßig ausgeführt werden muss, wird herausgerechnet und die Werte durch 4 dividiert. So erhält man wieder eine Schrittweite von 1. Somit erechnet sich die erste Wellenlänge einer Iteration wie folgt:

$$\lambda_{1,\mathrm{i}} = 324 + 4 * i \tag{57}$$

Der Shift beträgt hier 324, da das in Kapitel 4.2 erweiterte sichtbare Spektrum verwendet wird, welches einen Definitionsbereich von [324nm − 836nm] besitzt. Die folgenden Lambdas sind durch einfache Inkrementation bestimmbar. Liest man nun die Intensitäten aus und verrechnet sie durch Multiplikation, erhält man den in 11 angegebenen Code. Dabei wurden die Intensitäten mit einem Skalierungsfaktor von 2 versehen, damit die Helligkeit größer wird und die Objekte besser zu sehen sind. Dies führt allerdings zu einer Farbverschiebung.

```
float3 fragmentObjectLightTestWithPhong(vOut Input) : COLOR
{
    half3 xyzObj = {0,0,0};
    half3 xyzLight = {0,0,0};
    half4x4 cmf;
    half4 intensity, intensityObject, intensityLight;

    half illu = calculateLambert(Input.normal,Input.lightvec);
    half spec = calculateSpecular(Input.reflectvec,Input.eyevec);

    for(int i=9;i<128;i++)
    {
        intensityObject = getObjectIntensityCombined(i);
        intensityLight = getLightIntensityCombined(i);

        intensity = intensityLight*intensityObject*2;
        cmf = getCMFCombined(324+4*i);

        xyzObj += intensity.r*cmf[0] + intensity.g*cmf[1]
            + intensity.b*cmf[2] + intensity.a*cmf[3];

        xyzLight += intensityLight.r*cmf[0]
            + intensityLight.g*cmf[1]
            + intensity.b*cmf[2]
            + intensityLight.a*cmf[3];
    }
    xyzObj *= k;
    xyzLight *= k;

    if(MetalObject)
        return XYZToRGB(xyzObj*illu)+XYZToRGB(xyzLight*spec);
    else
        return XYZToRGB(xyzObj*(illu+spec));
}
```

Code 11: Beleuchtungsrechnung mit individuellen Intensitäten

Die Ergebnisse sind in Abbildungen 25a bis 25c zu sehen. Ein Beispiel einer Textur sowie eines enthaltenen Spektrums sind in Abbildung 24 wiedergegeben. Die Performanzmessungen wurden mit identischen Werten für die Objekt- und Ausgabegröße durchgeführt, so dass die Werte vergleichbar sind. Die Frameraten befinden sich in Tabelle 4.

In Abbildung 25a wird anstelle einer festen Intensität eine entsprechende aus einer Textur gelesen. Die Abbildungen 25b und 25c zeigen den Fall, dass farbiges Licht auf ein farbiges Objekt fällt. Hierbei ist zu beachten, dass es sich um eine subtraktive Farbmischung handelt und die einzelnen Intensitäten sich multiplizieren. Dies wird klar, wenn man ein rotes Objekt mit einem weißem Licht bestrahlt. Im Lambertmodell kann die resultierende Farbe nur eine rote Farbe sein.

Frameraten der Verfahren			
Bildgröße	`Single`	`Combined`	Kombiniert mit Phong
400×300	725	1776	1120
640×480	349	942	575
800×600	243	651	398
1024×768	154	409	254

Tabelle 4: Frameraten in FPS für die Modelle mit variablen Intensitäten. `Single` und `Combined` wurden nur mit einem Texturefetch verglichen.

Die Frameraten zeigen, dass das verwendete Modell eine gute Performanz erreicht. Der Ansatz, Spektren zur Definition der Farbe zu verwenden, kann daher sinnvoll in Anwendungen zum Einsatz gebracht werden. Die Wellenbasierte Berechnung erlaubt aber noch weitere Effekte, welche im nächsten Kapitel erläutert werden.

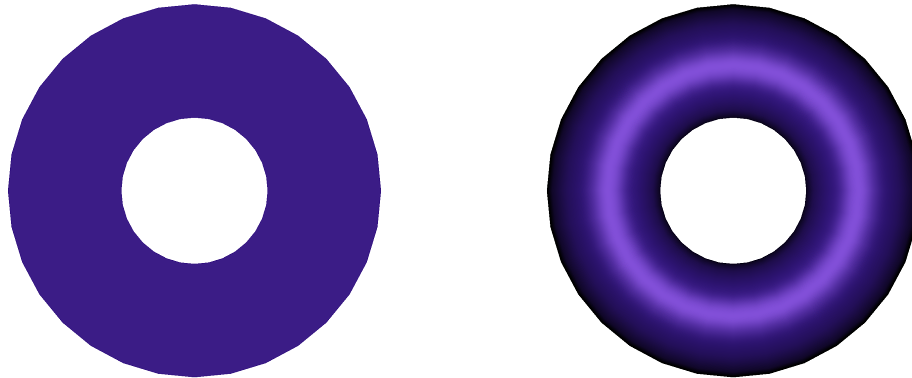
4.4 Implementation von Interferenzeffekten

Das Modell ist bis jetzt in der Lage, einzelne Intensitäten zu bestimmen und eine Farbmischung dieser durchzuführen. Da einzelne Wellenlängen vorhanden sind können Interferenzeffekte modelliert werden. Als Beispiels-Szenario werden hier dünne Filme gewählt, da sie ohne großen Raytracing-Anteil zu realisieren sind. Für andere Fälle, wie die Diffraktion, muss dies nicht immer zutreffen, da sich z.B. in einem Prisma die einzelnen Farben bis jetzt nur durch Raytracing bestimmen lassen.

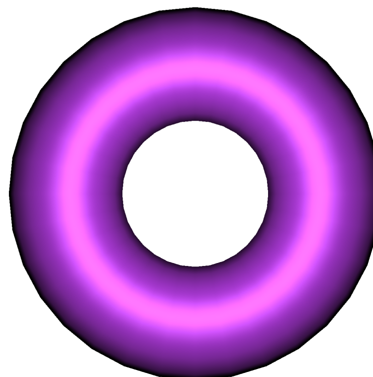
Da es sich bei der Interferenz eines dünnen Films um einen Effekt handelt, der die Lichtstrahlen direkt zurückwirft, lassen sich die Berechnungen für die einzelnen Interferenzfarben fragmentbasiert durchführen. Detailierte Informationen über benachbarte Sichstrahlen oder Geometrien werden nicht benötigt. Die reflektierten Strahlen, welche zum Auge gelangen, werden dabei rückwärts betrachtet. Dies bedeutet, bei der Berechnung der Interferenz wird der direkte Einfluss bestimmt, sowie die Strahlen, die eventuell mehrmals das Material durchlaufen. Durch den Weglängenunterschied kann man so für alle Punkte und Wellenlängen die Änderung der Intensität bestimmen. Dies resultiert dann in einer lokalen Auslöschung einzelner Wellen, was zu einem spezifischem Interferenzmuster führt.

Die erste Modellierung basiert dabei noch nicht auf den von Fresnel aufgestellten Gleichungen (Siehe 2.2.5), sondern nutzt eine direkte Berechnung, um den Unterschied einzelner Wellenlängen zu bestimmen. Dafür wird das Snellius'sche Brechungsgesetz (12) verwendet, sowie die Tatsache, dass sich Wellenlängen mit einem Phasenunterschied von $\frac{\pi}{2}$ auslöschen.

Die Berechnung der Brechung nach Snell ist dabei in verschiedenen Varianten implemen-



(a) Intensität aus 1 Textur. Spektrum 38b (b) 2 Texturen für Intensitäten. Spektren 38b,38c



(c) Spektren 38b,38d

Abbildung 25: Beleuchtung mit variablen Intensitäten

tiert worden. Die hauptsächlich benutzte Variante basiert dabei auf der Vektorrechnung und berücksichtigt, dass Sinus und Cosinus ineinander umgerechnet werden können. Der Cosinus des Winkels α zwischen Licht und Normalen ergibt sich mit dem Skalarprodukt der beiden Vektoren:

$$\cos(\alpha) = \vec{N} \odot \vec{L}$$

Nach Gesetzmäßigkeiten der Trigonometrie gilt:

$$\sin^2 + \cos^2 = 1$$

Der Sinus und der Cosinus des gebrochenen Winkels sind damit:

$$\sin(\alpha_r) = \sqrt{\left(\frac{n_1}{n_2}\right) * (1 - \cos(\alpha)^2)} \tag{58}$$

$$\cos(\alpha_r) = \sqrt{1 - \left(\frac{n_1}{n_2}\right) * (1 - \cos(\alpha)^2)} \tag{59}$$

Diese Gleichungen führen zu der Implementation, wie sie in Code 12 wiedergegeben ist.

```
1  half2 snell_cosAnglev(half dotnv, half n1, half n2)
2  {
3      half2 combi;
4      half sin2;
5
6      if(n1 == n2)
7          return dotnv;
8
9      combi = pow(half2((n1/n2),dotnv),2);
10     sin2 = combi.x*(1-combi.y);
11
12     return sqrt(half2(sin2,1-sin2));
13 }
```

Code 12: Vektorbasierte Berechnung des Brechungswinkels

Die Weglänge in der dünnen Schicht lässt sich einfach durch folgende Überlegung bestimmen: Dringt der Strahl in die dünne Schicht ein, so wird der Strahl nach dem Snellius'schen Gesetz gebrochen und legt den Weg l bis zum Ende der dünnen Schicht zurück. An dieser wird er reflektiert und legt den selben Weg l nochmals zurück. Der Weglängenunterschied ist somit:

$$\Delta \text{Weglänge} = 2 \cdot l$$

Der Brechungsindex gibt an, um welchen Faktor sich die Weglänge in dem Medium vergrößert (16). Somit berechnet sich der Weg wie folgt:

$$l = 2 \cdot \cos(\alpha) \cdot n_2 \cdot d$$

Dabei sind die Variablen wie folgt definiert:

- α: Der gebrochene Winkel nach Snell
- d: Die Dicke der Schicht
- n_2: Der Brechungsindex der dünnen Schicht

Rechnet man nun noch den Einfluss der Position des Auges mit ein (Winkel: θ_e), ergibt sich folgende Formel:

$$\text{difference} = 2 \cdot d \cdot n_2 \cdot \cos(\alpha) \cdot \cos(\theta_e) \tag{60}$$

Zur Berechnung der Intensität in Abhängigkeit von der Wellenlänge ergibt sich:

$$I = \frac{(2 \cdot d \cdot n_2 \cdot \cos(\alpha) \cdot \cos(\theta_e)) \mod \lambda}{\lambda} \tag{61}$$

Da sich die einzelnen Cosinus der Winkel in der Vektorrechnung durch ein Skalarprodukt ausdrücken lassen und hier immer die Winkel zu der Normalen der Oberfläche gemeint ist, kann die letzte Gleichung entsprechend umgestellt werden:

$$I = \frac{(2 \cdot d \cdot n_2 \cdot \vec{L} \odot \vec{N} \cdot \vec{V} \odot \vec{N} \mod \lambda)}{\lambda} \tag{62}$$

Benutzt man die Gleichung (62), so kann ein Fragment-Shader geschrieben werden, welcher als Basis die Funktion `calculateXYZValueInBounds` (Siehe Code 5) zur Berechnung der Interferenz benutzt. Der Quellcode für einen solchen Shader ist in Code 13 wiedergegeben. Die verwendete Funktion `PathDifferenceInterference` berechnet I mit der Gleichung (62) in Abhängigkeit von der aktuellen Wellenlänge und den Cosinus von Licht- und Kamera-Vektoren zur Normalen.

```
1  float3 fragmentThinFilm(vOut Input) : COLOR
2  {
3      //Initialization
4      ...
5
6      //Calculate angles
7      half angle = acos(dot(lightvec,normal));
8      half cosBeta = cos(snell_anglef(angle,nfactor0,nfactor1));
9      half cosEye = dot(eyevec,normal)*0.1+0.9;
10
11     for(int i=lower;i<=upper;i+=steps)
12     {
13         intensity = PathDifferenceInterference(cosBeta,cosEye,
14             i,(half) thickness)*illu;
15
16         xyzValues += intensity*getCMF(i);
17     }
18     xyzValues *=k*steps;
19
20     //Return RGB-value
21     ...
22 }
```

Code 13: Einfache Berechnung von Interferenz

Das von der Funktion `fragmentThinFilm` generierte Bild ist in Abbildung 26 zu sehen. Die erreichten Frameraten sind in Tabelle 5 aufgeführt.

