



Johann Wolfgang Goethe-Universität  
Frankfurt am Main

---

Fachbereich Biologie und Informatik

## **Diplomarbeit**

# **Generierung, Verwaltung und Verarbeitung von Bidirectional Texture Functions (BTFs) zur photorealistischen Bildgenerierung mit komplexen Oberflächenmodellen**

**Jing Qian**

Graphische Datenverarbeitung

Januar, 2006

Betreuer: Prof Dr.-Ing. Detlef Krömker

Zweit-Betreuer: Prof. Dr.-Ing. Wolfgang Müller

## **Erklärung**

Hiermit versichere ich, Jing Qian, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Frankfurt am Main, 30. Januar 2006

Jing Qian

## **Kurzfassung**

Diese Diplomarbeit beschäftigt sich sowohl mit der Akquisition, der Verwaltung und der winkelabhängigen, spektralen Reflexionsfunktion BRDF (*Bidirectional Reflectance Distribution Function*) und BTFs (*Bidirection Texture Functions*) von einigen ausgewählten realistischen Materialoberflächen bei fester Beleuchtung, als auch der Generierung der BTFs aus vorhandenen BRDFs und BTFs und der Anwendung der Beschreibung von Oberflächen mittels BRDFs und BTFs bei Erzeugungen von 3D-Szenen. Im Rahmen dieser Diplomarbeit werden Konzepte für eine effektivere Nutzung von BTFs in der photorealistischen Bilderzeugung entwickelt und prototypisch umgesetzt. Der Fokus liegt dabei auf einer vereinfachten Synthese von BTFs aus vorhandenen BRDF- und BTF-Daten, sowie in einer effizienten Nutzbarmachung dieser Informationen für Rendering-Prozesse.

**Schlüsselwörter:** BRDF, BTF, Beleuchtungsmodell, Shader, BMRT, RenderMan, *Image-based-rendering*, SVD-Zerlegung, PCA-Zerlegung, LZW-Algorithmus, Median-Cut-Algorithmus,

## **Danksagungen**

Während der Erstellung dieser Arbeit haben mir viele Menschen direkt oder indirekt geholfen, allen voran Prof. Detlef Krömker, dem ich hier für seinen Einsatz in Vorlesungen und Übungen an der Universität Frankfurt herzlich danken möchte. Erst durch seine Vorlesungen hat er mein besonderes Interesse an dem Gebiet der graphischen Datenverarbeitung geweckt. Auch Prof. Wolfgang Müller, der die Zweitbetreuung dieser Arbeit übernommen hat, möchte ich für seine sehr gute und geduldige Betreuung der Arbeit danken. Er hat dabei immer sowohl nützliche Vorschläge zur Lösung verschiedener Probleme, als auch interessante Ideen gegeben. Zur Entwicklung von Konzepten und Anwendungen haben auch Tobias Breiner, Frederik Naskrent, Daniel F. Abawi, Ashraf Abu Baker (Institut für GDV der Universität Frankfurt) und Dr. Paul Grimm (Fachhochschule Erfurt) durch Diskussion verschiedene Anregungen beigetragen; außerdem natürlich viele Freunde wie Dong Han, Kunkun Wang, Lei Shang und Xing Zhao (Technische Universität Darmstadt). Ein besonderer Dank gebührt Thomas Frank (OLED-Labor, Merck AG) für seine Hilfe bei den optischen Erkenntnissen.

Abschließend danke ich auch besonders herzlich meinen Eltern für ihr Verständnis und ihre Unterstützung während des Studiums.

*Frankfurt a. M., im Januar 2006*

*Jing Qian*

## **Kontaktadresse**

Wenn Sie Fragen, Anregungen oder Tipps zu dieser Arbeit haben stehen Ihnen folgende Möglichkeiten der Kontaktaufnahme offen:

**smail:** Jing Qian

Vilbeler Landstr. 77

60388 Frankfurt am Main

Deutschland

**email:** [jingqian@hotmail.com](mailto:jingqian@hotmail.com)

**phone:** +49(0)6109-504961

Ich bin für jeden Kommentar dankbar, sogar für eine kleine Bestätigung, dass irgendjemand ausser meinen Betreuern diese Arbeit gelesen hat.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung.....</b>	<b>1</b>
<b>2</b>	<b>Grundlagen.....</b>	<b>4</b>
2.1	Radiometrie .....	4
2.1.1	Bogenwinkel und Raumwinkel.....	4
2.1.2	Bestrahlungsstärke (Irradiance) .....	5
2.1.3	Strahldichte (Radiance).....	5
2.2	Interaktion mit Materie .....	6
<b>3</b>	<b>BRDF-Modell .....</b>	<b>7</b>
3.1	Definition von BRDFs .....	7
3.2	Eigenschaften von BRDFs.....	7
3.3	Rendering-Gleichung.....	8
3.4	Analytical-BRDFs und einige Modelle.....	8
3.4.1	Definition von Analytical-BRDFs .....	8
3.4.2	Zwei Klassen von BRDFs.....	9
3.4.3	Anteile einer BRDF.....	10
3.4.4	Lambert-Modell .....	10
3.4.5	Phong- und Blinn/Phong-Modell.....	11
3.4.6	Cook/Torrance-Modell.....	13
3.4.7	Andere BRDF-Modelle.....	16
3.5	Measured-BRDFs .....	16
3.5.1	Definition von measured-BRDFs .....	16
3.5.2	Akquisition von BRDFs.....	17
3.5.3	Lineare Faktorisierungen von BRDFs .....	24
3.5.4	Anwendungen von measured-BRDFs.....	36
3.6	Zusammenfassung.....	43
<b>4</b>	<b>BTF-Akquisition .....</b>	<b>44</b>
4.1	Definition von BTFs .....	44
4.2	Akquisition von BTFs .....	45
4.2.1	BTF-Datenbank von CURet .....	45
4.2.2	BTF-Datenbank von Bonn.....	46
4.2.3	Andere BTF-Datenbanken im Internet.....	47
<b>5</b>	<b>BTF-Kompression.....</b>	<b>49</b>
5.1	Zwei Darstellungen von BTFs .....	49
5.2	Chained Matrix Factorization (CMF) für BTF .....	50
5.3	Principal Component Analysis(PCA) .....	52
5.4	Singular Value Decomposition (SVD).....	53

5.5 Per-view Matrix Factorization (PVMF).....	54
5.6 Per-cluster Matrix Factorization (PCMF).....	55
5.7 Die Vergleichliste der Methoden.....	56
<b>6 BTF-Synthese .....</b>	<b>57</b>
6.1 Definition und Bedeutung der BTF-Synthese.....	57
6.2 Algorithmen der Synthese von 2D-Texturen .....	57
6.2.1 Tiling der Texture.....	57
6.2.2 Pixelweise Synthese der Textur .....	58
6.2.3 Schnelle pixelweise Synthese der Textur.....	61
6.2.4 Blockweise Synthese der Texture .....	64
6.3 BTF-Synthese mittels 2D-Algorithmen.....	66
6.4 Andere Methoden bei BTF-Synthese.....	67
<b>7 Implementierung von BTFs bei Anwendungen.....</b>	<b>69</b>
7.1 Das Konzept der Implementierung .....	69
7.2 Der anpassende Datenträger von BTFs.....	69
7.3 Wiederaufbau der BTF-Bilder .....	71
7.4 Kompression der TIFF-Bilder mittels LZW-Algorithmen .....	73
7.5 Spezifikation im Modeler von RenderMan.....	75
7.6 Surface-Shader von RenderMan .....	77
7.6.1 Die Berechnung von vier Winkeln.....	78
7.6.2 Repräsentation von BTFs im Shader .....	80
7.6.3 Textur-Abbildung (texture mapping).....	82
7.7 Die gerenderten Bilder einer Glühbirne.....	83
7.8 Rendern mit den originalen BTF-Bildern .....	84
<b>8 Generierung von BTFs .....</b>	<b>87</b>
8.1 Bedeutung der Generierung von BTFs .....	87
8.2 Median-Cut-Algorithmus.....	87
8.3 Indexbilder und BRDF-Block.....	90
8.4 Im Vergleich zu vorherigen Rendern-Methoden.....	94
8.5 Einsatz neuer BTFs beim Rendern .....	96
8.5.1 Erzeugung von BTFs verschiedener Farben.....	96
8.5.2 Rendern der neuen BTFs in RenderMan .....	98
8.6 Offene Probleme .....	99
<b>9 Zusammenfassung und Ausblick.....</b>	<b>101</b>
9.1 Zusammenfassung.....	101
9.2 Ausblick .....	102
<b>Anhang A Anleitung zum Anwendungsprogramm.....</b>	<b>103</b>
<b>Literaturverzeichnis.....</b>	<b>108</b>

# Kapitel 1

## Einleitung

Die Generierung photorealistischer Bilder aus künstlichen Szenen stellt eine der anspruchsvollsten Aufgaben der 3D-Computergraphik dar [Möller99]. Die physikalisch korrekte Simulation der Reflexionseigenschaften natürlicher Materialien ist eine der zentralen Problemstellungen in diesem Kontext. Neben allgemeinen Kriterien wie Farbe, Transparenz, Rauigkeit etc. verwendet man dazu in letzter Zeit in immer stärkerem Maße komplexe Oberflächenmodelle in Form der BRDF (*Bidirectional Reflectance Distribution Function*) [Wynn00], die das Abstrahlverhalten realer Oberflächen sehr detailliert beschreiben und auch anisotropische Effekte umfassen können [Latta02].

Unter grober Skalierung, wobei lokale Variationen einer Materialoberfläche Subpixel sind und die lokale Intensität uniform ist, kann das äußerliche Erscheinungsbild (*appearance*) durch BRDF charakterisiert werden. Jedoch unter feiner Skalierung, bei der lokale Variationen einer Materialoberfläche zu Variationen der lokalen Intensitäten führen, kann es durch BRDF nicht mehr nahtlos beschrieben werden. In einem solchem Fall von fein strukturierten Oberflächen werden entsprechend BTFs (*Bidirection Texture Functions*) als Erweiterung der BRDF vergleichbar zu Texturen verwendet. Die von Kristin J. Dana [Dana99] eingebrachte BTF ist zuerst eine sechsdimensionale Funktion, die entweder annähernd eine geordnete Textur von *per-Texel*-BRDFs oder eine Menge von gemessenen Textur-Bildern in verschiedene Kamerarichtungen und Lichtrichtungen repräsentiert.