Da dieses Modell aber eine starke Vereinfachung benutzt, müssen noch exaktere Betrachungen durchgeführt werden, um ein physikalisch korrekteres Modell zu erhalten. Die Fresnel-Gleichungen liefern hierbei einen entscheidenen Beitrag, da das Licht abhängig von seiner Polarisierung und des Einfallwinkels unterschiedlich transmittiert bzw. reflektiert. Dadurch wird

die Intensität der betroffenen Wellenlänge verändert. Man erhält damit ein feineres Modell, als jenes, das mit Gleichung (62) durch Modulo-Rechnung erreicht wird. Die Fresnel-Gleichungen sind dabei abhängig von dem Brechungsindex der jeweiligen Schicht, aber nicht von der expliziten Wellenlänge. Geht man von konstanten Brechungsindizes aus, so kann die Berechnung wellenlängenunabhängig stattfinden und so Prozessorleistung eingespart werden. Die Berechnung der Gleichungen wurde in entsprechenden Funktionen codiert, zusätzlich wurde eine Variante für das kombinierte Auslesen implementiert und ist Code 14 zu entnehmen. Diese Funktion wird am meisten genutzt, da sowohl Perpendicular- als auch Parallelanteil der Intensität bestimmt werden müssen, um die korrekten Werte zu erhalten.

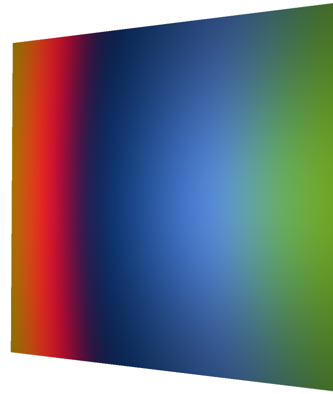


Abbildung 26: Interferenz mit der Weglängenmethode. Variation der Dicke mit den Texturkoordinaten. Ausgangsdicke 136nm, Enddicke 363nm. Brechungsindizes: $n_1 = 1.0, n_2 = 2.0, n_3 = 1.5$. Lichtspektrum: 38a

```
1 half2 fresnelReflectanceCombined(half cosThi, half cosTht,
2         half n1, half n2)
3 {
4     half nti1 = n2*cosThi;
5     half ntt1 = n1*cosTht;
6     half nti2 = n1*cosThi;
7     half ntt2 = n2*cosTht;
8     return half2((nti1-ntt1)/(nti1+ntt1),(nti2-ntt2)/(nti2+ntt2));
9 }
```

Code 14: Funktion zur Bestimmung des Reflexionsfaktors für beide Polarisierungen

Der Weglängenunterschied kann noch einfacher definiert und unter Benutzung der trigonometrischen Funktionen bestimmt werden. Man erhält damit folgende Gleichung [Hea91]:

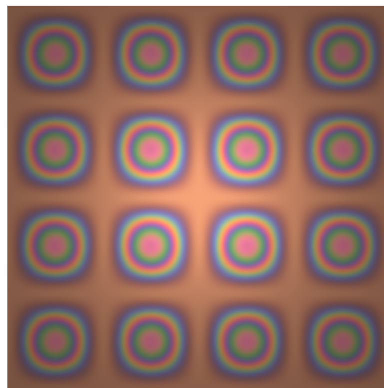
$$\Delta = 2 \cdot n \cdot d \cdot \cos\theta_i \tag{63}$$

Dieser Weglängenunterschied wird genutzt, um einen sogenannten Phasenwinkel [DK06, SW08] zu definieren, welcher für die Berechnung der resultierenden Intensität benötigt wird. Mit der von Durikovic [DK06] hergeleiteten Formel zur Berechnung der Intensitäten bei der Interferenz an dünnen Schichten erhält man die in Code 15 aufgeführte Implementierung.

Die Funktion `FresnelInterference` kann nun in die bestehende Wellenlängen-Rechnung

integriert werden, da hier schon finale Intensitäten entstehen. Es sind somit keine weiteren Anpassungen nötig und man erhält die Ergebnisse von Abbildung 27a. Dabei werden die Texturkoordinaten des entsprechenden Objektes verwendet, um eine Variation der Dicke der dünnen Schicht zu erzielen. Unterschiedliche Schichtstärken können gleichzeitig auf einem Objekt modelliert werden und in die Berechnung einfließen. Die variierten Schichten beschreiben in ihrer Form eine umgedrehte Parabel, um einen tropfenartigen Effekt zu erzeugen. Dabei ist die Grundstärke der Schicht und die Variation getrennt einstellbar, so dass unterschiedliche Effekte erzielt werden können (Siehe Abbildung 27b). Natürlich kann die Variation auch auf 0 gesetzt werden, so dass man eine homogene Schicht erhält.

(a) Standard Schichtdicke von 100nm



(b) Variation von Schichtdicke und Parabelhöhe von 300nm

Abbildung 27: Ergebnisse der Methode von Durikovic [DK06]. Lichtspektrum: 38a

Code 15 führt noch viele Berechnungen durch, die ausgelagert werden können, um mehr Performanz im System zu erhalten. Die Berechnungen sind im Vergleich zu den einfachen Intensitäten wesentlich umfangreicher, weshalb Optimierungen sinnvoll sind. So können, wie bereits erwähnt, die Berechnungen der Reflexionsfaktoren ausgelagert werden. Auch sind einige Multiplikationen im voraus berechenbar, wodurch sich eine Steigerung der Performanz einstellt. Die Frameraten der beiden Methoden sind in Tabelle 5 einzusehen.

Die Berechnung der einzelnen Reflexionen der Schichten kann noch auf eine alternative Art und Weise durchgeführt werden. Hierbei werden die Intensitäten von transmittierendem und reflektiertem Anteil jeweils multipliziert. Durch die Verrechnung mit der Weglänge in dem entsprechenden Medium kann der Reflexionsfaktor ermittelt werden. Mithilfe dessen ist es dann möglich, die resultierende Intensität zu bestimmen. Die Berechnung basiert dabei auf dem Tool von [thi]. Der Wegunterschied setzt sich dabei wie folgt zusammen:

Frameraten der einfachen dünnen Film Modelle				
Bildgröße	PathDifference	Fresnel	FresnelFaster	FresnelAlternative
300×400	905	420	730	623
640×480	446	199	356	298
800×600	309	137	247	205
1024×768	196	86	156	130

Tabelle 5: Ergebnisse der einfachen dünnen Film-Modelle

```
half FresnelInterference(half cosangle, half cosangleinn1,
        half cosangleinn2, half lambda, half thickness)
{
    half delta,gamma,cgamma;
    half2 r10,r12,result; // r10 -> reflectance from layer 1 to 0

    delta = 2*nfactor1*thickness*cosangleinn1;
    gamma = twoPI/ lambda * delta;
    cgamma = cos(gamma);

    r10 = fresnelReflectanceCombined(cosangleinn1,cosangle,
            nfactor1,nfactor0);
    r12 = fresnelReflectanceCombined(cosangleinn1,cosangleinn2,
            nfactor2,nfactor1);

    result = ((r10+r12*cgamma)/(1+(r12*r10*cgamma)));

    return 0.5*(dot(result,result));
}
```

Code 15: Berechnung der Interferenz mit Fresnel

$$\Delta = \frac{2\pi}{\lambda} \underbrace{(2 \cdot d \cdot n_2) \cdot \frac{1}{\cos\beta}}_{w_1\text{: Weglänge in dünner Schicht}} - \frac{2\pi}{\lambda} \cdot \underbrace{(2 \cdot d \cdot n_1) \cdot \tan\beta \cdot \sin\alpha}_{w_2\text{: Weglänge in ursprünglichen Medium}} \tag{64}$$

Bestimmt man mit Hilfe der Fresnel-Gleichungen nun die Reflexionsfaktoren, so kann man die einzelnen Komponenten unter Benutztung von $t_{i,j} = |1 - r_{i,j}|$ berechnen. Man erhält für die kombinierten Werte:

$$trt = t_{01} \cdot r_{12} \cdot t_{10}$$
$$trrrt = t_{01} \cdot r_{12} \cdot r_{10} \cdot r_{12} \cdot t_{10}$$

Die resultierende Intensität ist dann gemäß [thi]:

$$R = (trt \cdot \sin\Delta + trrrt \cdot \sin(2\cdot\Delta))^2 + (r_{01} \cdot \cos\Delta + trrrt \cdot \cos(2\cdot\Delta))^2 \tag{65}$$

Einen Großteil der Berechnungen kann man wellenlängenunabhängig durchführen, so dass diese entsprechendend ausgelagert wurden. Die Ergebnisse weichen jedoch von der von Durikovic und Sun vorgestellten und implementierten Versionen ab [DK06, SW08]. Man erkennt allerdings Ähnlichkeiten im Ergebnis, so dass der Unterschied wohl an der Genaugkeit bzw. Approximationen in den verschiedenen Berechnungen liegt. In Abbildung 28 ist ein Ergebnisbild dargestellt, die Frameraten sind in Tabelle 5 aufgeführt. Die Intensitäten der Wellenlängen des verwendeten Lichtes ist dem Spektrum 38a zu entnehmen.

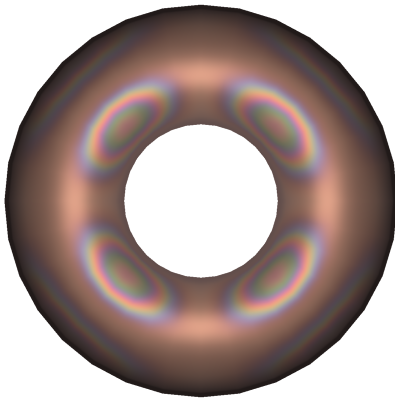


Abbildung 28: Interferenz nach alternativer Berechnung. Lichtspektrum: 38a

Sun et. al. veröffentlichten einen weiteren Ansatz, welcher es auch ermöglicht, komplexe Brechungsindizes in die Rechnung mit einzubeziehen. Die Beschreibung dieses Modells ist in Kapitel 3.2.2 aufgeführt. Entsprechende Anpassungen an das bisherige System wurden vorgenommen und die Ergebnisse sind in Abbildung 29 dargestellt. Frameraten des von Sun et. al. vorgestellten Seifenblasensystems sind in Tabelle 6 aufgeführt. Als Spektrum dient Spektrum 38a.