Auf der einen Seite eröffnet die Verwendung der BRDF und BTF die Möglichkeit, echte Materialien mit komplexen Reflexionsverhalten darzustellen und rechnerisch verwertbar zu machen. Auf der anderen Seite behebt BTF das Problem ärmlicher Texturen und schwacher Lichteffekte in traditionellen 3D-Modellen („Sie sehen echt aus und nicht nur glaubhaft!“ von Attila Neumann, Projektkoordinator des EU-Projekts *Realreflect*).

Aber die Beschreibung von Oberflächen mittels BRDFs und BTFs bringt verschiedene Probleme mit sich. Eine erste Schwierigkeit bildet die Erzeugung solcher Oberflächenmodelle. Die Akquisition von BRDFs und BTFs auf Grundlage realer Material-Muster ist aufwendig und bedarf spezieller und technisch komplexer Aufnahmegeräte. Einmal erstellte Modelle lassen sich nicht parametrisieren oder variieren. Die in einem solchen Akquisitionsprozess generierten Datenmengen sind



sehr groß und auf solchen Daten basierende komplexe BRDF- und BTF-Modelle bereiten beim Rendering vielfältige Probleme, insbesondere im Bezug auf den notwendigen Speicherplatz.

Das Hauptziel dieser Arbeit steht daher in der Entwicklung eines Algorithmus, der in der Lage sein sollte, gemessene BTFs zu analysieren und zu verwalten, die neuen BTFs aus vorhandenen BRDF- und BTF-Daten zu generieren und sie in einer effizienten Nutzbarmachung mit den Informationen für Rendering-Prozesse anzuwenden.

Die Struktur der Arbeit sieht wie folgt aus:

- Kapitel 2 dient der Erklärung der grundlegenden Erkenntnisse von Radiometrie und Licht-Materie-Interaktion, die zum Verständnis von BRDFs gebraucht werden und teilweise auch eine umgewandelte Formel der BRDFs anbieten.
- In Kapitel 3 werden zwei verschiedene Klassen von BRDF-Modellen vorgestellt: *Analytical*- und *Measured*-BRDF. *Analytical*-BRDF ist einfach implementierbar und funktioniert sehr gut in Echt-Zeit. In der Vergangenheit wurden schon viele Modelle wie z. B. Phong-, Blinn/Phong-, Cook/Torrance-[Cook81] und Ward-Modell [Ward92] entwickelt. *Measured*-BRDF basiert im Prinzip auf *Image-based-Rendering*. Zur Beschreibung anisotroper Materialien können *Measured*-BRDFs als zusätzliche Beleuchtungsmodelle in Renderer integriert werden. Dazu wird aber mehr Speicherplatz benötigt, deshalb werden einige Zerlegungsmethoden diskutiert, die später zur Kompression von BTFs auch dienen können, wenn sich eine BTF als eine Texture von BRDFs ansehen läßt.
- In Kapitel 4, 5 werden wir detailliert betrachten, wie und wo BTFs der verschiedenen Datenquellen akquisitiert wurden, welche Datenquelle aus welchem Grund meistens in unserer Arbeit verwendet wird; mit welchen Methoden und wie weitere BTFs mit riesigen Datenmengen komprimiert werden können, was es ermöglicht, BTFs beim Rendern in Echt-Zeit einzusetzen.
- Kapitel 6 beschäftigt sich mit der Synthese von BTFs. Wegen der Größe des gemessenen Samples hat das Textur-Bild begrenzte Auflösung (z. B. die Textur-Bilder von Bonn [BonnTex06] haben eine Auflösung von 256 x 256 Pixel), die selbstverständlich Schwierigkeit hat, die große Oberfläche von Objekten ohne Verzerrung zu bedecken. Die Kernidee hier ist eine BTF als eine Textur von *per-Texel*-ABRDFs zu beschreiben, damit die Synthesemethoden, die eigentlich für 2D-Texturen entwickelt wurden, bei BTF-Synthese auch anwendbar sind.

- In Kapitel 7 verwenden wir BTF in Shader von RenderMan [Upstill92]. Dabei stellen wir zuerst die Struktur und Arbeitsweise eines *Surface*-Shader vor und betrachten schrittweise, wie das Reflexionsverhalten der einzelnen Punkte von Oberflächen durch BTF-Daten in Shader implementiert wird.
- In Kapitel 8 bringen wir ein interessantes Thema ein: wie neue BTFs von vorhandenen BRDFs und BTFs generiert werden können. Die Akquisition von BTFs auf der Grundlage realer Material-Muster ist aufwendig und bedarf spezieller und technisch komplexer Messgeräte. Einmal erstellte Modelle lassen sich nicht parametrisieren oder variieren. Aus dem Grund wollen wir einen weiteren Weg neben Akquisition finden. Unsere Idee ist zuerst die Muster-Struktur aus Textur-Bildern einer BTF mittels Median-Cut-Algorithmus auszuziehen und dann so entstandene Indexbilder mit anderen BRDFs zu verbinden, damit vielartige BTFs mit unterschiedlichen Farben erzeugt werden. Ein weiterer Vorteil dieser Technik ist, dass wir im Vergleich zur PCA- oder SVD-Kompression mehr Speicherplatz gewinnen können.
- Zum Abschluss wird in Kapitel 9 eine Anwendung präsentiert, die die Funktionen der Anzeige von BTF-Bildern, der Kompression mittels SVD und der Rekonstruktion von BTFs, sowie die Generierung von BTFs mittels Median-Cut-Algorithmus zusammenstellt.

# Kapitel 2

## Grundlagen

### 2.1 Radiometrie

#### 2.1.1 Bogenwinkel und Raumwinkel

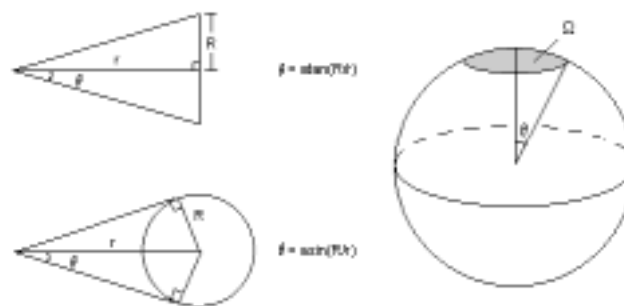


Abbildung 2.1: Die Definition des Raumwinkels [Moon04]

In der Ebene wird ein Winkel üblicherweise durch ein Bogenmaß gemessen [Moon04]. Das Bogenmaß misst einen Winkel als die Länge des auf dem Einheitskreis überdeckten Kreisbogens. Analog zum Bogenwinkel wird ein Raumwinkel durch ein Flächenmaß gemessen. Das Flächenmaß misst einen Raumwinkel als die Größe der auf der Einheitskugel überdeckten Kugeloberfläche.

$$W = \frac{A}{r^2} (sr) \quad (1)$$

Seine Einheit wird in  $sr = \text{steradian}$  angegeben.

Wir können auch den differentiellen Raumwinkel durch den differentiellen Azimut- und Deklinationswinkel repräsentieren (Siehe Abbildung 2.1).

$$dA = r^2 \sin \theta d\theta d\phi \quad (2)$$

Wenn wir beide Seiten der Gleichung mit dem Faktor  $1/r^2$  multiplizieren, erhalten wir die Gleichung für den Raumwinkel

$$dw = \sin \theta d\theta d\phi \quad (3)$$

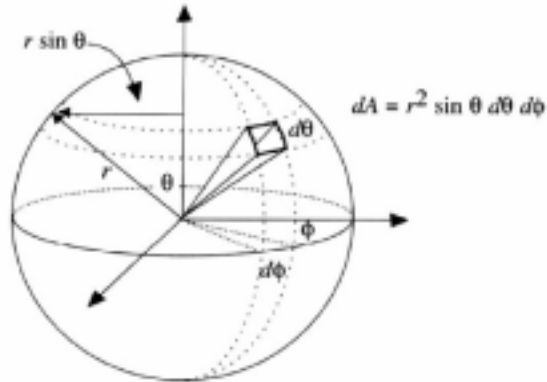


Abbildung 2.2: Die differentielle Fläche [Cohen93]

### 2.1.2 Bestrahlungsstärke (Irradiance)

Die Bestrahlungsstärke beschreibt, wieviel Strahlungsleistung von einer einheitlichen Lichtquelle gestrahlt wird oder auf eine bestimmte Fläche auftritt.

Die Einheit der Bestrahlungsstärke ist  $Watt/m^2$ . Sie nimmt mit dem Quadrat der Entfernung zur Strahlquelle ab. Ein gutes Beispiel einer Strahlungsquelle ist die Sonne, deren Bestrahlungsstärke  $500 Watt/m^2$  beträgt.

### 2.1.3 Strahldichte (Radiance)

Strahldichte beschreibt, wieviel Bestrahlungsstärke in einem bestimmten Raumwinkel einer Oberfläche ankommt. Die Einheit ist Irradiance pro einheitlichen Raumwinkel. Sie ist von der Entfernung zur Strahlungsquelle unabhängig.

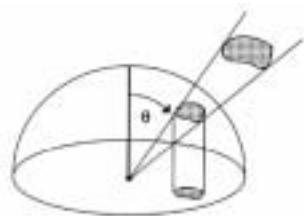


Abbildung 2 3: Raumwinkel in Unabhängigkeit von Entfernung [Cohen93]

$$L = \frac{E}{sr}$$

$$L = \frac{Watt / m^2}{sr} \quad (4)$$

$$L = \frac{Watt}{m^2 sr}$$

Wir bezeichnen die Strahldichte mit  $L(\theta, \phi)$ , wobei  $\phi$  und  $\theta$  die Azimut- und Deklinationswinkel sind.

## 2.2 Interaktion mit Materie

Sobald das Licht von einer Lichtquelle abgestrahlt wurde und auf die Oberfläche eines Materials trifft, wird die Strahlung zum Teil reflektiert, zum Teil vom Material absorbiert und durch das Material transmittiert. Die Formel hierzu sieht folgendermaßen aus:

$$\textit{light incident at surface} = \textit{light reflected} + \textit{light absorbed} + \textit{light transmitted}$$

In dieser Arbeit werden wir *nur* den Idealfall diskutieren, in dem die Absorption und die Transmission benachlässigt werden.

Die Strahlungsleistung auf der differentialen Fläche  $dA$  über den Raumwinkel  $d\omega$  wird berechnet aus:

$$E_i = \int_0^{2\pi} \int_0^{\frac{\pi}{2}} L_i(\theta, \phi) d\theta d\phi \quad (5)$$

In dieser Gleichung betrachten wir eine Hemisphäre. Aus diesem Grund müssen wir für den Deklinationswinkel lediglich von 0 bis  $\frac{\pi}{2}$  integrieren.

Die Beziehung zwischen der einfallenden Strahldichte  $L_i$  und der reflektierten Strahldichte  $L_r$  kann wie folgt formuliert werden:

$$L_r = L_i(N \cdot L) = L_i \cdot \cos \theta \quad (6)$$

Wenn wir die beiden Gleichungen zusammenfassen, dann erhalten wir:

$$E_r = \int_0^{2\pi} \int_0^{\frac{\pi}{2}} L_i(\theta, \phi) \cos \theta d\theta d\phi \quad (7)$$

Eine Funktion, welche die Reflexionseigenschaft eines Punktes auf der Materialoberfläche beschreibt, kann bezeichnet werden als:

$$P(V, L) = \frac{dL_r}{dE_i} \quad (8)$$

$$P(V, L) = \frac{dL_r(V)}{L_i(L) \cos \theta d\omega}$$

wobei  $L_r$  die reflektierte Strahldichte und  $L_i$  die einfallende Strahldichte ist.  $V$  und  $L$  sind die Richtung von Kamera und Lichtquelle. Es wird das auf die Fläche treffende Licht gemessen.

Die Funktion  $P(V, L)$  ist eine der umgewandelten Formen von BRDF, die wir im nächsten Kapitel intensiv ermitteln werden.

# Kapitel 3

## BRDF-Modell

### 3.1 Definition von BRDFs

BRDF [Nicodemuß77] ist die Abkürzung von *Bidirectional Reflectance Distribution Function* (Bidirektionale Reflexionsverteilungsfunktion). Wie wir im letzten Kapitel bereits erwähnt haben, beschreibt eine BRDF die optischen Materialeigenschaften einer Oberfläche und wie Licht einer gegebenen Wellenlänge von einer Oberfläche in einen bestimmten Winkel reflektiert wird.

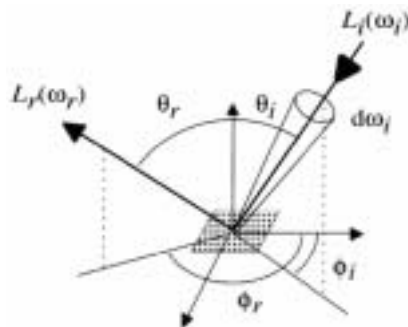


Abbildung 3.1: BRDF-Modell [Pfister99]

### 3.2 Eigenschaften von BRDFs

Für BRDFs gelten einige wichtige Eigenschaften:

1. Helmholtz-Prinzip

Die BRDF bleibt unverändert, auch wenn man die Richtung der einfallenden und ausfallenden Strahlung umkehrt.

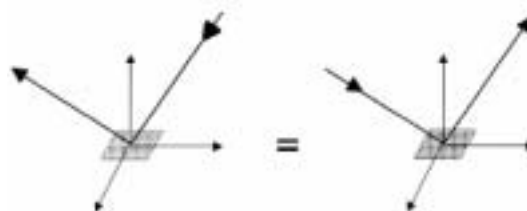


Abbildung 3.2: Helmholtz-Prinzip [Wynn00]

## 2. Energieerhaltung

Wegen der Licht-Materie-Interaktion kann die Summe der Strahlung, die in alle Richtungen reflektiert wird, nicht größer sein als die einfallende Strahlung, da die Materie einen Teil der Strahlung absorbiert.

## 4. Im Allgemeinen gilt Anisotropie

Die BRDF ändert sich demnach bei Drehung der Fläche um ihre Normale nicht.

## 3. Superposition

Wird Licht aus einer neuen Richtung zugefügt, hat das keinen Einfluß auf das aus anderen Raumrichtungen reflektierte Licht. Die Reflexionen haben lineare Charakteristik und können linear überlagert werden.

## 3.3 Rendering-Gleichung

Die Rendering-Gleichung gibt ein Modell für Lichtaustausch an und wird formuliert als

$$L_r(X, V) = L_e(X, V) + \int_{\omega_i \in \Omega(N)} f_r(X, L, V) \cdot L_i(X, L) \cdot \cos(N \cdot L) d\omega_i \quad (9)$$

Wobei

$X$  : betrachteter Punkt,

$L$  : Kugelkoordinaten des einfallenden Lichts,

$V$  : Kugelkoordinaten des reflektierten Lichts,

$L_r(X, V)$  : Strahldichte des von  $X$  ausgesendeten Lichts in Richtung  $V$  ,

$L_e(X, V)$  : Strahldichte des von  $X$  emittierten Lichts in Richtung  $L$  ,

$L_i(X, L)$  : Strahldichte des in  $X$  einfallenden Lichts in Richtung  $L$  ,

$f_r(X, L, V)$  : BRDF des Punktes  $X$  zwischen den Kugelkoordinaten  $L$  und  $V$  ,

$N$  : die Normale des Punktes  $X$  ,

$\Omega(N)$  : die  $N$  zugewandte Hemisphäre um  $X$  .

## 3.4 Analytical-BRDFs und einige Modelle

### 3.4.1 Definition von Analytical-BRDFs

*Analytical*-BRDF ist ein prozedurales Modell, welches eine mathematische Form mit anpassbaren Parametern ist und die Reflexion in einen bestimmten Punkt der Materialoberfläche bei Rendern simuliert. Eine solche BRDF gilt nicht als physikalisch plausible Definition, sollte aber in den meisten Fällen für

Computer-Graphik gut funktionieren, bei der die Absorption und Transmission der Strahlungsenergie vernachlässigt werden.

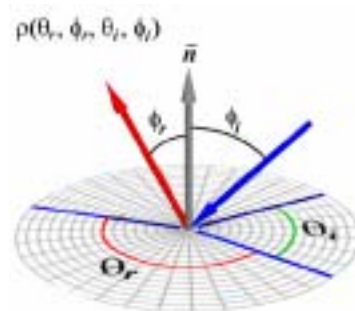


Abbildung 3.3: Die Definition von BRDF [Pfister99]

### 3.4.2 Zwei Klassen von BRDFs

Die BRDFs können in zwei Klassen eingeteilt werden: für anisotrope und isotrope Materialien. Anisotropie bedeutet, dass die Reflexionsfunktion nicht mehr gleich bleibt, wenn die Fläche um den Azimutwinkel gedreht wird und die beiden Deklinationswinkel beibehalten werden, z. B. gebürstetes Metall, Stoffe, Haare, Fell, etc.

Sollte die BRDF durch eine 4D-Funktion formuliert werden, kann sie bei isotropen Materialien auf 3D reduziert werden: mit den zwei Deklinationswinkeln und der Differenz zwischen den beiden Azimutwinkeln. (Siehe Abbildung 3.4)

$$f_{isotrop}(\theta_i, \phi_i, \theta_r, \phi_r) = f(\theta_i, \theta_r, \phi_i - \phi_r) \quad (10)$$

wobei

1. Einfallswinkel  $\theta_i$
2. Einfallswinkel  $\phi_i$
3. Ausfallswinkel  $\theta_r$
4. Ausfallswinkel  $\phi_r$

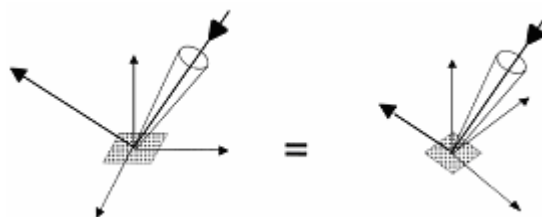


Abbildung 3.4: Isotrope Materialoberfläche



### 3.4.3 Anteile einer BRDF

BRDF setzt sich aus drei Teilen zusammen

$$\text{BRDF} = \text{spiegelnde} + \text{diffuse} + \text{glänzende Reflexion}$$

Bei ideal spiegelnder Reflexion ist der Reflexionswinkel gleich dem Einfallswinkel. (Siehe Abbildung 3.5(b))

$$\begin{aligned}\theta_i &= \theta_r \\ \phi_i &= \phi_r \pm \pi\end{aligned}\quad (11)$$

Bei ideal diffuser Reflexion bleibt die BRDF über die Hemisphäre konstant. (Siehe Abbildung 3.5(a))

$$f(\theta_i, \phi_i, \theta_r, \phi_r) = \int_0^\pi \frac{kd}{\pi} d\theta \quad (12)$$

$$f(\theta_i, \phi_i, \theta_r, \phi_r) = kd$$

wobei  $kd$  eine Konstante ist, welche den Anteil der diffusen Reflexion beschreibt.

Glänzende Reflexion entsteht durch die Kombination verschiedener Effekte, die in einfachen Modellen wie Blinn-Modell nicht erfasst werden. (Siehe Abbildung 3.5(c))

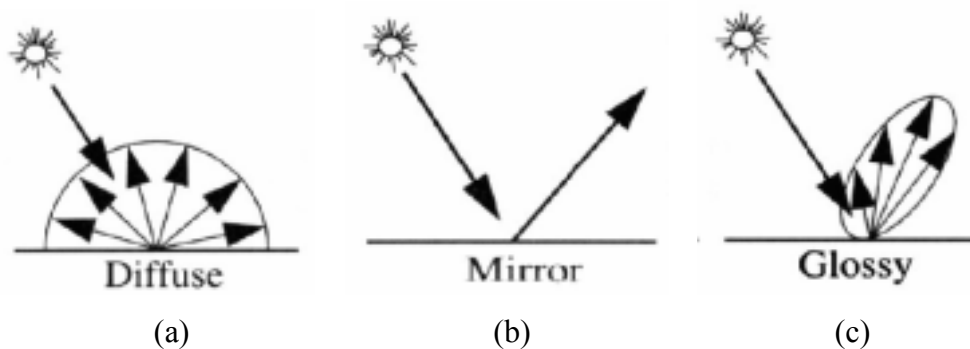


Abbildung 3.5(a), (b), (c): Die diffuse, spiegelnde, glänzende Reflexion [Möller99]

### 3.4.4 Lambert-Modell

Wollen wir eine Oberfläche mit hoher Geschwindigkeit, aber mit geringerer Qualität der Szene rendern, so ist ein einfaches Beleuchtungsmodell sehr nützlich. Dieses Modell wird Lambert-Modell [Hill03] genannt und beschreibt nur den diffusen Teil des reflektierten Lichts. In diesem Fall hängt die ausgesendete Leuchtdichte nicht von der Richtung ab, in der das Licht durch eine Kamera oder durch ein menschliches Auge aufgenommen wird, sondern nur vom Einfallswinkel, bzw. dem Winkel zwischen der Normalen der Fläche und der einfallenden Strahlung [Nayar89]. An jedem Punkt wird die gleiche Menge an Licht in alle Richtungen gestreut und in jede

betrachtete Richtung wird die gleiche Menge an Licht aus allen Richtungen genommen. Deshalb enthält die BRDF die einzige Konstante  $kd$ , die den Anteil der diffusen Reflexion beschreibt. Abbildung 3.6 zeigt eine mit dem Lambert-Modell gerenderte Kugel.