Eine weitere Methode, Interferenz zu bestimmen, wurde von Fabry und Pérot erläutert. Auch bei dieser tritt der Effekt durch die Phasenverschiebung der Lichtstrahlen auf. In dem zu grundeliegenden Versuch wurde ein sogenanntes Etalon angestrahlt, welches einen bestimmten Reflexionsfaktor aufweist. Ein Etalon ist ein Interferometer nach Fabry-Pérot mit festen Spiegelabstand. Die Transmittanz ist maximal, wenn die Wellen in Phase

Frameraten der komplexen dünnen Film-Modelle				
Bildgröße	`MultiLayer`	`MultiLayerFaster`	`Bubble`	`Fabry-Pérot`
300 × 400	99	163	748	1103
640 × 480	44	74	347	513
800 × 600	30	52	244	362
1024 × 768	19	33	154	229

Tabelle 6: Ergebnisse der erweiterten dünnen Film- Modelle. Die MultiLayer-Systeme bestehen aus 4 dünnen Schichten.

sind, d.h. die Phasenverschiebung ein Vielfaches von $\Delta = \frac{2\pi}{\lambda} \cdot (2nd\cos\theta)$ beträgt.

Durch das Kriterium, dass der Reflexionsfaktor und der Transmissionsfaktor als Summe 1 ergeben ($T + R = 1$), kann die maximale Intensität bei Reflexion wie folgt durch die Transmission des Etalons bestimmt werden:

$$F = \frac{4R}{(1-R)^2}$$
$$T_e = \frac{1}{1 + F \cdot \sin^2(\frac{\Delta}{2})} \tag{66}$$

Dabei bezeichnet $R = \frac{1}{2} \cdot (|r_{\|}| + |r_{\perp}|)$ die resultierende Intensität, die sich aus den Fresnel-Gleichungen ergibt.

Die Interferenzeffekte erscheinen sehr ähnlich zu der von Sun und Durikovic präsentierten Methode und scheinen eine gute Grundlage zu bieten. Das Interferenzkriterium nach Fabry-Pérot wird für die Beschreibung von Interferenz bei Pfau-Federn genutzt und kann bei einer entsprechenden Kalibration gute Ergebnisse liefern. Die Performanz liegt im Bereich der bisherigen Implementationen und ist in Tabelle 6 notiert. In Abbildung 30 sieht man die Interferenzbeleuchtung für die Standardwerte.

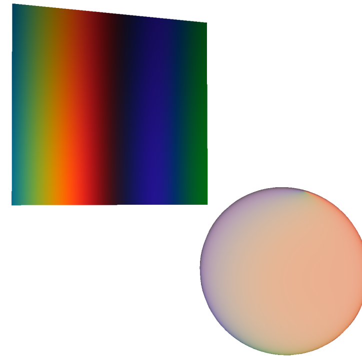


Abbildung 29: Interferenzmodelle nach Sun [SW08]. Lichtspektrum: 38a

Das von Sun vorgeschlagene System wurde mit dem mehrschichtigen Modell von Hirayama [HKYM01] verknüpft. Dies hat zur Folge, dass es sich bei der untersten Schicht des Modells um eine komplexwertige handeln kann, welche somit nicht transparent ist [SW08]. Für die Berechnung eines mehrschichtigen Systems schlagen Hirayama et. al. eine rekursiv definierte Funktion vor, welche die einzelnen Schichten definieren.

Man verwendet dafür zwei rekursive Funktionen γ und τ, die folgende Startwerte haben:

$$\gamma_0 = r_{N+1}$$
$$\tau_0 = t_{N+1}$$

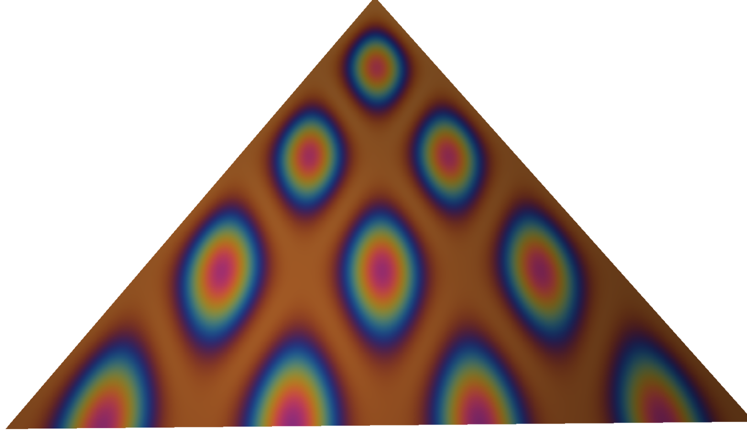


Abbildung 30: Interferenz nach Fabry-Pérot. Lichtspektrum: 38a

Die rekursive Funktion (für $j = N, \ldots, 1$) ergibt sich dann wie folgt:

$$\gamma_{N-j+1} = \frac{r_j + \gamma_{N-j} \exp(2 \cdot i \cdot \Delta_j)}{1 + r_j \cdot \gamma_{N-j} \exp(2 \cdot i \cdot \Delta_j)} \tag{67}$$

$$\tau_{N-j+1} = \frac{t_j \cdot \tau_{N-j} \exp(2 \cdot i \cdot \Delta_j)}{1 + r_j \cdot \gamma_{N-j} \exp(2 \cdot i \cdot \Delta_j)} \tag{68}$$

Dabei ist Δ_i der von Heavens [Hea91] hergeleitete Weglängenunterschied multipliziert mit $\frac{2\pi}{\lambda}$. Eine Herleitung dieser Formel kann mit einem einfachen 3-Schichtsystem (Luft, Film, Objekt) leicht durchgeführt werden und die rekursiven Formeln lauten:

$$\gamma = r_1 + \sum_{m=1}^{\infty} t_1 r_2 (r'_1)^{m-1} t'_1 \exp(2 \cdot i \cdot \Delta_i \cdot m)$$

$$\tau = \sum_{m=0}^{\infty} t_1 (r_2 r'_1)^m t_2 \exp(i \cdot \Delta_i \cdot (2m+1))$$

Dabei stehen r' und t' für die invertierten Reflexionen oder Transmissionen an der entsprechenden Schicht. Nutzt man den Zusammenhang zwischen diesen aus, lassen sich die Formeln vereinfachen:

$$\gamma = \lim_{m \to \infty} \frac{r_1 + \left(r_1^2 + \frac{\hat{n}_1 \cdot \cos\theta_1}{\hat{n}_0 \cdot \cos\theta_0} t_1^2\right) r_2 e^{2i\Delta_1} - r_2 \frac{\hat{n}_1 \cos\theta_1}{\hat{n}_0 \cdot \cos\theta_0} t_1^2 e^{2i\Delta_1} \left(-r_1 r_2 e^{2i\Delta_1}\right)^m}{1 + r_1 r_2 e^{2i\Delta_1}}$$

$$\tau = \lim_{m \to \infty} \frac{t_1 t_2 e^{2i\Delta_1} - t_1 t_w e^{2i\Delta_1} (-r_1 r_2 e^{2i\Delta_1})^m}{1 + r_1 r_2 e^{2i\Delta_1}}$$

Des weiteren folgt aus den Fresnel-Gleichungen: $r^2 + \frac{\hat{n}_1 \cdot \cos\theta_1}{\hat{n}_0 \cdot \cos\theta_0} t^2 = 1$

Da die geometrische Reihe konvergiert ($|r_1 r_2 e^{2i\Delta}(1)| < 1$), ergibt sich als endgültige rekursive Formel:

$$\gamma = \frac{r_1 + r_2 e^{2i\Delta_1}}{1 + r_1 r_2 e^{2i\Delta_1}}$$

und

$$\tau = \frac{t_1 t_2 e^{2i\Delta_1}}{1 + r_1 r_2 e^{2i\Delta_1}}$$

Schliesst man nun von 3 Schichten auf $N+2$ Schichten, so erhält man die oben angegebene Formel. Da die letzte Schicht hierbei zuerst berechnet werden muss, kann eine entsprechende Implementierung sehr effizient in das System, welches in dieser Diplomarbeit beschrieben wird, übernommen werden. Im Grunde sind alle Reflexionsfaktoren wellenlängenunabhängig berechenbar und könnten somit ausgelagert werden. Die einzige Limitation bietet hierbei `CG`, da große Datenmengen zwischen Funktionen nicht mittels eines Pointers gehandhabt werden können. Dies bedeutet, für jede Iteration muss die komplette, vorberechnete Datenmenge übergeben werden. Die Problematik liegt darin, dass die Anzahl der Schichten im Voraus nicht bekannt ist und die Vorberechnungen von jedem Fragment-Shader durchgeführt werden. Da keine Limitation in das System eingefügt werden soll, wurde mit einem sehr großen Array (512 Elemente) experimentiert. Da die Geschwindigkeit stark eingebrochen ist, wurde das Array immer weiter verkleinert, bis schließlich bei einem 10 Schichtsystem eine ausreichende Performanz erzielt wurde. Der Geschwindigkeitsvorteil, gegenüber der Implementation ohne Array, ist jedoch vernachlässigbar und beschränkt das System auf eine maximale Anzahl von 10 Schichten. Die Vorberechnung von oberster und unterster Schicht bietet eine ausreichende Beschleunigung und wird im folgenden verwendet.

Berechnet man die Reflexionsfaktoren von Objekt- und Luftschicht, so kann man dennoch eine Beschleunigung erzielen, auch wenn sie mit der Übergabe aller einzelnen Reflexionsfaktoren wesentlich größer wäre. Ein 3-Schichtsystem kann somit ohne Rekursion durchgeführt werden. Die Methode für die Berechung des komplexen Reflexionsfaktors wurde so verwirklicht, dass eine Fallunterscheidung angewendet wird. Im Fall von $\kappa = 0$ kann die vereinfachte Rechnung durchgeführt werden. Eine Implementation von variablen Brechungsindizes und Extinktionskoeffizienten lässt sich leicht realisieren.

Die Implementation von Hirayama et. al. wurde auf einem Raytracer-System durchgeführt, welches vorberechnete Reflexions- und Transmissionfaktoren zur Verfügung hatte. Dies bedeutet, dass einzelne Änderungen zur Laufzeit nicht möglich sind. Zum Zeitpunkt der Implementation (2000) benötigte der von Hirayama entwickelte Raytracer für ein 800×600 Bild mit

Frameraten in Abhängigkeit der Schichtzahl				
1-Schicht	4-Schichten	8-Schichten	12-Schichten	16-Schichten
82	25	14	9	7

Tabelle 7: Ergebnisse des MultiLayer-Systems mit variabler Schichtzahl bei einer Auflösung von 640×480 Pixeln

2×2 Super Sampling[23] ca. 100 Minuten [HKYM01]. Die Implementation auf dem in dieser Arbeit eingeführten System bringt, mit Verwendung der angegebenen Optimierungen, mit der verwendeten Hardware (Intel Core Duo E6750, 4 GB Ram, nVidia Quadro FX 4600) interaktive Frameraten. Je nach verwendeter Tiefe variiert hierbei die resultierende Geschwindigkeit annähernd linear. Für eine Auflösung in der Standardkonfiguration sind die Frameraten in Tabelle 7 aufgeführt.