Abbildung 3.6: Eine gerenderte Kugel mit Lambert-Modell [Gebhardt00]

### 3.4.5 Phong- und Blinn/Phong-Modell

Der Nachteil des Lambert-Modells ist, dass der Teil der spiegelnden Reflexion des Licht noch nicht betrachtet wird. Das heißt, wir bekommen immer das selbe Bild von einer Oberfläche, unabhängig davon, an welcher Position unsere Kamera aufgestellt wird. Für viele Materialoberflächen bedeutet dies, dass der realistische Glanzeffekt verloren geht. Dieses Modell passt aber in den meisten Fällen nicht zu den physikalischen Theorien, da das Energieerhaltungssatz nicht immer eingehalten werden kann, wenn wir z. B. die spiegelnde Reflexion zuviel einsetzen, kann das reflektierte Licht mehr als das einfallende Licht sein.

$$f_{Phong-spec}(X, L, V) = k_S \frac{(R \cdot V)^{M_{shi}}}{(N \cdot L)} M_{shi} \quad (13)$$

$$R = 2(N \cdot L) \cdot N - L \quad (14)$$

wobei,

$k_S$ : eine Konstante, die den Anteil der spiegelnden Reflexion bezeichnet.

$X$ : betrachteter Punkt,

$L$ : die Richtung vom  $X$  Punkt auf der Oberfläche zur Lichtquelle,

$V$ : die Richtung vom Betrachter zum  $X$  Punkt auf der Oberfläche,

$N$ : die Normale der Oberfläche im  $X$  Punkt

$R$ : die reflektierte Richtung, die der gespiegelten Richtung über  $N$  entspricht.

$M_{shi}$ : eine Variable, die den Glanzeffekt beeinflusst.

Wenn wir jetzt die diffuse Reflexion und den Glanzlichteffekt zusammenfassen, indem wir das Lambert- und Phong-Modell linear addieren, können wir es vermeiden, dass manche Materialien etwas unrealistisch aussehen.

$$f_{Phong-spec, Lambert-diff}(X, L, V) = kd + k_S \frac{(R \cdot V)^{M_{shi}}}{(N \cdot L)} \quad (15)$$

In der Praxis beobachtet man, dass sich der Teil der spiegelnden Reflexion verkleinert,

wenn sich der Winkel zwischen der Licht- und Betrachterrichtung vergrößert. Umgekehrt wird der Wert der Intensität maximal, wenn dieser Winkel Null Grad beträgt. Dazu gibt Blinn eine weitere, bessere Variante des Beleuchtungsmodells [Blinn77], bei dem er einen Halbvektor zwischen der Licht- und Betrachterrichtung verwendet.

$$H = \frac{L+V}{|L+V|} \quad (16)$$

Die BRDF des Blinn/Phong-Modells sieht wie folgt aus:

$$f_{\text{Blinn-Phong-spec,Lambert-diff}}(X, L, V) = kd + ks \cdot \frac{(N \cdot H)^{M_{shi}}}{(N \cdot L)} \quad (17)$$

Eine ähnliche Beziehung zwischen den beiden Varianten kann auch durch folgende Gleichung formuliert werden:

$$(R \cdot V)^{M_{shi}} \approx (N \cdot H)^{4M_{shi}} \quad (18)$$

Die Variable  $M_{shi}$  beschreibt die Intensitätstufe der Materialoberfläche. Je größer  $M_{shi}$  in der Gleichung eingesetzt wird, desto glänzender sieht die Oberfläche aus.

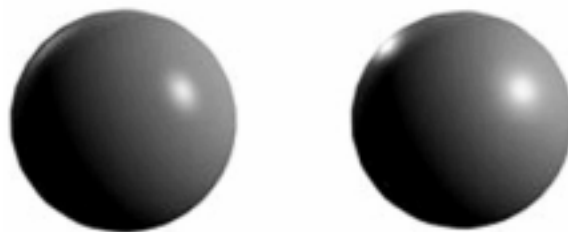


Abbildung 3.7: Zwei durch Phong- und Blinn/Phong-Modell gerenderte Kugeln [Calkins]

(Links: Phong-Modell. Rechts: Blinn/Phong-Modell)

In der realen Welt gibt es immer Strahlung, die von einer Fläche reflektiert wird und auf eine andere Fläche auftrifft. Aber in Phong- und Blinn/Phong-Modellen wird das Licht direkt auf die Oberfläche der Objekte gestrahlt und solch indirekt reflektierte Strahlungen werden weder in der diffusen noch in der spiegelnden Reflexion mit einberechnet. Um diese zu simulieren, werden wir uns im nächsten Absatz auf die Reflexionen zwischen den Flächen untereinander konzentrieren.

### 3.4.6 Cook/Torrance-Modell

Das von Cook und Torrance vorgestellte Modell [Cook81] geht davon aus, dass die Oberfläche eines Materials aus einer Menge diskreter kleiner Flächen (*Microfacetten*) besteht. Jede kleine Fläche aus der Menge wird als eine ideale glatte Fläche behandelt und reflektiert das einfallende Licht gemäß ihrer jeweiligen Neigung. Die unterschiedlichen Rauheiten der Oberfläche können durch die Neigung der einzelnen Flächen implementiert werden, d.h. die Oberfläche, deren kleine Flächen relativ niedrige Neigungen haben, sieht relativ glatt aus.

Cook und Torrance nutzen die folgende Gleichung zur Berechnung einer BRDF:

$$f(V, L) = F(V, L) \frac{D(V, L)G(V, L)}{(N \cdot V)(N \cdot L)} \quad (19)$$

wobei  $F$  der Fresnel-Reflexions-Koeffizient ist, der als eine Funktion des einfallenden Lichts behandelt wird und gibt an, welcher Teil davon weiter reflektiert wird.

Wegen der Licht-Materie-Interaktion entsteht die Reflexionswelle, deren Leistung mit der vom Material absorbierten Energie gleich der einfallende Strahlungsenergie sein sollte. Zu einer Ebene, die von der Normalen  $N$  und der Richtung zum Licht  $L$  aufgespannt wird, kann die einfallende Lichtwelle eine parallele (Fresnel-Koeffizient,  $F_p$ ) und eine senkrechte (Fresnel-Koeffizient,  $F_s$ ) Polarisierung haben.

$F_p$  und  $F_s$  werden wie folgt definiert:

$$F_s = \frac{(c - G_1)^2 + G_2^2}{(c + G_1)^2 + G_2^2} \quad (20)$$

$$F_p = \frac{(uc - G_1)^2 + (2nkc - G_2)^2}{(uc + G_1)^2 + (2nkc - G_2)^2}$$

$$G_1^2 = \frac{1}{2}((u - s) + \sqrt{(u - s)^2 + 4n^2k^2})$$

$$G_2^2 = \frac{1}{2}(-(u - s) + \sqrt{(u - s)^2 + 4n^2k^2})$$

$$u = n^2 - k^2 \quad (21)$$

$$c = \cos \theta_i'$$

$$s = \sin^2 \theta_i'$$

$$\theta_i' = \cos^{-1}(L \cdot H)$$

wobei  $k$  den Absorptionsindex und  $n$  den Brechungsindex bezeichnen.

Mit Kenntnis der beiden Fresnel-Koeffizienten läßt sich der verwendete Fresnel-Koeffizient  $F$  durch eine lineare Kombination ausdrücken:

$$F = s \cdot F_p + t \cdot F_s \quad (22)$$

wobei  $s$  und  $t$  die Anteile bezeichnen, bei denen die Lichtwelle parallel, bzw. senkrecht zu der von  $L$  und  $N$  aufgespannten Ebene einfällt.

In der BRDF ist  $D$  eine Verteilungsfunktion der *Microfacetten*. Sie gibt die Verteilungen der *Microfacetten* in eine bestimmte Richtung aus.

Für einen bestimmten Halbvektor  $H$  gibt die Verteilungsfunktion die Verteilung der *Microfacetten* aus, deren Normalen in die selbe Richtung wie der Halbvektor  $H$  orientiert sind. Im Cook/Torrance-Modell [Cook81] gibt es viele verschiedene Verteilungsfunktionen, die bei der Berechnung gute Dienste leisten. In dieser Arbeit betrachten wir nur die oft verwendete Backmann-Verteilungsfunktion, die von Cook und Torrance empfohlen wurde und z. B. bei einer rauhen Oberfläche gut funktioniert.

$$D_{\text{Backmann}}(V, L) = \frac{1}{4\pi m^2 (N \cdot H)} e^{\frac{(N \cdot H)^2 - 1}{m^2 (N \cdot H)^2}} \quad (23)$$

wobei  $m$  die Wurzel der gemittelten Quadrate der Neigung der Flächen bezeichnet. Für eine rauhe Materialoberfläche wird für  $m$  ein großer Wert eingesetzt, so dass die Verteilung der Orientierung der *Microfacetten* sehr breit ist. Im Gegensatz dazu wird für  $m$  bei einer glatten Materialoberfläche ein kleiner Wert eingesetzt, um die Verteilung der Orientierung der *Microfacetten* zu verkleinern, so dass eine stärkere spiegelnde Reflexion entsteht.

Um spezielle Oberflächen beschreiben zu können, die mehr als eine Art der Rauheit, also mehrere Neigungen  $m$  besitzen, kann man die Verteilung auch als gewichtete Summe mehrerer Verteilungsfunktionen ausdrücken.

Die Summe der Gewichtungen  $w_m$  muß hierbei gleich 1 sein.

$$D(V, L) = \sum_{m=1}^n w_m D_m(V, L) \quad (24)$$

Nachdem wir die zwei Faktoren für Reflexion und Rauheit betrachtet haben, kommen wir zum dritten Faktor  $G$ , den sogenannten geometrischen Abschwächungsfaktor.



Abbildung 3.8: Links: die Selbstbeschattung  
Rechts: die Ablendung [Nayar89]

Es gibt Licht, das mit bestimmter Wahrscheinlichkeit von angrenzenden Nachbarn überschattet wird, bevor es in eine *Microfacette* eintrifft oder nachdem es von einer *Microfacette* reflektiert wurde.

Wenn wir mit Dreiecks-Kennntnis für beide Fälle berechnen, wieviel Strahlung durch

Selbstbeschattung und Ablendung abgeschattet wird, werden die zwei Abwächungsfaktoren wie folgt ausgedrückt:

$$G_{\text{Selbstbeschattung}} = \frac{2(N \cdot H)(N \cdot L)}{(V \cdot H)} \quad (25)$$

$$G_{\text{Ablendung}} = \frac{2(N \cdot H)(N \cdot V)}{(V \cdot H)} \quad (26)$$

Dabei müssen wir aber den dritten Fall beachten, bei dem das Licht möglicherweise überhaupt nicht überschattet werden kann. In diesem Fall wird die Strahlung ohne Leistungsverlust im Auge des Betrachters eintreffen und die Abwächungsfunktion gibt 1 aus.

Die Zusammenfassung der drei Fälle wird wie folgt definiert:

$$G(V, L) = \min \left\{ 1, G_{\text{Selbstbeschattung}}, G_{\text{Ablendung}} \right\} \quad (27)$$

setzen wir den Fresnel-Koeffizienten  $F$ , die Verteilungsfunktion  $D$  der *Microfacetten* und den geometrischen Abschwächungsfaktor  $G$  zusammen, dann erhalten wir eine BRDF des Cook/Torrance-Modells für die spiegelnde Reflexion. Um die gesamte BRDF zu beschreiben, müssen wir noch mit (12) die diffuse Komponente einsetzen. Das Cook/Torrance-Modell eignet sich in der Computergrafik gut zum Beschreiben der Oberfläche von Metallen, aber auch für andere Materialien mit verschiedenen Rauheiten, wenn wir unterschiedliche Parameter eingeben. In Abbildung 3.9 zeigen wir 12 gerenderte Vasen mit verschiedenen Parametern und Farben.



Abbildung 3.9: Cook's Vasen [Cook81]

### 3.4.7 Andere BRDF-Modelle

Auf der Basis von Blinn/Phone- und Cook/Torrance-Modellen wurden in der Vergangenheit viele weitere BRDF-Modelle entwickelt.

Das He-Modell [He91] [Yee02] berücksichtigt auch die von den vorhergehenden Modellen meist ignorierten Effekte, wie z. B. Lichtstreuung unterhalb der Oberfläche und anisotrope Oberflächen.

Um physikalisch korrekt, einfach und schnell in Computergrafik berechenbar zu sein, wurde 1992 das Ward-Modell von Gregory Ward vorgestellt [Ward92][Wynn00]. Dieses Modell ist physikalisch korrekt, enthält die Helmholtz-Reziprozität und genügt im Gegensatz zum Phong-Modell dem Energieerhaltungssatz.

Christophe Schlick stellte 1994 das Schlick-Modell vor [Schlick94], das ähnlich wie das Ward-Modell einfach und effizient in Hardwareimplementierung einsetzbar ist. Die BRDF dieses Modells setzt sich ebenso wie das Blinn/Phone-Modell aus diffusen und spiegelnden Komponenten zusammen. Dabei wird der geometrische Abwächungsfaktor zum Beschreiben der Selbstbeschattung und Abblendung der *Microfacetten* vernachlässigt. Lafortune generalisiert das Cosine-Lobe-Modell und stellte 1993 das Lafortune-Modell vor [Lafortune97]. Sonst wurde das Ashikhmin-Modell im Jahr 2000 vorgestellt [Ashikhmin00].

Wenn man sich weiter an den BRDF-Modellen interessiert, kann man auch [Addy04] lesen, in der eine Evaluation für einige häufig verwendete Modelle aufgestellt wird.



Abbildung 3.10: Kugel mit Material Chrome  
(von links an) Original, Ashikhmin-, Blinn/Phong-,  
Cook/Torrance-, Lafortune-, Ward-Modell [Addy04]

## 3.5 Measured-BRDFs

### 3.5.1 Definition von measured-BRDFs

Obwohl die BRDF-Modelle, die wir zuvor vorgestellt haben, mit höher Geschwindigkeit in Hardwareimplementierung zur Computergrafik gut funktionieren, sind sie nicht immer greifbar und geeignet für praktische BRDFs. Viele komplizierte Materialoberflächen, die anisotrop sind und deren Reflexionseigenschaften bzgl. eines Drehwinkels nicht mehr konstant bleiben, können mit diesen Modellen nicht genau simuliert werden. Eine weitere Variante zum Aufbau von solchen BRDFs basiert auf der Technik von *Image-based-Rendering*. Um zweckmäßig, effektiv und präzise die Oberfläche eines Materials zu beschreiben und praxisnah zu simulieren, ist es eine der

besten Arten, die physikalischen Reflexionsdaten durch Messgeräte direkt zu akquirieren. Allgemein im Gegensatz zur analytischen Beschreibung werden in diesem Fall die BRDFs so repräsentiert, durch punktuelle Samples für die Reflexionseigenschaften, zwischen denen dann interpoliert werden muss.

### 3.5.2 Akquisition von BRDFs

#### 3.5.2.1 Das Messgerät und die Messanlagen

Um die BRDF einer opaken Oberfläche und die Reflexionsverteilung zu ermitteln, muß die einfallende und reflektierte Strahlung gemessen werden.

Als Messgerät dient oft das hochlösende Spektrometer OVID (Optical Visible and Infrared Detector). Das Gerät ist teuer, selten und funktioniert sehr gut. Die Pixel des CCD-Chips im Gerät sind in einem zweidimensionalen Feld (Matrix) angeordnet. Bei der Akquisition kann auch die Lichtwelle durch die Messung von RGB-Farben ersetzt. Sobald das Licht im Gerät eintrifft, wird es durch einen gekrümmten Spiegel fokussiert und fällt anschließend auf eine CCD-Kamera im Detektor. Die Kamera nimmt die Intensität des Lichts auf und speichert dann den entsprechenden in RGB umgewandelten Wert im Speicher ab.

In der Praxis gibt es drei Messmethoden:

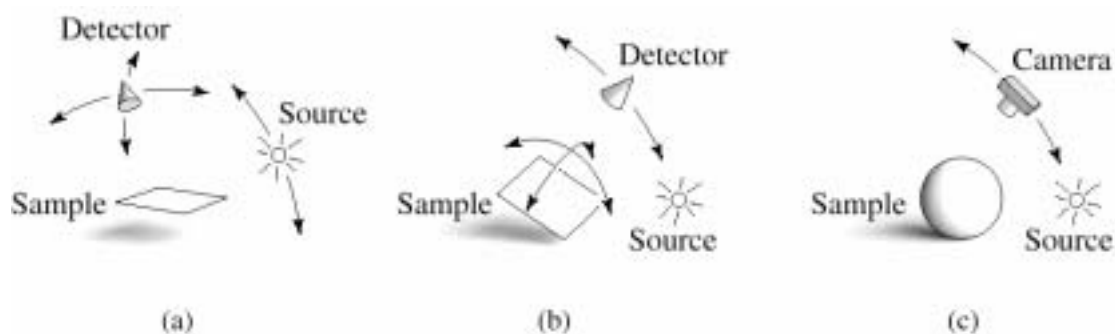


Abbildung 3.11: Drei verschiedene Anlagen der Messung [Marschner99]

Sollte eine anisotrope Materialoberfläche gemessen werden, muß eine Messanlage mit einem dreidimensionalen Messtisch aufgebaut werden, so dass sich der Detector flexibel auf jede Position in einem bestimmten Deklinations- und Azimutwinkel bewegen kann. (Siehe Abbildung 3.11(a)). Wenn die Daten in jede mögliche Richtung gesammelt werden müssen, ist diese Methode sehr aufwendig, weil es zu viele überlappende Punkte gibt. Tatsächlich sind für uns lediglich die Positionen des Samples, der Lichtquelle und der Kamera wichtig. Deshalb ist die zweite Anlage verwendbar (Siehe Abbildung 3.11(b)), in der die Kamera eine feste eindimensionale Bewegung und das Sample eine zweidimensionale Bewegung hat, für die zu messenden Punkte bleibt das dreidimensionale System unverändert. Eine oft verwendete Methode ist die dritte Anlage, die die Intensität akquiriert. Kamera und Lichtquelle bewegen sich in Deklinations- und Azimutrichtung und die Messung wird



gleichmäßig auf einer halben Hemisphäre durchgeführt. Das Intervall zwischen 0 und  $\pi$  in Azimutrichtung entspricht einer Spiegelung zum Intervall zwischen  $\pi$  und  $2\pi$ .

$$\theta_i \in \left(0, \frac{\pi}{2}\right), \theta_r \in \left(0, \frac{\pi}{2}\right), \phi_r \in (0, \pi), \phi_i \in (0, \pi), \phi_r \in (0, \pi)$$

Wenn die Intensität der Lichtquelle und die vom der Kamera gemessene Intensität in verschiedenen Kombinationen vom Deklinations- und Azimutwinkel abgespeichert werden, sind wir in der Lage, die Reflexionseigenschaft des Materials und die BRDF der Fläche zu ermitteln.

### 3.5.2.2 Quellen der BRDF-Datenbanken

Zur Zeit haben wir zwei sehr bekannte Datenquellen der gemessenen BRDFs. Die Erste ist die an der Universität-Cornell mittels Gonioreflectometer akquirierte Datenbank [Cornell06]. Darauf wird die Menge der Datensätze für die zwei Materialien *mystique* (*Mystic lacquer*) und *cayman* (*Dupont Cayman lacquer*) angeboten. (Siehe Abbildung 3.12)

Die Messpunkte sind regelmäßig über eine halbe Hemisphäre verteilt. Es wird in der Deklinationsrichtung innerhalb  $\pi/2$  in  $10^\circ$ -Schritten gemessen, in den einzelnen Deklinationswinkeln werden Punkte mit verschiedenen Azimutwinkeln ausgewählt.