Kombiniert man das so entstandene System mit den Spektraltexturen, so spielt zusätzlich zu den Interferenzfaktoren noch die grundlegende Intensität des verwendeten Objektes eine Rolle. Somit kann es zu verschiedenen Interferenzerscheinungen kommen, da abhängig von den vorhandenen Wellenlängen im Spektrum, diese unter den entsprechenden Winkel auftreten. Da das mehrschichtige System auch durch die Optimierungen in der Lage ist, das normale 3 Schichtsystem ohne große Geschwindigkeitseinbrüche zu modellieren, wurde auf eine Implementation des statischen 3 Schicht-Systems verzichtet. Die Ergebnisse des MultiLayer-Systems sind in Abbildung 31 zu sehen.

Da die Beleuchtungsrechnung möglichst universell gehandhabt wurde, ist es auch möglich, das System in ein Bumpmapping-Verfahren einzubinden. Durch die Bumpmaptextur kann man die Stärke der Schicht durch eine Textur variieren und zusätzliche Oberflächenstrukturen erzeugen. Für das Bumpmapping werden die einzelnen Vektoren allerdings idealerweise in den sogenannten *TangentSpace* transformiert, wodurch eine Reihe von Vorteilen entstehen. Die Beleuchtungsrechnung ändert sich nicht, jedoch ist das Pertubieren der Normalen, welches

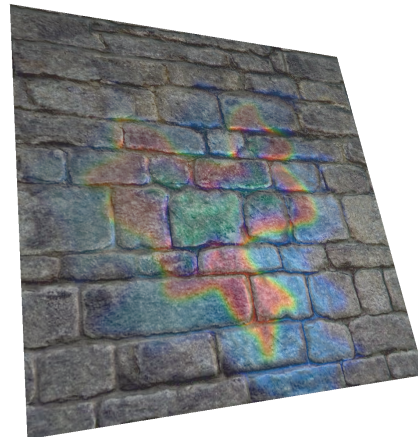


Abbildung 32: Interferenz in Kombination mit Bumpmapping

[23] Super Sampling: Berechnung mehrer Pixel, welche zusammengefasst werden. Wird zur Vermeidung von Alias-Effekten genutzt

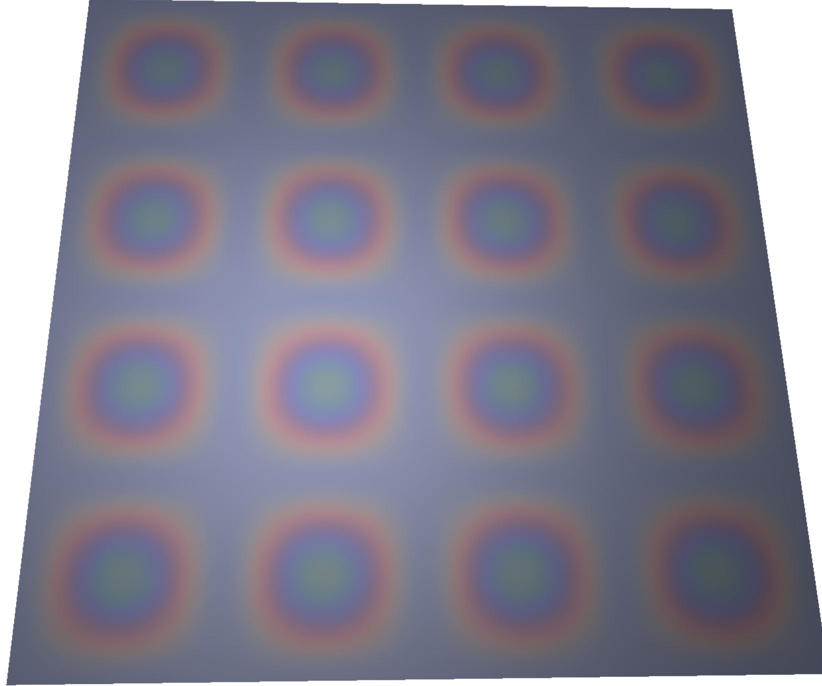


Abbildung 31: MultiLayer System, in Kombination mit komplexen Brechungsindizes und Spektraltexturen. Lichtspektrum: 38b, Materialspektrum: 39a

die Oberfläche simuliert, sehr einfach durch-
zuführen. Der *TangentSpace* wird durch 3 Vektoren, die jeweils senkrecht aufeinander stehen, erzeugt. Durch die Normale, die Tangente und Binormale kann eine Transformationsmatrix erstellt werden. Multipliziert man die entsprechenden Vektoren mit dieser Transformationsmatrix, so erhält man den Vektor im *TangentSpace*. Der Raum ist dabei wie folgt definiert:

$$M_{\text{TangentSpace}} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix} \tag{69}$$

Die Vektoren sollten alle normalisiert sein und sich entweder im *World*- oder *View*-Space befinden, damit die Rotation der Vektoren, die sich im selben Vektorraum befinden müssen, korrekt funktioniert. Im Fragment-Shader hat man dann den Vorteil, dass die Normale in z-Richtung zeigt, also durch den Vektor $\vec{n} = (0, 0, 1)$ definiert ist. Nutzt man nun eine sogenannte Normalmap[24], so kann die für das Fragment interpolierte Texturkoordinate genutzt

[24] In einer Normalmap wird für jede Texturkoordinate eine Normale gespeichert

Frameraten des `MultiLayer`-Modells			
Bildgröße	Spektrumtexturen	1 Schicht	*complex IOR* ($\kappa = 2$)
300×400	333	500	395
640×480	144	217	172
800×600	101	152	120
1024×768	63	96	75

Tabelle 8: Ergebnisse der einfachen dünnen Film-Modelle

werden, um aus der Normalmap eine neue Normale auszulesen. Diese „neue“ Normale kann ohne weitere Transformation für die Beleuchtungsrechnung verwendet werden. Dadurch entsteht die Illusion einer rauhen Oberfläche. Da dies unabhängig von einzelnen Wellenlängen berechnet werden kann, ist das eigene System ohne Probleme mit Bumpmapping erweiterbar. Dazu wird ein Vertex-Shader implementiert, welcher die einzelnen Vektoren in den *Tangent-Space* anstelle des *World*-Space transformiert. Die darauf folgende Berechnung der Interferenz bleibt identisch. Ein Beispiel mit Interferenzeffekten, die abhängig von der Normalmap entstehen, ist in Abbildung 32 zu sehen. Simuliert man eine Wasseroberfläche, so erhält man sehr realistische Ergebnisse. Ein Bild der Shader-Ausgabe ist in Abbildung 33 gegeben.

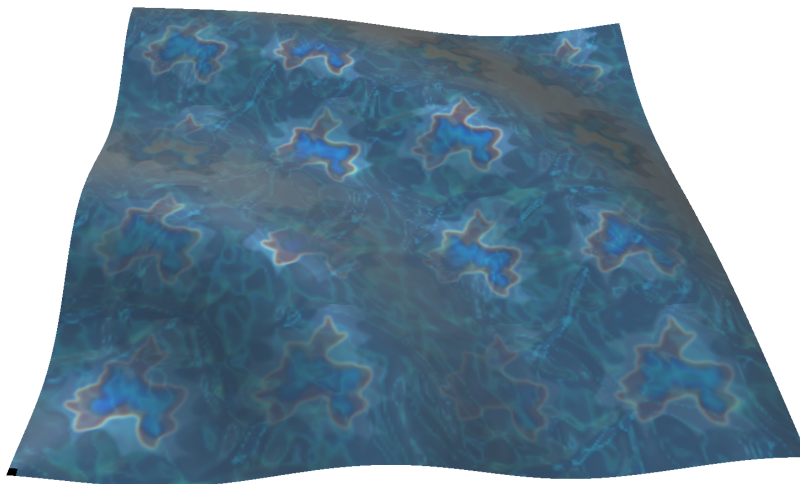


Abbildung 33: Interferenz auf einer Wasseroberfläche

4.5 Modellierung der Diffraktion an einer CD

Eine CD schillert in der Sonne oder einem beliebigen Licht, da die Sichtstrahlen interferieren. Dabei ist eine räumliche Kohärenz vorhanden, wenn man eine CD als ein Reflexionsgitter modelliert. Dies bedeutet, das Licht interferiert wie bei einem Gitter, jedoch wird das Licht nicht transmittiert, sondern reflektiert. Statt die Interferenz hinter dem Gitter auf einem Schirm zu beobachten (Siehe Kapitel 2.2.4), werden die Lichtstrahlen zurückgeworfen und bilden, abhängig von der Wellenlänge, einzelne Maxima, die unter verschiedenen Winkeln zu beobachten sind. Da hier das Licht kein Objekt durchdringt, ist der Aufwand der Berechnung überschaubar und mit einem Polygon-Renderer modellierbar, da der Effekt direkt auf der Oberfläche entsteht. Wie bei der Inferenz an einer dünnen Schicht wird die Strahlberechnung für jedes Pixel unabhängig durchgeführt, da man den Einfluss anderer Positionen hier nicht berücksichtigen muss.

Abbildung 34: CD mit eigenem Shader. Lichtspektrum: 38a

Auf Grund der Spuren (nicht der Pits und Lands) kommt es zur Interferenz, in dem die Strahlen sich dann mit einer Phasenverschiebung überlagern. Bei einer CD beträgt der Spurabstand ca. 1600nm, der Spurabstand einer DVD beträgt ca. 740nm.

Zur Berechnung der Spur wird im Fragment-Programm mittels der Texturkoordinaten ein Vektor zum Mittelpunkt des Objektes bestimmt. Berechnet man zu diesem (2-dimensionalen) Vektor die Normale, so kann man mittels eines Skalarproduktes feststellen, ob das Licht sich senkrecht zu den Gitterlinien befindet. Des weiteren kann ermittelt werden, wie groß der Abstand der Spuren zueinander ist. Liegt das Licht senkrecht zur Spur, so ist der Abstand minimal, ansonsten wächst dieser mit dem Cosinus des Winkels an.

$$\text{influence} = |\vec{G} \odot \vec{L}|$$

$\vec{G}$ ist dabei der Vektor der Normalen des Vektors zum Mittelpunkt des Objektes, also in Richtung der Spuren.

Die Interferenz findet dann statt, wenn zwei Strahlen an verschiedenen Stellen des Reflexionsgitters auftreffen und diese einen Weglängenunterschied von $\Delta = n\lambda$ haben, mit $n \in \mathbb{N}_0$.

Dabei hängt der Weglängenunterschied sowohl vom einfallenden Licht, als auch von der Position der Kamera ab. Man erhält als Weglängenunterschied:

$$\Delta = \text{Griddistance} \cdot |\sin\alpha - \sin\beta| \tag{70}$$

Dies folgt aus der Geometrie des Aufbaus, wie man Abbildung 35 entnehmen kann. Aus der Winkelbeziehung folgt für die beiden Strecken x, y:

$$x = \cos(90^\circ - \beta) \cdot d = \sin\beta \cdot d$$
$$y = \cos(90^\circ - \alpha) \cdot d = \sin\alpha \cdot d$$

Daraus ergibt sich mit $d = \text{Griddistance}$ die obere Formel.