Abbildung 3.12: Material cayman und mystique [Cornell06]

Da das einfallende Licht in niedrigen Deklinationswinkeln kaum Auswirkung auf die Reflexion der Oberfläche hat, wird bei kleinen Deklinationswinkeln geringe Intensität gemessen, bei dem Punkt mit einem Einfallswinkel gleich Null wird ein einziger Punkt in Reflexionsrichtung gemessen. Im Gegensatz werden mehrere Punkte ausgemessen, falls die Deklinationswinkel in dieser Richtung etwas größer sind (Siehe Abbildung 3.13). Insgesamt werden 1439 Datensätze mit Kombinationen von Einfallswinkel-, Einfallswinkel-, Ausfallswinkel- und Ausfallswinkel abgespeichert.

Da die Materialien beide isotrop sind, wird auf die Eingabe des Einfallswinkels verzichtet und stattdessen Null für jeden Datensatz eingesetzt. D.h. bei der Berechnung brauchen wir uns nur um den Einfallswinkel- und Ausfallswinkel,

sowie die Differenz zwischen dem Einfalls- und Ausfallsazimutwinkel zu kümmern. Sonst entsteht bei dieser Datenquelle die Problematik, dass negative Werte wegen Messfehler in manchen Datensätzen existieren, obwohl die Werte für die Lichtwelle im RGB- oder XYZ-Farbraum definiert werden. Das bringt selbstverständlich Umstände beim Rendern mit sich. In unserem Programm ersetzen wir die negativen Werte durch Null. Die andere ist die *CURet*-Datenbank [CURet06], die an der Universität- Columbia-Utrecht von K. J. Dana aufgebaut wird und die frei herunterladbare Messdaten für BRDF und BTF (*Bidirectional Texture Functions*) enthält.

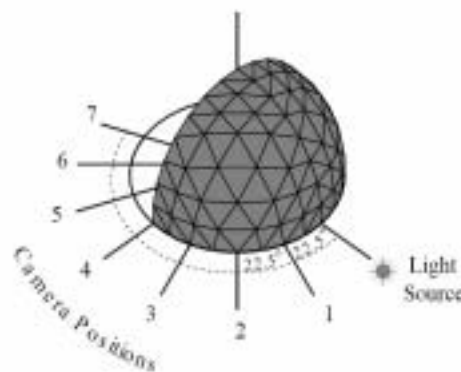


Abbildung 3.13: Der Messprozess von K. J. Dana [Dana99]

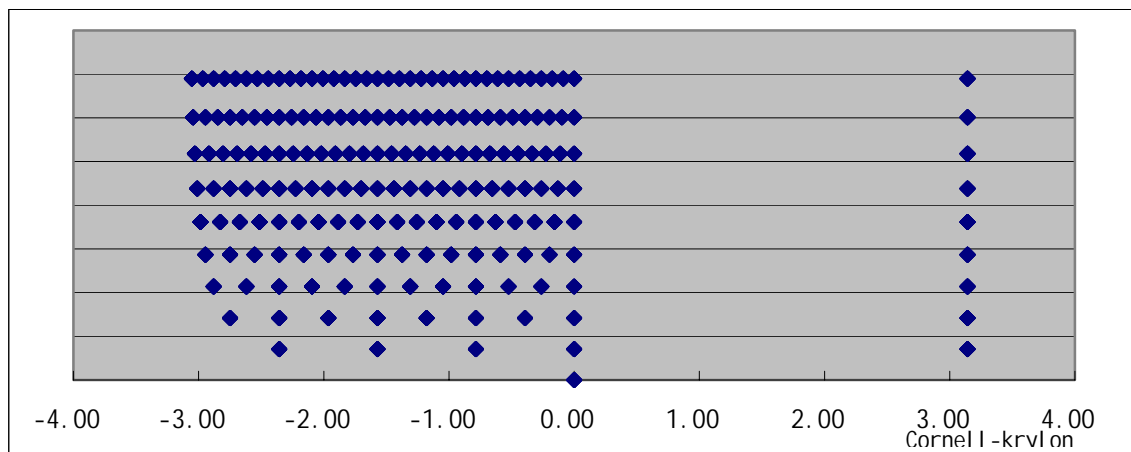


Abbildung 3.14: Die Verteilung der Messpunkte der Cornell-Datenbank

Die Datenbank für die BTF werden wir später betrachten. In der BRDF-Datenbank sind die Messdaten der Oberflächen für 61 verschiedene Materialien enthalten, welche in der realen Welt gesammelt werden, um einen möglich großen Bereich des Materials mit den speziellen geometrischen und photometrischen Eigenschaften zu umfassen. Die Proben beinhalten isotrope und anisotrope Materialoberflächen, die auch manche scharfe spiegelnde oder diffuse Reflexionen aufweisen.

Für jedes der 61 Materialien werden Bilder zur Berechnung der reflektierten Strahlung unter 205 unterschiedlichen Kombinationen von Lichtquelle- und

Kameraposition aufgenommen und die Datensätze weiter in einer Tabelle abgespeichert. Die Tabelle wird in drei getrennten Dateien für die R, G, B-Farbwerte ins Internet gestellt. Aus Sicht der Speicherersparnis ist eine solche Tabelle mit 205 Datensätzen einfach und schnell im Rendering-Programm implementierbar, es ist sogar unproblematisch, eine Szene mit vielen Materialien in Echtzeit zu rendern. Aber wenn sich die Reflexionseigenschaft einer Materialoberfläche zwischen den Messpunkten heftig verändert, ist die geringe Anzahl von 205 Datensätzen kaum in der Lage, die Oberfläche komplett und präzise zu beschreiben. Sollte die Verteilung der reflektierten Strahlung von einer Oberfläche relativ gleichmäßig sein, kann sie gut in Computer-Grafik simuliert werden, da die möglichen betrachteten Richtungen, die in den Messdaten nicht vorhanden sind, durch Interpolation der nächsten Richtungen ausgerechnet werden können. Die Kugeln, die durch BRDF für die 61 verschiedenen Materialien gerendert wurden, werden unten gezeigt:

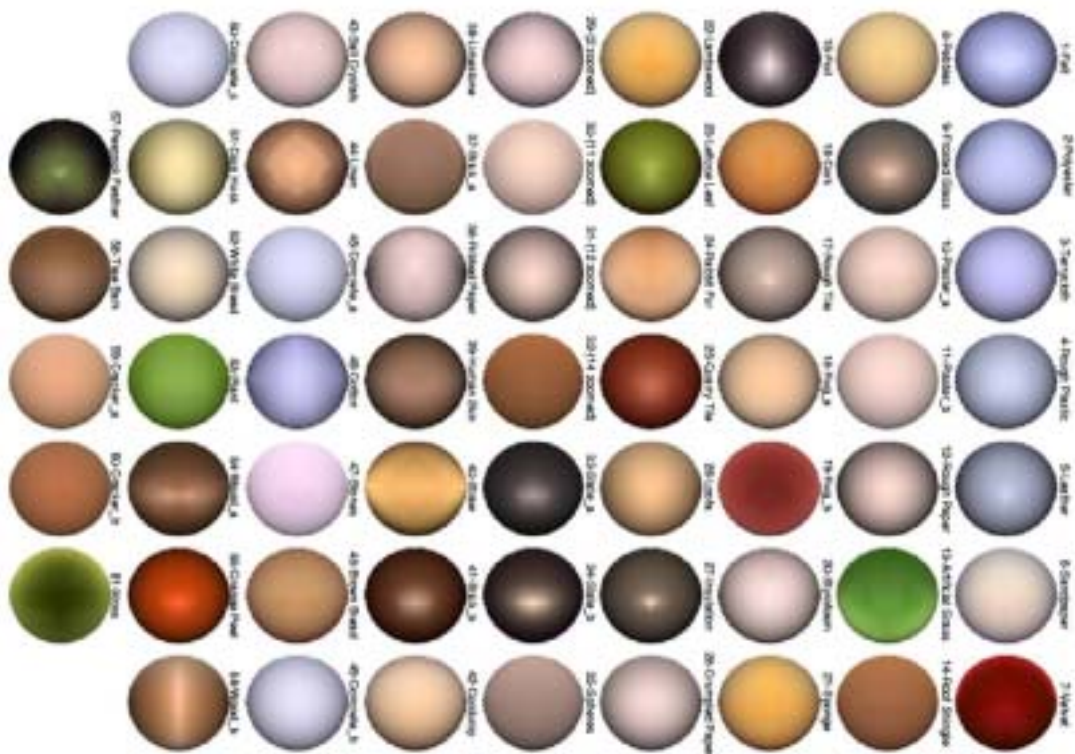


Abbildung 3.15: Die gerenderten Kugeln aus 61 Materialien [Curet06]

### 3.5.2.3 Interpolation zwischen den Messpunkten

In der Mathematik gibt es die Möglichkeit der Interpolation, die hier bei Datenmangel oder zur Speicherersparnis eingesetzt wird. Sind die Werte der Nachbarn eines zu berechnenden Punktes bekannt, so kann der Wert für diesen durch Interpolation bestimmt werden. Hierbei kann es jedoch vorkommen, dass der interpolierte Wert in einigen Fällen nicht ganz korrekt ist.

Falls ein Punkt innerhalb eines Dreiecks liegt und die Werte für die drei Ecken bereits

bekannt sind, können wir zuerst eine Linie durch den Punkt bestimmen, die parallel zu einer der Verbindungslinien der Dreieckspunkte verläuft. Beide Enden dieser Linie können ohne Grenze in die zwei Richtungen verlängert werden, so dass wir die zwei Schnittpunkte mit den anderen beiden Verbindungslinien berechnen können. Aus den beiden unterschiedlichen Neigungen der Schnittpunkte erhalten wir die Gewichtungen  $s_1$  und  $s_2$ , die wir zur Interpolation zwischen den zwei Schnittpunkten verwenden. (Siehe Abbildung 3.16).

Bezeichnen wir die Werte der drei Ecken des Dreiecks mit  $W(p_1)$ ,  $W(p_2)$ ,  $W(p_3)$  und definieren die Funktion  $L(x, y)$  als die Distanz zwischen den Punkten  $x$  und  $y$ , so ist der betrachte Punkt  $q$ :

$$\begin{aligned}
 s_1 &= \frac{L(p_1, p_2)}{L(q_1, p_2)} \\
 s_2 &= \frac{L(p_1, p_3)}{L(q_2, p_3)} \\
 W(q_1) &= s_1 \cdot W(p_1) + (1 - s_1) \cdot W(p_2) \\
 W(q_2) &= s_2 \cdot W(p_1) + (1 - s_2) \cdot W(p_3)
 \end{aligned} \tag{28}$$

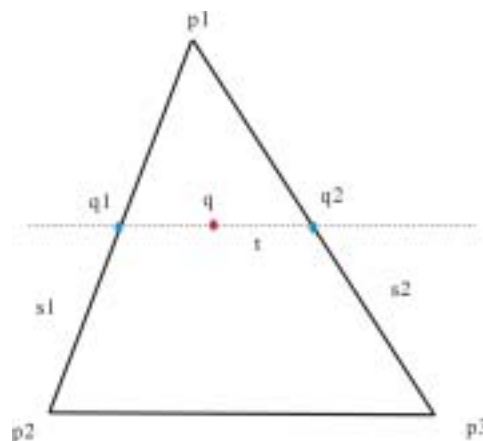


Abbildung 3.16: Interpolation in Dreieck

Auf dieser Weise ist der Wert des betrachteten Punktes auch aus den Werten der zwei Schnittpunkte berechenbar.

$$\begin{aligned}
 t &= \frac{L(q_1, q_2)}{L(q, p_2)} \\
 W(q) &= t \cdot W(q_1) + (1 - t) \cdot W(q_2)
 \end{aligned} \tag{29}$$

Im Prinzip funktioniert die Interpolation für ein Viereck genauso. Hier wählen wir zuerst eine Seitenlinie und dann eine Parallele hierzu, die den betrachteten Punkt enthält. Nachdem wir die Schnittpunkte mit den anderen zwei Seitenlinien gefunden haben, haben wir auch die entsprechenden Gewichtungen, durch die der Wert für den betrachteten Punkt berechnet werden kann.

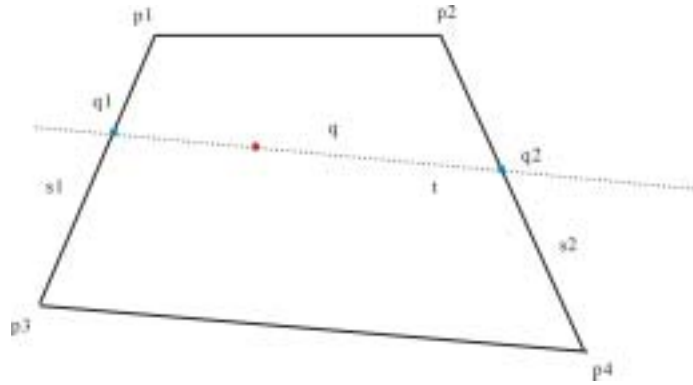


Abbildung 3.17: Interpolation in einem Viereck

Wir bezeichnen die Werte der vier Ecken des Vierecks mit  $W(p_1)$ ,  $W(p_2)$ ,  $W(p_3)$ ,  $W(p_4)$ , und den betrachteten Punkt mit  $p$ .

$$s_1 = \frac{L(p_1, p_3)}{L(q_1, p_3)}$$

$$s_2 = \frac{L(p_2, p_4)}{L(q_2, p_4)}$$

$$W(q_1) = s_1 \cdot W(p_1) + (1 - s_1) \cdot W(p_3) \quad (30)$$

$$W(q_2) = s_2 \cdot W(p_2) + (1 - s_2) \cdot W(p_4)$$

$$t = \frac{L(q_1, q_2)}{L(q, p_2)}$$

$$W(q) = t \cdot W(q_1) + (1 - t) \cdot W(q_2)$$

Im Vergleich zu den beiden Methoden wird die Berechnung von BRDFs komplizierter. Gehen wir davon aus, dass wir die Viereck-Interpolation anwenden, wird dabei eine viermalige Bilineare-Interpolation durchgeführt, die parallel auf beide Richtungen durchgeführt wird. Da wir die Lichtquelle- und Kamerarichtungen bei jedem Punkt der Messung von BRDFs festlegen müssen, haben wir vier Werte der Deklinations- und Azimutwinkel für die Einfallrichtung, sowie vier Werte für die Ausfallrichtung zu interpolieren. Zum einen suchen wir zuerst für den Einfallswinkel die vier nächsten Messpunkte, die sowohl zwei Grenzen in Deklinationsrichtung, als auch zwei Grenzen in Azimutsrichtung haben, zum anderen suchen wir auf dieselbe Weise die vier Messpunkte für den Ausfallswinkel. Für jede Kombination, die von einem der vier Messpunkte des Einfallswinkels und den vier Grenzen des Ausfallswinkels gebildet wird, führen wir, wie oben beschrieben, eine Interpolation im Viereck durch, um die vier interpolierten Farbwerte der entsprechenden Messpunkte zu berechnen. Danach kann der Farbwert des Einfallswinkels durch die gerade berechneten Farbwerte interpoliert werden. (Siehe Abbildung 3.18).

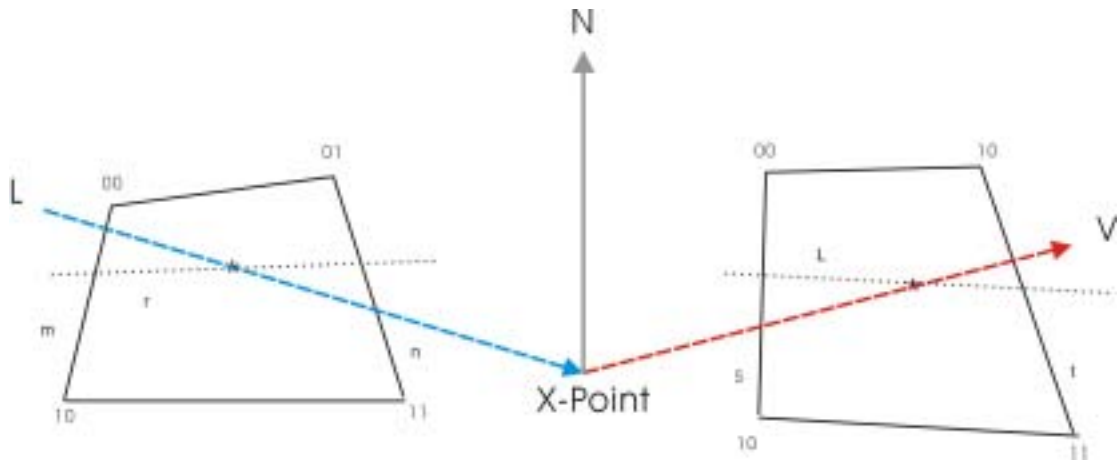


Abbildung 3.18: Interpolation für zwei Richtungen

Bezeichnen wir den Farbwert der Ausgabe von BRDFs mit  $f(X, \theta_i, \phi_i, \theta_r, \phi_r)$  und für eine bestimmte Lichtrichtung  $W(\theta_i, \phi_i)$  und Reflexionsrichtung  $W(\theta_r, \phi_r)$ .

$$\begin{aligned}
 \theta_0 &\leq \theta_i \leq \theta_1, \phi_0 \leq \phi_i \leq \phi_1 \\
 \theta'_0 &\leq \theta_r \leq \theta'_1, \phi'_0 \leq \phi_r \leq \phi'_1 \\
 0 &\leq t, s, l, m, n, r \leq 1
 \end{aligned}
 \tag{31}$$

$$\begin{aligned}
 q_{i,j} &= t \cdot f(\theta_i, \phi_j, \theta'_0, \phi'_0) + (1-t) \cdot f(\theta_i, \phi_j, \theta'_1, \phi'_1) \\
 p_{i,j} &= s \cdot f(\theta_i, \phi_j, \theta'_0, \phi'_0) + (1-s) \cdot f(\theta_i, \phi_j, \theta'_1, \phi'_1) \\
 f_{i,j} &= l \cdot q_{i,j} + (1-l) \cdot p_{i,j} \quad \text{for } i, j = 0, 1
 \end{aligned}$$

wobei

$\theta_0$  und  $\theta_1$  : Unten- und Obergrenze des Einfallswinkels,

$\phi_0$  und  $\phi_1$  : Unten- und Obergrenze des Einfallswinkels,

$\theta'_0$  und  $\theta'_1$  : Unten- und Obergrenze des Ausfallswinkels,

$\phi'_0$  und  $\phi'_1$  : Unten- und Obergrenze des Ausfallswinkels,

$t, s, l, m, n, r$  : Gewichtungen

sind.

Sobald die vier interpolierten Werte für einen bestimmten Einfallswinkel bekannt sind, können wir den Farbwert des betrachteten Punktes durch die folgende Gleichung berechnen.

$$\begin{aligned}
f_{right} &= m \cdot f_{0,0} + (1-m) \cdot f_{1,1} \\
f_{left} &= n \cdot f_{0,1} + (1-n) \cdot f_{1,0} \\
f_{inpl} &= r \cdot f_{right} + (1-r) \cdot f_{left}
\end{aligned} \tag{32}$$

Wobei die Gewichtungsfaktoren  $m, n, r$  zwischen 0 und 1 liegen.

### 3.5.3 Lineare Faktorisierungen von BRDFs

Wie wir oben bereits erwähnt haben, kann die Reflexionseigenschaft von Oberflächen umso genauer und konkreter beschrieben werden, je mehr Datensätze einer gemessenen BRDF angeboten werden. Sind jedoch in 3D-Szenen mehrere Materialien von verschiedenen Objekten vorhanden und jede BRDF enthält mehr als tausend Datensätze, so werden für ein Rendern in Echtzeit die Kosten wegen der Speicherbeschränkung des Rechners ziemlich hoch. Daher benötigen wir zur Reduktion des Speicherbedarfs einige effektive Verfahren.