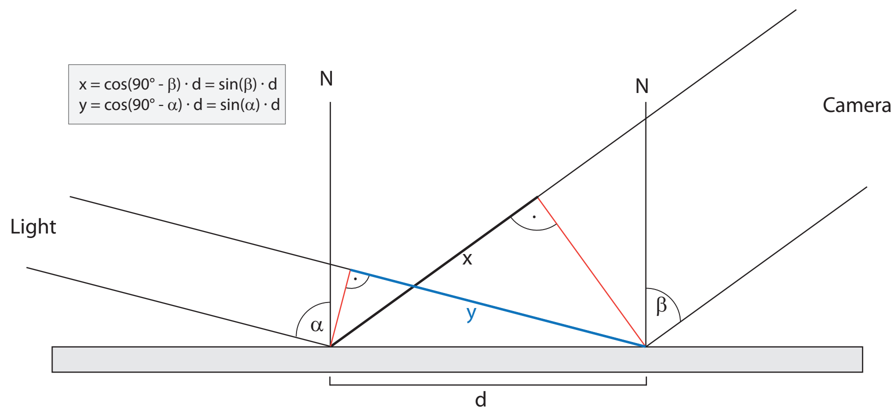


Abbildung 35: Strahlengang am Reflexionsgitter

Ein Maximum ergibt sich, wenn

$$\Delta \mod \lambda = 0 \tag{71}$$

Zur Berechnung der Intensität wurden folgende Vereinfachungen angenommen:

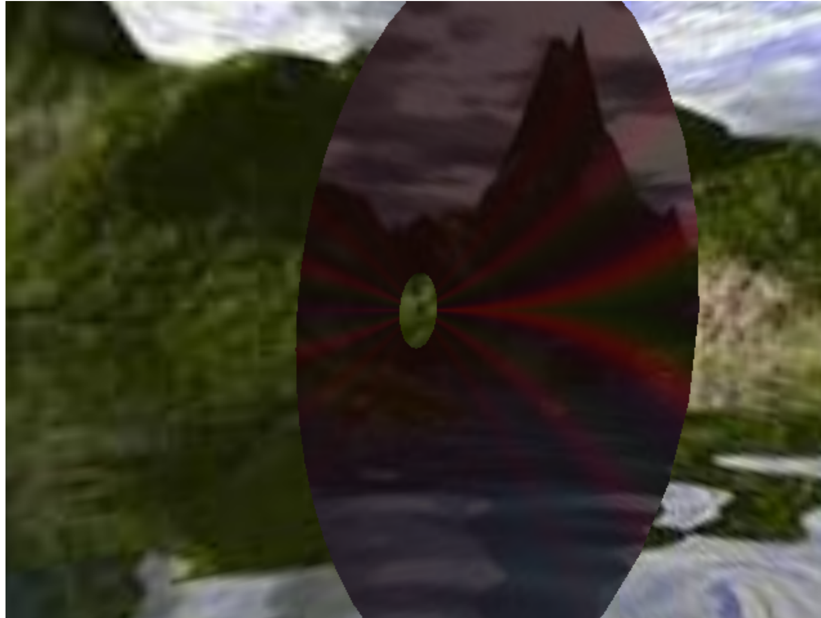
- Der maximale Spurabstand beträgt das 3-fache des ursprüngliches Wertes
- Die Intensitäten fallen linear vom Maximum ab

Dadurch ergibt sich die folgende Intensitäten-Formel:

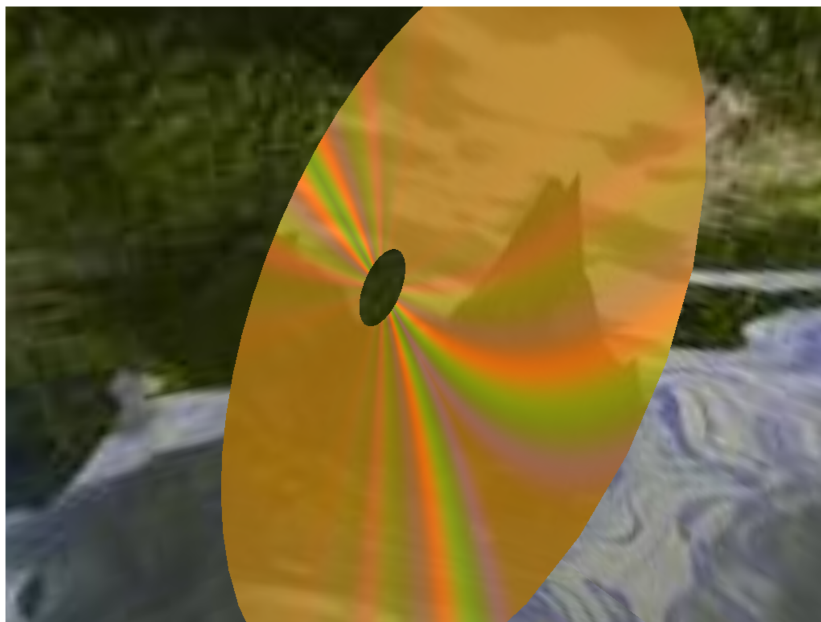
$$I_R = \frac{(\text{Griddistance} \cdot |\cos\theta_{\text{camera}} - \cos\theta_{\text{light}}| \mod \lambda)}{\lambda} \tag{72}$$

Die Intensität wird noch hart abgeschnitten, so dass nur Werte größer als ein Schwellenwert angenommen werden, ansonsten wird sie auf 0 gesetzt. Dadurch kommen nur die

Interferenzen mit großer Intensität zum Tragen. Versieht man die Szenerie um die CD nun mit einer Environmentmap in Form einer Cubemap, so ergeben sich Abbildungen 34 und 36.



(a) Lichtspektrum: 38b, Substratspektrum: 39c



(b) Lichtspektrum: 39b, Substratspektrum: 39d (Gold)

Abbildung 36: CD mit Substratschicht. Aufnahme mit unterschiedlichen Licht und Substratspektren

4.6 Level Of Detail System

Eine Erweiterung ist ein LOD[25]-System, welches je nach angeforderter Genauigkeit, einzelne (unwichtige) Wellenlängen in der Berechnung auslässt. Dadurch kann eine Geschwindigkeitssteigerung erfolgen, da die Performanz direkt von der Anzahl der gesampelten Wellenlängen abhängt.

Eine Möglichkeit ist es, die CMF-Werte nur durch Stützpunkte zu repräsentieren und die Zwischenwerte durch eine lineare Interpolation zu erhalten. Da `CG` eine entsprechende Funktion für die Interpolation zur Verfügung stellt, kann davon ausgegangen werden, dass diese Methode sehr effizient ist. Auf diese Weise kann man die Anzahl der Texturzugriffe vermindern, da einzelne Werte berechnet werden können. Die Stützpunkte der CMF-Werte sind durch eine Formel bestimmbar und die Stufe des LOD-Systems kann dynamisch angepasst werden.

Geht man von 513 abgetasteten Werten aus, so lassen sich die Stützpunkte durch Aufteilung in 2 Hälften beschreiben. Man erhält nach der ersten Iteration die gleichgroßen Blöcke $[1-256]$ und $[257-513]$. Diese lassen sich entsprechend wieder in 2 Hälften teilen. Somit erhält man auf der höchsten Stufe die größte Genauigkeit, welche dann nach unten immer weiter abnimmt. Die Randpunkte der einzelnen Blöcke definieren dabei, welche Punkte der CMF-Kurven ausgelesen werden, die Bereichslänge, wieviele Werte interpoliert werden müssen.

Durch dieses Verfahren ist es in gewissen Grenzen möglich, eine Beschleunigung in dem Framework zu erzielen. Die größte Beschleunigung erzielt man, wenn der Compiler die einzelnen Schleifen entfalten kann und die Kosten für die Interpolation und die Texturzugriffe sich in einem Gleichgewicht halten. Für das bestehende Framework wurde eine rudimentäre Implementation vorgenommen und die erzielten FPS werden in Tabelle 9 aufgeführt. Das LOD-System kann auf alle vorhandenen Spektren angewendet werden und erlaubt dadurch eine dynamische Anpassung der Genauigkeit in der Abtastung.

LOD-Systeme kommen hauptsächlich für Objekte zum Tragen, die in einem Bild eine eher untergeordnete Rolle spielen. So können weiter entfernte oder kleine Objekte mit einem geringeren Aufwand berechnet werden, so dass mehr Rechenleistung für große und wichtige Elemente eines Bildes zur Verfügung steht.

Für die Interferenzberechnung kann es jedoch wichtig sein, dass die Werte genauer verarbeitet werden (ohne Interpolation), da eventuell einzelne Peaks in den Intensitäten nicht beachtet werden. Dies kann zu einem dramatischen Farbunterschied je nach Abtastung führen. Daher muss die Verwendung eines LOD-Systems genau abgewägt und eventuell andere Optimierungen gefunden werden.

[25] Level Of Detail

Eine Gewichtung der Stützpunkte nach ihrer Intensität würde zu einem eventuell besseren Ergebnis führen, doch müssen diese auch von den CMF-Werten entsprechend aufgefangen werden. Da die Stützpunkte dementsprechend zufällig auf dem Spektrum verteilt sein können, ist es mit dem LOD-System nicht mehr möglich, die zugehörigen CMF-Werte effizient zu berechnen.

Nutzt man die Gewichtung in der CMF-Textur, so könnten die wichtigen Wellenlängen zuerst betrachtet werden. Dazu muss man in der Textur im Alphakanal zusätzlich die zugehörige Wellenlänge speichern, so dass den einzelnen Werten eine eindeutige Wellenlänge zugeordnet werden kann. Eine Realisierung als **byte**-Textur wäre somit nicht mehr möglich. Zusätzlich fällt der wahlfreie Zugriff auf die Wellenlängen weg, so dass einzelne Besonderheiten des Systems nicht mehr verfügbar sind.

Die bisherigen Systeme von Durikovic und Sun zeigen keine LOD-Systeme und lassen dieses Kapitel auch vollkommen unangetastet. So bleibt in diesem Bereich noch viel Raum für Forschung offen.

Vergleich der LOD Stufen		
LOD-Stufe	**byte**-Textur	**float**-Textur
Full Fetch	524	282
7	620	619
6	674	624
6 (ohne Unroll)	595	236

Tabelle 9: LOD-System mit unterschiedlicher Implementation. Bildgröße von 1024×768 Pixel

5 Aussichten und Diskussion

Das in dieser Arbeit vorgestellte Framework zeigt, dass es möglich ist, Beleuchtungsrechnung mit einzelnen Wellenlängen durchzuführen. Die neusten Generationen von GPUs ermöglichen es, große Mengen von Daten effektiv zu verarbeiten. Die von Sun und Durikovic [SW08, DK06] vorgestellten Verfahren gehen dabei jedoch in verschiedene Richtungen. Diese Arbeit versucht mehr in den physikalen Bereich vorzustoßen und auf eine entsprechende Beschreibung und Wiedergabe zu zielen. Dabei basieren die einzelnen Shader-Programme auf den präsentierten physikalischen Formeln.

Im Unterschied zu dem von Sun vorgeschlagenen System, wird keine Transformation von RGB-Werten durchgeführt, sondern Spektren definiert. Da eine RGB-Farbe unendlich viele Repräsentationsmöglichkeiten durch Spektren besitzt, mag dies auf den ersten Blick nicht sinnvoll erscheinen, da man mehr Daten speichert, die nach der RGB-Transformation wieder verloren gehen (Metamerie). Jedoch bleibt die Möglichkeit bestehen, Interferenz mit dem Original-Spektrum zu berechnen und dadurch die einzelnen Intensitäten explizit hervorzuheben. Die Charakteristika der Materialien kommen somit mehr zum Tragen. Zusätzlich ermöglicht die Wellenlängenrepäsentation der Farben ein unabhängiges System für die Farbwiedergabe zu wählen. Es gibt mehrere Ansätze für eine Transformation von RGB-Werten in das XYZ-System [SW08, Smi99].

Diese Arbeit zeigt auch, dass in dem entwickelten Modell die bekannten Beleuchtungsrechnungen implementiert und ohne große Verluste darauf angewendet werden können. Die Ergebnisse zeigen zufriedenstellende Resultate. Es können zwar auch RGB-Farben modelliert werden, jedoch erscheinen alternative Verfahren sinnvoller. Da die Berechnung per Pixel durchgeführt wird, leidet die Performanz des Frameworks stark, denn jeder Fragment-Shader muss eine Iteration über alle Wellenlängen durchführen. Das vorgestellte LOD-System ermöglicht hierbei zwischen Genauigkeit und Korrektheit zu skalieren.

5.1 Texturdefinition

Das Framework nutzt bis dato aus, dass einzelne Spektren auf die RGBA-Kanäle einer Textur verteilt werden. Jedoch kann es zur Berechnung von Interferenzen an einer dünnen Schicht oder Brechungen nach Snell sinnvoll sein, eine andere Repräsentation zu wählen. Da Brechungsindizes, sowie Extinktionskoeffizienten von Materialien auch von der Wellenlänge abhängen können, können diese in Kombination mit den Intensitäten des aktuellen Spektrums gespeichert werden. So erhält man mit einem Texturzugriff alle Daten, die für die Berechnung von Interferenzen für das vorgeschlagene Multi-Layer-System benötigt werden.

Da eine Schicht nicht nur einen wellenlängenabhängigen Brechungsindex und Extinkti-

onskoeffizienten besitzen kann, sondern auch ein eigenes Spektrum, muss das Framework zur Berechnung der resultierenden Farbe eigentlich dieses mit berücksichtigen. Da die Intensitäten sich hierbei durch die subtraktive Farbmischung ergeben, können die einzelnen Intensitäten multipliziert werden. Dadurch würde die Farbe der Schicht das Resultat der Interferenz direkt beeinflussen. In der bisherigen Implementation des Frameworks werden jedoch nur transparente Schichten mit konstanten Brechungsindizes und Extinktionskoeffizienten betrachtet.

Nutzt man das Framework in seiner bestehenden Form, fällt schnell auf, dass es ein großes Problem sein kann, einzelne Spektren für Materialien oder Licht zu erlangen. Solche Daten sind nicht einfach zu erhalten. Manche Glühbirnen-Hersteller zeigen Spektralkurven, die mit einem entsprechendem Programm, z.B. *Spectral Librarian* von Sebastian Schäfer, eingelesen werden können. Daher fällt es gerade bei Interferenzeffekten schwer, vergleichbare Ergebnisse zu produzieren. So lässt sich selbst durch die Verwendung von den Brechungsindizes von Wasser und Öl zwar ein Interferenzmuster erzielen, doch ist es mit realen Bildern (Abbildung 37) so nicht vergleichbar, da auch Öl und Wasser indiviuelle Spektralkurven besitzen.

Abbildung 37: Ölschicht auf einer Wasserpfütze. Aus [hyp]

Auch wenn diese bekannt wären, so könnte das Bild 37 nur in guter Näherung approximiert werden, da wesentlich mehr Faktoren in die finale Beleuchtung miteinfließen. So spielt die Dicke einer Schicht, die auf der Wasseroberfläche nur in Näherung konstant ist, eine entscheidende Rolle. Zusätzlich wird die Ölschicht nicht nur von einem Objekt beleuchtet - die globale Illumination ist für den resultierenden Eindruck ein entscheidender Faktor. Jedoch sollte es möglich sein - in guter Näherung - diese mit lokalen Beleuchtungsmodellen darzustellen.

Brechungsindex, Extinktionskoeffizient und Dicke sind nur für Objekte interessant, da Lichtquellen diese Informationen nicht benötigen. Das System kann in diesem Punkt weiter spezialisiert werden, indem auf die ursprüngliche kombinierte Darstellung zurückgegriffen wird, wodurch einzelne Texturzugriffe optimiert werden können. Jedoch müsste eine individuelle Behandlung für Licht- und Objektspektren stattfinden. Nutzt man das neue Texturformat, so könnten weitere Spektren in den G, B, und A Kanälen gespeichert werden. Dadurch wird Speicherplatz gespart oder das Framework, durch eine vorherige Anordnung der Spektren, optimiert.

Denkbar wäre, ein lokales und ein globales Licht in jeder Texturzeile zu speichern, so dass bis zu 4 Lichtquellen gleichzeitig berechnet werden können. Dabei sind kaum Leistungseinbrüche zu erwarten, da die GPU Vektordaten sehr effizient verarbeitet. Die Daten werden

intern als Vektoren dargestellt, nicht verwendete Komponenten eines 4-elementigen Vektors werden mit 0-Werten aufgefüllt. Für die Berechnung auf der GPU macht es somit keinen Unterschied, ob man 1-elementige oder 4-elementige Vektoren für die Berechnung nutzt. Da in der gleichen Zeit 4 Daten berechnet werden können, ist die Effizienz bei einer solchen Implementation entsprechend höher.

Da nicht immer eine Berechnung auf Wellenlängen-Ebene nötig ist, kann es durchaus sinnvoll sein, das System entsprechend zu optimieren. Die bisherigen Systeme können dies einfach durchführen, da sie grundsätzlich in dem RGB-Farbmodell arbeiten und diese dann in das XYZ-Modell transformieren. Möchte man nur die Farbe des Objektes bestimmen, die sich aus dem gegebenen Spektrum ergibt, so fällt die Transformation weg und die angegebene Farbe wird ausgegeben.

In dem präsentierten Framework ist dies bis dato nicht möglich, da nur Spektren zur Verfügung stehen, die auf jeden Fall erst verrechnet werden müssen. Es bietet sich daher an, die Textur zu erweitern. Speichert man hinter den Spektren die resultierenden RGB-Farben, welche entstehen würden, wenn man sie mit den Tristimulus-Werten verrechnet, so kann die Farbe durch einen Texturefetch ermittelt werden. Dies sollte wesentlich schneller sein, als 1024 Werte aus den Texturen auszulesen (512 für das Spektrum, und 512 für Tristimulus-Werte).

Dadurch wäre es mit dem Framework auch effizient möglich, normale Objekte, welche keine Interferenz- oder wellenlängenbasierte Berechnung benötigen, anzuzeigen. Da die Speicherung einer Textur auf den aktuellen GPUs und APIs nicht mehr abhängig von einer Größe zur Basis 2 ist, lassen sich auch entsprechende Texturen einfach definieren. Da die Anzahl der zusätzlichen Informationen ein konstanter Wert ist, können die einzelnen Spektren trotzdem über die Farbe eines Objektes referenziert werden.

Um eine gleichbleibende Breite einer Textur zu erlangen, so dass Licht- und Objekttexturen von der gleichen Grundroutine bearbeitet werden können, muss ein entsprechender Bereich zwischen 2 Spektren aufgefüllt werden. Hierbei bestimmt die größte Anzahl von Zusatzinformationen, wie groß dieser Bereich wird. Bei einer Lichttextur werden zusätzlich 4 Farben gespeichert, da sich in jedem Kanal ein unterschiedliches Spektrum befindet. Bei einer Objekttextur werden Spektrum, Brechungsindex und Extinktionskoeffizient gespeichert. Am Ende der spektrumbasierten Daten wird eine Farbe abgelegt. Zusätzlich können die einzelnen Daten normiert werden, so dass die Genauigkeit der Daten bei Speicherung als **byte**-Textur erhöht wird.

Die normierten Daten erhalten dann ein Maximum und ein Minimum, welches am Ende der Textur gespeichert wird. Geht man bei den Intensitäten davon aus, dass sie immer normiert sind, braucht man 2 zusätzliche Informationen, die in einem RGBA-Texel gespeichert werden können. Der R und G-Kanal würden Minimal- und Maximalwerte des Brechungsindex

beinhalten (in der Regel größer als 1), und der B und A-Kanal die Werte des Extinktionskoeffizienten. Die benötigte Breite würde 4 Texel betragen, wodurch sich das Format festlegt.

5.2 Weitere Ergänzungsmöglichkeiten

Das Framework ermöglicht es, beliebige Beleuchtungsmodelle mit Wellenlängen zu berechnen und dazu einen Polygon-Renderer zu nutzen. Die Verwendung von Texturen und deren Verarbeitung bleibt dabei aussen vor, sollte aber dennoch betrachtet werden. Texturen werden in den meisten Anwendungen genutzt, da sie auf einfache Weise Photorealismus erzeugen.

Durikovic präsentierte dafür ein Verfahren, eine RGB-Environmentmap für die Beleuchtung der Szenerie zu verwenden. Doch bleibt auch hier eine direkte Anwendung einer Textur auf ein Objekt nicht behandelt.

Eine Kombination aller bisher vorhanden Effekte in einem konsistenten System würde es ermöglichen, sehr realistische Bilder zu generieren, die auf Interferenzeffekten aufbauen. Dabei könnte die Environmentmap-Beleuchtung von Durikovic, von Sun die Transformation von RGB- in XYZ-Koordinaten (bzw. Spektren) und die hier präsentierte Multi-Layer-Interferenz Berechnung verwendet werden. Mit den Optimierungen des LOD-Systems und der erweiterten Textur sollten dabei interaktive Frameraten möglich sein.

A Herleitung des komplexen Reflexions- und Transmissionsfaktors

Für die Rechnung mit komplexen Brechungsindizes ist eine aufwändige Herleitung nötig, die im folgenden durchgeführt wird.

Licht fällt von einem transparenten auf ein undurchsichtiges Material. Der einfallende Winkel ist durch das transparente Material gegeben als θ_1, somit folgt nach Snellius für den Winkel in $\hat{n}_2$:

$$\sin\theta_2 = \frac{n_1 \sin\theta_1}{n_2 + i\kappa_2} = \frac{n_2 - i\kappa_2}{n_2^2 + \kappa_2^2} n_1 \sin\theta_1$$

Dies ergibt sich aus: $(x+iy)\cdot(x-iy) = x^2 - i^2y^2 = x^2 + y^2$ mit $i = \sqrt{-1}$.

Aus den einfachen Sinus, Cosinus Beziehungen folgt auch:

$$\cos\theta_2 = \sqrt{1-\sin^2\theta_2} = \left[1 - \frac{n_2^2 - 2in_2\kappa_2 - \kappa_2^2}{(n_2^2+\kappa_2^2)} n_1^2 \sin^2\theta_1\right]^{\frac{1}{2}}$$

Da es sich bei $\cos\theta_2$ um eine komplexe Zahl handelt, kann man sie schreiben als:

$$\cos\theta_2 = u + iv$$

Wobei u und v real sind. Quadriert man nun diese Zahl, so ergibt sich

$$\cos^2\theta_2 = u^2 + 2iuv - v^2$$

Führt man nun einen Koeffizientenvergleich durch, so erhält man

$$u^2 - v^2 = 1 - \frac{n_2^2 - \kappa_2^2}{(n_2^2+\kappa_2^2)^2} n_1^2 \sin^2\theta_1,$$

$$2uv = \frac{2n_2\kappa_2}{(n_2^2+\kappa_2^2)^2} n_1^2 \sin^2\theta_1$$

Quadriert man wiederrum die Gleichungen, so erhält man:

$$(u^2-v^2)^2 = 1 - 2\frac{n_2^2 - \kappa_2^2}{(n_2^2+\kappa_2^2)^2} n_1^2 \sin^2\theta_1 + \frac{(n_2^2-\kappa_2^2)^2}{(n_2^2+\kappa_2^2)^4} n_1^4 \sin^4\theta_1$$

$$4u^2v^2 = \frac{4n_2^4\kappa_2^4}{(n_2^2+\kappa_2^2)^4} n_1^4 \sin^4\theta_1$$

Addiert man diese Gleichungen, so erhält man durch $(a-b)^2 + 4ab = (a+b)^2$ folgende Formel:

$$(u^2+v^2)^2 = 1 - 2\frac{n_2^2 - \kappa_2^2}{(n_2^2+\kappa_2^2)^2} n_1^2 \sin^2\theta_1 + \frac{(n_2^2-\kappa_2^2)^2}{(n_2^2+\kappa_2^2)^4} n_1^4 \sin^4\theta_1 + \frac{4n_2^2\kappa_2^2}{(n_2^2+\kappa_2^2)^4} n_1^4 \sin^4\theta_1$$

Wir substituieren nun wie folgt:

$$a = (n_2^2 - \kappa_2^2)$$
$$b = n_1^2 \cdot \sin^2 \theta_1$$
$$c = (n_2^2 + \kappa_2^2)^2$$
$$d = 2 \cdot n_2 \cdot \kappa_2$$

Damit erhalten wir die folgende Formel:

$$(u^2 + v^2)^2 = 1 - \frac{2ab}{c} + \frac{a^2b^2}{c^2} + \frac{d^2b^2}{c^2} = 1 - \frac{2ab}{c} + \frac{b^2 \cdot (a^2 + d^2)}{c^2}$$

Wir können nun $(a^2 + d^2)$ vereinfachen:

$$(n_2^2 - \kappa_2^2)^2 + 4 \cdot n_2^2 \cdot \kappa_2^2 = (n_2^2 + \kappa_2^2)^2 = c$$

Somit erhalten wir dann:

$$(u^2 + v^2)^2 = 1 - \frac{2ab}{c} + \frac{b^2c}{c^2} = 1 - \frac{2ab}{c} + \frac{b^2}{c}$$

Wir bringen nun alles auf den Nenner c und erhalten damit:

$$(u^2 + v^2)^2 = \frac{c - 2ab + b^2}{c}$$

Wir können c in a umwandeln, in dem wir das gemischte Glied von $(n_2^2 + \kappa_2^2)^2$ durch $-2n_2^2\kappa_2^2$ ersetzen. Dazu ziehen wir $4n_2^2\kappa_2^2 = d^2$ ab und addieren es später hinzu (Erweitern mit 1). Wir erhalten damit

$$(u^2 + v^2)^2 = \frac{a - 2ab + b^2 + d^2}{c} = \frac{(a - b)^2 + d^2}{c}$$

Machen wir die Substitution rückgängig, so erhalten wir die Formel aus [SW08]:

$$(u^2 + v^2)^2 = \frac{(n_2^2 - \kappa_2^2 - n_1^2 \cdot \sin^2 \theta_1)^2 + 4 \cdot n_2^2 \cdot \kappa_2^2}{(n_2^2 + \kappa_2^2)^2}$$

bzw.

$$u^2 + v^2 = \frac{\sqrt{(n_2^2 - \kappa_2^2 - n_1^2 \cdot \sin^2 \theta_1)^2 + 4 \cdot n_2^2 \cdot \kappa_2^2}}{n_2^2 + \kappa_2^2}$$

Um nun u bzw. v zu erhalten, addiert man $u^2 + v^2$ mit $u^2 - v^2$ bzw. subtrahiert beide von einander. Man erhält dann:

$$2u^2 = \frac{\sqrt{(n_2^2 - \kappa_2^2 - n_1^2 \cdot \sin^2\theta_1)^2 + 4 \cdot n_2^2 \cdot \kappa_2^2}}{n_2^2 + \kappa_2^2} + 1 - \frac{n_2^2 - \kappa_2^2}{(n_2^2 + \kappa_2^2)^2} n_1^2 \sin^2\theta_1$$

$$2v^2 = \frac{\sqrt{(n_2^2 - \kappa_2^2 - n_1^2 \cdot \sin^2\theta_1)^2 + 4 \cdot n_2^2 \cdot \kappa_2^2}}{n_2^2 + \kappa_2^2} - 1 + \frac{n_2^2 - \kappa_2^2}{(n_2^2 + \kappa_2^2)^2} n_1^2 \sin^2\theta_1$$

Dividiert man diese Gleichungen nun durch 2 und zieht davon die Wurzel erhält man schießlich u und v.

$$u = \sqrt{\frac{1}{2} \cdot \left(\frac{\sqrt{(n_2^2 - \kappa_2^2 - n_1^2 \cdot \sin^2\theta_1)^2 + 4 \cdot n_2^2 \cdot \kappa_2^2}}{n_2^2 + \kappa_2^2} + 1 - \frac{n_2^2 - \kappa_2^2}{(n_2^2 + \kappa_2^2)^2} n_1^2 \sin^2\theta_1 \right)} \tag{73}$$

$$v = \sqrt{\frac{1}{2} \cdot \left(\frac{\sqrt{(n_2^2 - \kappa_2^2 - n_1^2 \cdot \sin^2\theta_1)^2 + 4 \cdot n_2^2 \cdot \kappa_2^2}}{n_2^2 + \kappa_2^2} - 1 + \frac{n_2^2 - \kappa_2^2}{(n_2^2 + \kappa_2^2)^2} n_1^2 \sin^2\theta_1 \right)} \tag{74}$$

$$\tag{75}$$

Da sowohl n_2 als auch κ_2 positive Werte annehmen, haben auch u und v positive Werte. Wenn u positiv ist, und $\kappa_2 = 0$ gilt, so wäre $u = \cos\theta_2$ und $v = 0$, und somit die Gleichung $\cos\theta_2 = u + iv$ erfüllt.

Mit den nun bestimmten Werten von u und v ($\cos\theta_2 = u + iv$) können nun die Fresnel-Gleichungen aufgestellt und gelöst werden. Wir betrachten wieder den Fall, dass eine Schicht real (n_1) und eine komplex ($\hat{n}_2 = n_2 + \kappa_2$)ist:

$$r_\parallel = \frac{(n_2 + i\kappa_2)\cos\theta_1 - n_1(u + iv)}{(n_2 + i\kappa_2)\cos\theta_1 + n_1(u + iv)} \tag{76}$$

$$r_\perp = \frac{n_1\cos\theta_1 - (n_2 + i\kappa_2)(u + iv)}{n_1\cos\theta_1 + (n_2 + i\kappa_2)(u + iv)} \tag{77}$$

Teilt man die Gleichungen nun nach real und imaginär Teil auf, so erhält man für $r_\parallel$ und $r_\perp$:

$$r_\parallel = \frac{(n_2\cos\theta_1 - n_1 u) + i(\kappa_2\cos\theta_1 - n_1 v)}{(n_2\cos\theta_1 + n_1 u) + i(\kappa_2\cos\theta_1 + n_1 v)}$$

$$r_\perp = \frac{(n_1\cos\theta_1 - n_2 u + \kappa_2 v) - i((n_2 v + \kappa_2 u)}{(n_1\cos\theta_1 + n_2 u - \kappa_2 v) + i((n_2 v + \kappa_2 u)}$$

Erweitert man nun $r_\parallel$ mit $(n_2\cos\theta_1 + n_1 u) - i(\kappa_2\cos\theta_1 + n_1 v)$ und $r_\perp$ mit $(n_1\cos\theta_1 + n_2 u - \kappa_2 v) - i(n_2 v + \kappa_2 u)$ so erhält man als Nenner eine reale Zahl und man kann beide Reflexionsfaktoren in real und imagiär Teil aufspalten.

$$r_\parallel = \frac{n_2^2\cos^2\theta_1 - n_1^2u^2 - in_2\kappa_2\cos^2\theta_1 - in_2n_1v\cos\theta_1 + in_1u\kappa_2\cos\theta_1 + in_1^2uv +}{(n_2\cos\theta_1 + n_1u)^2 + (\kappa_2\cos\theta_1 + n_1v)^2}$$
$$\frac{+in_2\kappa_2\cos^2\theta + i\kappa_2n_1u\cos\theta_1 - in_1vn_2\cos\theta_1 - in_1^2uv + \kappa_2^2\cos^2\theta_1 - n_1^2v^2}{(n_2\cos\theta_1 + n_1u)^2 + (\kappa_2\cos\theta_1 + n_1v)^2}$$

$$r_\perp = \frac{n_1^2\cos^2\theta_1 - n_2^2u^2 + 2n_2u\kappa_2v - \kappa^2v^2 - in_1n_2v\cos\theta_1 - in_1\cos\theta_1\kappa u + in_2^2uv +}{(n_1\cos\theta_1 + n_2u - \kappa_2v)^2 + (n_2v + \kappa_2u)^2)}$$
$$\frac{+in_2\kappa_2u^2 - i\kappa_2n_2v^2 - i\kappa_2^2uv - in_1\cos\theta_1n_2v - in_1\cos\theta_1\kappa_2v - in_2^2uv - i\kappa_2n_2u^2 +}{(n_1\cos\theta_1 + n_2u - \kappa_2v)^2 + (n_2v + \kappa_2u)^2}$$
$$\frac{+i\kappa_2n_2v^2 + i\kappa_2^2uv - n_2^2v^2 - 2n_2u\kappa_2v - \kappa^2u^2}{(n_1\cos\theta_1 + n_2u - \kappa_2v)^2 + (n_2v + \kappa_2u)^2}$$

Diese Formel lassen sich vereinfachen zu:

$$r_\parallel = \frac{(n_2^2 + \kappa_2^2)\cos^2\theta_1 - n_1^2(u^2 + v^2) + i(2n_1\cos\theta_1(\kappa_2u - n_2v))}{(n_2\cos\theta_1 + n_1u)^2 + (\kappa_2\cos\theta_1 + n_1v)^2} \quad (78)$$
$$r_\perp = \frac{n_1^2\cos^2\theta_1 - (n_2^2 + \kappa^2)(u^2 + v^2) + i(-2n_1\cos\theta_1(n_2v + \kappa_2u))}{(n_1\cos\theta_1 + n_2u - \kappa_2v)^2 + (n_2v + \kappa_2u)^2} \quad (79)$$

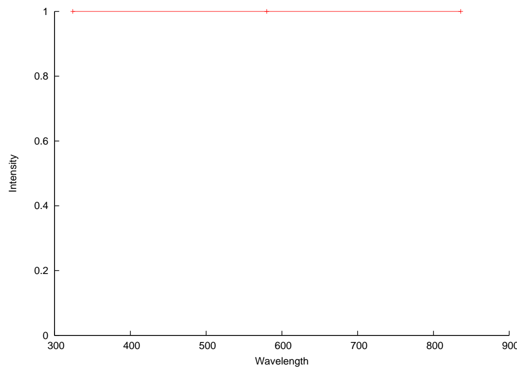
Diese Gleichungen lassen sich leicht in Real- und Imaginärteil aufspalten. Die resultierende Intensität erhält man durch:

$$R_\parallel = |r_\parallel|^2 = [\mathrm{Re}(r_\parallel)]^2 + [\mathrm{Im}(r_\parallel)]^2$$
$$R_\perp = |r_\perp|^2 = [\mathrm{Re}(r_\perp)]^2 + [\mathrm{Im}(r_\perp)]^2$$

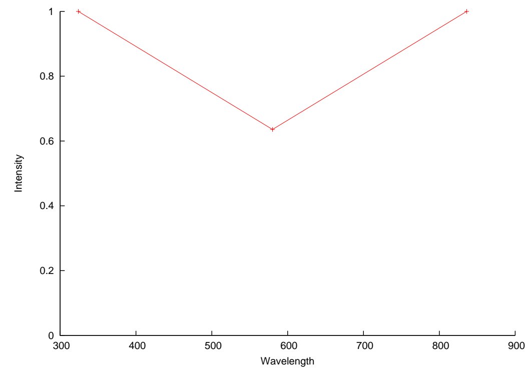
Die Intensität folgt in einfacher Weise aus der Definition des Betrages einer komplexen Zahl.

B Spektrenverzeichnis

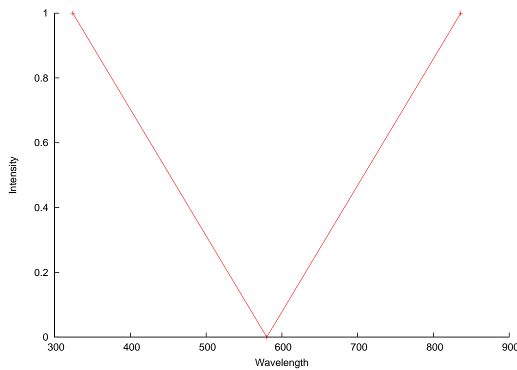
Im folgenden wird eine Auflistung der verwendeten Spektren gegeben. Handelt es sich um reale (eingelesene) Spektren, so ist dies in der Bildunterschrift vermerkt.



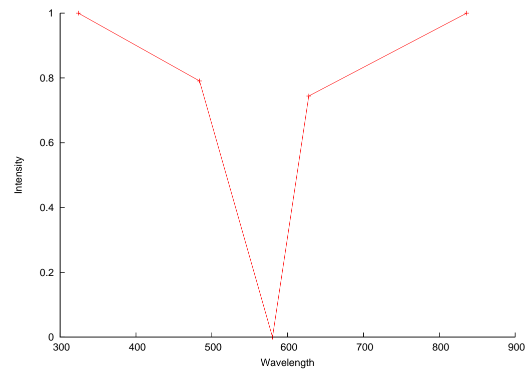
(a) Konstantes Spektrum



(b) Spektrum 1

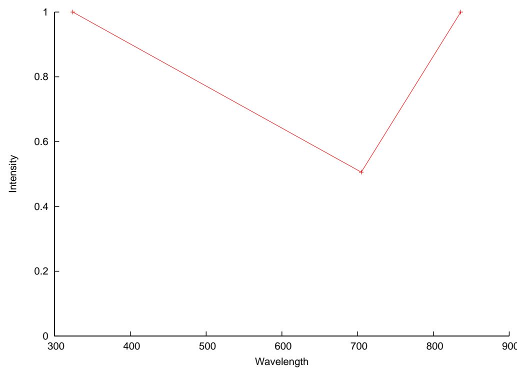


(c) Spektrum 2

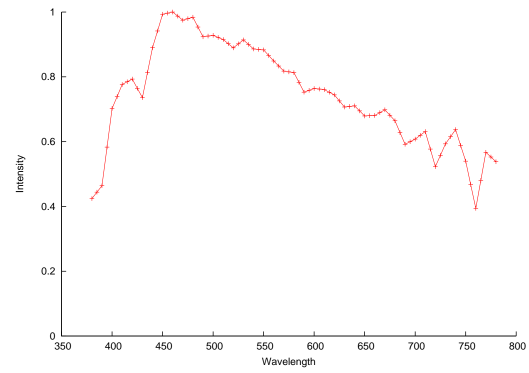


(d) Spektrum 3

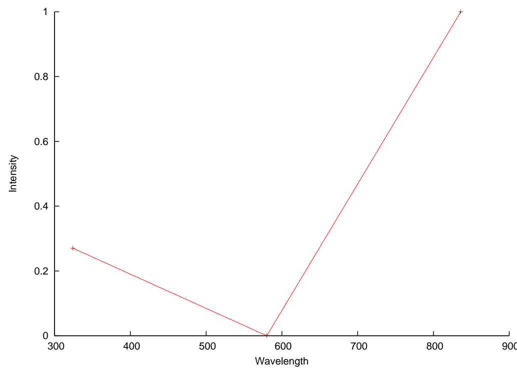
Abbildung 38: Spektren 0-3. Spektrum 0 entspricht dem Illuminant E.



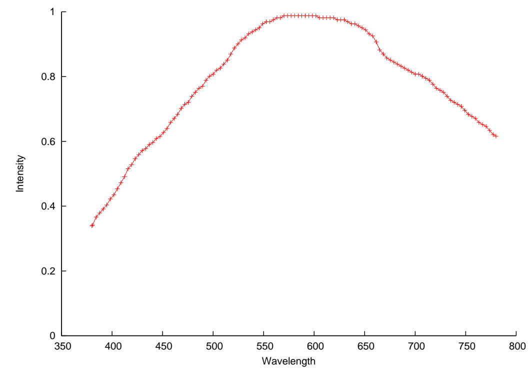
(a) Spektrum 4



(b) Spektrum 5 (D65 Illuminant)



(c) Spektrum 6



(d) Spektrum 7 (Gold)

Abbildung 39: Spektren 4-7. Spektren 5 und 7 wurden aus Daten geparsed. Quelle [CIE].

C Literatur

[Bre07] Dr. T. Breiner. Visualisierung - farbsysteme und -skalen, 2007.

[CIE] CIE. http://www.cie.co.at/publ/abst/datatables15_2004/CIE_sel_colorimetric_tables.xls. Aufgerufen 01.10.2008.

[DK06] Roman Durikovic and Ryou Kimura. Gpu rendering of the thin film on paints with full spectrum. In *IV '06: Proceedings of the conference on Information Visualization*, pages 751–756, Washington, DC, USA, 2006. IEEE Computer Society.

[Don06] Weiming Dong. Rendering optical effects based on spectra representation in complex scenes. In Tomoyuki Nishita, Qunsheng Peng, and Hans-Peter Seidel, editors, *Computer Graphics International*, volume 4035 of *Lecture Notes in Computer Science*, pages 719–726. Springer, 2006.

[EKM01] Sergey Ershov, Konstantin Kolchin, and Karol Myszkowski. Rendering pearlescent appearance based on paint-composition modeling. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum, Proceedings of Eurographics 2001*, pages 227–238, Manchester, UK, 2001. Eurographics, Blackwell.

[ESK96] Jose Encarnacao, Wolfgang Strasser, and Reinhard Klein. *Graphische Datenverarbeitung 1: Geraetetechnik, Programmierung und Anwendung graphischer Systeme*. R. Oldenbourg Verlag, Muenchen, 4. auflage edition, 1996.

[ESK97] Jose Encarnacao, Wolfgang Strasser, and Reinhard Klein. *Graphische Datenverarbeitung 2: Modellierung komplexer Objekte und photorealistische Bilderzeugung*. R. Oldenbourg Verlag, Muenchen, 4. auflage edition, 1997.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[GMN94] Jay S. Gondek, Gary W. Meyer, and Jonathan G. Newman. Wavelength dependent reflectance functions. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 213–220, New York, NY, USA, 1994. ACM.

[Hea91] O. S. Heavens. *Optical Properties of Thin Solid Films.* Dover Pubn Inc., 1991.

[HKY+99] H. Hirayama, K. Kaneda, H. Yamashita, Y. Yamaji, and Y. Monden. Visualization of optical phenomena caused by multilayer films with complex refractive indices. *pg*, 00:128, 1999.

[HKYM01] Hideki Hirayama, Kazufumi Kaneda, Hideo Yamashita, and Yoshimi Monden. An accurate illumination model for objects coated with multilayer films. *Computers & Graphics*, 25(3):391–400, 2001.

[Hof07] Gernot Hoffmann. Cie color space. http://www.fho-emden.de/~hoffmann/ciexyz29082000.pdf, 2007.

[hyp] http://hyperphysics.phy-astr.gsu.edu/Hbase/phyopt/oilfilm.html. Aufgerufen 22.09.2008.

[IMN04] Kei Iwasaki, Keichi Matsuzawa, and Tomoyuki Nishita. Real-time rendering of soap bubbles taking into account light interference. In *CGI '04: Proceedings of the Computer Graphics International*, pages 344–348, Washington, DC, USA, 2004. IEEE Computer Society.

[JMW64] D. B. Judd, D. L. MacAdam, and G. Wyszecki. Spectral Distribution of Typical Daylight as a Function of Correlated Color Temperature. *Journal of the Optical Society of America (1917-1983)*, 54:1031–+, August 1964.

[Jr.04] D. Sim Dietrich Jr. Shader model 3.0 - no limits. Technical report, nVidia, April 2004.

[Kil05] Mark J. Kilgard. Cg and nvidia. Technical report, nVidia, 2005.

[Krö08] Prof. Dr. D. Krömker. Einführung in die computergraphik, 2008.

[Ngu07] Hubert Nguyen. *Gpu gems 3.* Addison-Wesley Professional, 2007.

[NPG05] Mangesh Nijasure, Sumanta N. Pattanaik, and Vineet Goel. Real-time global illumination on gpus. *journal of graphics tools*, 10(2):55–71, 2005.

[nVi05] nVidia. *CG Toolkit User's Manual*, September 2005.

[nVi06] nVidia. *nVidia GPU Programming Guide*, March 2006.

[nVi08a] nVidia. *CG Language Reference*, 2008.

[nVi08b] nVidia. *CG Language Specification*, 2008.

[Ohn99] Dr. Y. Ohno. Osa handbook of optics, volume iii visual optics and vision - chapter for photometry and radiometry, 1999.

[Poy03] Charles Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[Ros05] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[Sad07] Iman Sadeghi. A physically based anisotropic iridescence model for rendering morpho butterflies photo-realistically. http://www.jacobsschool.ucsd.edu/uploads/re/2008/ImanSadeghi-MorphoButterfly.pdf, 2007.

[SAG+05] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen Marschner, Erik Reinhard, Kelvin Sung, William Thompson, and Peter Willemsen. *Fundamentals of Computer Graphics, Second Ed.* A. K. Peters, Ltd., Natick, MA, USA, 2005.

[Sch94] C. Schlick. An inexpensive BRDF model for physically-based rendering . *j-CGF*, 13(3):C/233–C/246, 1994.

[SFDC00] Yinlong Sun, F. David Fracchia, Mark S. Drew, and Thomas W. Calvert. Rendering iridescent colors of optical disks. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 341–352, London, UK, 2000. Springer-Verlag.

[Smi99] Brian Smits. An rgb-to-spectrum conversion for reflectances. *J. Graph. Tools*, 4(4):11–22, 1999.

[SS06] Andrew Stockman and Lindsay Sharpe. http://cvision.ucsd.edu/database/data/cmfs/ciexyz64_1.txt, 2006. Aufgerufen am (05.09.2008).

[Sun06] Yinlong Sun. Rendering biological iridescences with rgb-based renderers. *ACM Trans. Graph.*, 25(1):100–129, 2006.

[SW08] Yinlong Sun and Qiqi Wang. Interference shaders of thin films. In *Computer Graphics Forum*. The Eurographics Association and Blackwell Publishing Ltd., 2008.

[SWND05] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition) (OpenGL)*. Addison-Wesley Professional, 2005.

[thi] http://www.calctool.org/CALC/phys/optics/thin_film. Aufgerufen 05.09.2008.

[War92] Gregory J. Ward. Measuring and modeling anisotropic reflection. In *SIGGRAPH '92: Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 265–272, New York, NY, USA, 1992. ACM.