Eine BRDF besitzt zwei Richtungsvektoren. Der eine wird von der Lichtquelle und dem betrachteten Punkt definiert, der andere vom betrachteten Punkt und der Kameraposition. Dazu ist teilbare Zerlegung (*Separable Decomposition*) oft verwendet, um eine BRDF als eine Summe von Produkten der Zerlegungsfunktionen bezeichnen zu können.

$$f(\theta_i, \phi_i, \theta_r, \phi_r) \approx \sum_{k=1}^n p_k(\theta_i, \phi_i) \cdot q_k(\theta_r, \phi_r) \tag{33}$$

Die Vorgehensweise der Faktorisierung ist, die Einfalls- und Ausfallsrichtung auf die Pixel einer Textur abzubilden. Dadurch wird eine BRDF in mehrere wichtige Teile mit niedrigeren Dimensionen (2D oder 1D) zerlegt, so dass die Richtungsvektoren auf einer Textur interpoliert werden können. In der Textur der Einfallsrichtung besitzt jedes Pixel der  $UV$ -Koordinate eine einfallende Richtung mit bestimmtem Winkel, die Textur der Ausfallsrichtung ebenso.

Bei der Anwendung können die Texturen der Einfalls- und Ausfallsrichtungen für verschiedene Zerlegungsfunktionen in einer Vorarbeitungsphase (*offline*) produziert werden. Wenn ein Punkt von Oberflächen gerendert werden soll, müssen ihre Richtungsvektoren ausgerechnet und weiter in die  $UV$ -Koordinaten

$(u_i, v_i)$  und  $(u_r, v_r)$  der Texturen umgerechnet werden. Dadurch können die Pixelwerte

der Texturen direkt gefunden oder interpoliert werden. Durch Multiplizieren dieser Pixelwerte berechnen wir weitere Pixelwerte für andere Zerlegungsfunktionen. Am Ende addieren wir alle auf diese Weise ausgerechneten Werte für den Farbwert des betrachteten Punktes.

### 3.5.3.1 Normalisierte Zerlegung (Normalized Decomposition, ND)

Das von Chris Wynn [Wynn] vorgestellte ND-Verfahren ist einfach und schnell zu implementieren. Es kann so entwickelt werden, dass der Speicherbedarf einer BRDF die Begrenzung von Speichern eines aktuellen Rechners nicht überschreitet.

Die Grundidee der Zerlegung besteht darin, dass wir eine BRDF als eine 4D-Funktion behandeln und diese anschließend in einen zweidimensionalen Raum projizieren. Um unser Ziel zu erreichen, können wir die ausführbaren mathematischen Operationen auf dem zweidimensionalen Raum einsetzen.

Da eine BRDF als eine Funktion mit vier Parametern  $f(\theta_i, \phi_i, \theta_r, \phi_r)$  bezeichnet werden kann, speichern wir die Werte einer Menge von ein paar, regelmäßig auf einer Hemisphäre verteilten Messpunkten für bestimmte Eingaben, bzw. die Deklinations- und Azimutwinkel ab und verwenden für die andere, dazwischenliegende Punkte die Interpolation zur Berechnung ihrer Intensitäts- oder Farbwerte. Dabei ist das Intervall von  $\theta$  im Bereich von  $[0, \pi/2]$  und  $\phi$  von  $[0, 2\pi]$ . Bei der Implementierung des Programms kann ein vierdimensionales Array  $BRDF[i][j][l][k]$  zur Aufnahme der BRDF genutzt werden.

Statt die Datensätze einer BRDF in einem 4D-Array abzuspeichern, können wir sie in einer Matrix abbilden. Eine Richtung wird als ein paar  $(\theta, \phi)$  wiedergegeben. In dieser Matrix-Repräsentation enthält jede Spalte die Messwerte für die Einfallrichtung  $L$  und alle gemessenen Reflexionsrichtungen  $V$ , und jede Zeile enthält die Farbwerte für eine feste Reflexionsrichtung und alle gemessenen Einfallrichtungen. Jede Position der Matrix entspricht einer spezifischen Kombination von Einfall- und Ausfallrichtung und jeder Wert der Matrix ist eine Ausgabe der BRDF in Abhängigkeit von den entsprechenden vier Parametern.

Nun haben wir eine Abbildung einer 4D-BRDF auf eine 2D-Matrix und sind in der Lage, die in einer 2D-Matrix dargestellte 4D-Funktion in zwei 2D-Funktionen zu zerlegen. Das Ziel ist zwei 2D-Funktionen zu suchen, so dass

$$f(\theta_i, \phi_i, \theta_r, \phi_r) \cong G(\theta_i, \phi_i) \cdot H(\theta_r, \phi_r) \quad (34)$$

Auf jeder Zeile der BRDF-Matrix berechnen wir einen Normwert, der ein Typ des durchschnittlichen Werts eines Vektors ist und hier als Zweite-Norm bezeichnet wird

$$Norm(\vec{V}) = (|v_1|^2 + |v_2|^2 + \dots + |v_n|^2)^{\frac{1}{2}} \quad (35)$$

Bezeichnen wir die Matrix mit  $(m_{i,j})_{i \in \{V\}, j \in \{L\}}$  und nehmen an, es gibt  $n$  Punkte in Einfallrichtung und  $l$  Punkte in Reflexionsrichtung, so ist der Vektor  $vec$



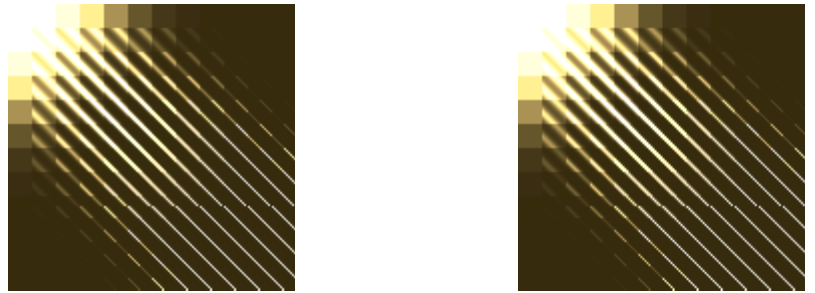
$$\begin{bmatrix} \text{norm}(m_{1,1}, m_{1,2}, \dots, m_{1,n-1}, m_{1,n}) \\ \vdots \\ \text{norm}(m_{l,1}, m_{l,2}, \dots, m_{l,n-1}, m_{l,n}) \end{bmatrix} \quad l \times 1 - \text{Vektor } \vec{H} \quad (36)$$

Auf jeder Spalte der BRDF-Matrix berechnen wir den Durchschnitt der normalisierten Werte.

$$\left[ \frac{1}{l} \sum_{k=1}^l \frac{m_{k,1}}{nvec_k} + \frac{1}{l} \sum_{k=1}^l \frac{m_{k,2}}{nvec_k} + \frac{1}{l} \sum_{k=1}^l \frac{m_{k,3}}{nvec_k} + \dots + \frac{1}{l} \sum_{k=1}^l \frac{m_{k,n-1}}{nvec_k} + \frac{1}{l} \sum_{k=1}^l \frac{m_{k,n}}{nvec_k} \right] \quad (37)$$

$1 \times n - \text{Vektor } \vec{G}$

Die zwei Vektoren stehen für  $H(\theta_r, \phi_r)$  und  $G(\theta_i, \phi_i)$ , die zur Approximation der BRDF-Matrix dienen. Dazu zeigen wir ein Beispiel von [Wynn00]:



(a) Das originale Bild der Matrix      (b) Zwei Vektoren für  $G, H$   
Abbildung 3.19: ND-Zerlegung [Wynn00]

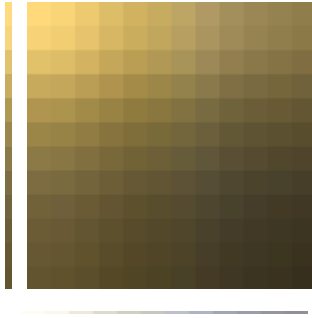
Beim Rendern brauchen wir nur die zwei Vektoren in den Speicher zu laden, um die ganze BRDF-Matrix wiederzugeben. Der Quotient des vorherigen und des neuen

Speichedarfs beträgt  $\frac{(l \times n)}{(l + n)}$ , von dem wir einen guten Gewinn haben.

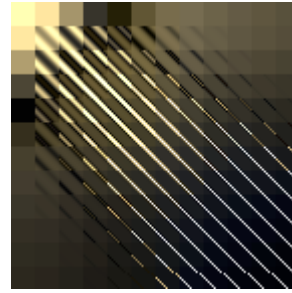
Die zwei Vektoren für  $H(\theta_r, \phi_r)$  und  $G(\theta_i, \phi_i)$  können ebenso bei der Anwendung in zwei getrennten Texturen abgespeichert werden, damit der Farbwert jedes Punktes auf der Textur schnell und einfach interpoliert werden kann

Um die BRDF-Matrix zu rekonstruieren, müssen wir die zwei Vektoren miteinander multiplizieren.

$$m'_{i,j} = \vec{G}_i \cdot \vec{H}_j \quad (38)$$



(a) Bild der rekonstruierten Matrix



(b) Differenz der zwei Matrizen, welche die Abweichung der Approximation zeigt.

Abbildung 3.20: Rekonstruktion der BRDF-Matrix [Wynn00]

### 3.5.3.2 Gram-Schmidt-Halbwinkel-Differenz (G. S. Halfangle-Difference, GSHD)

Um den Fehler der Approximation möglicherweise zu verringern, wird eine andere Variante in [Wynn] vorgestellt. Statt Lichtrichtung  $L$  und Kamerarichtung  $V$  sind in diesem Verfahren ein Halbvektor  $H$  zwischen  $L$  und  $V$  und ein Differenz-Vektor  $D$  als Parameter von BRDFs einzugeben.

$$H = \frac{L+V}{|L+V|} \quad (39)$$

Mit dem *Gram-Schmidt orthogonalization process* [Giraud04] können wir die anderen zwei Vektoren  $t$  und  $s$  bestimmen, die zusammen mit  $H$  ein lokales Koordinatensystem bilden. Die drei Vektoren  $\{H, t, s\}$  sind normiert,  $H$  ist nach oben orientiert und  $t, s$  stehen orthogonal zueinander.

$$\begin{aligned} s &= H \times t, \\ t &= H \times s, \\ H &= t \times s \end{aligned} \quad (40)$$

Für den Differenz-Vektor  $D$  ist die intuitive Bedeutung schwer zu erklären. Man kann es sich so vorstellen, dass er die Abweichung des  $L$ -Vektors zum lokalen Koordinatensystem darstellt. Er wird wie folgt definiert:

$$D = \begin{pmatrix} L \cdot t \\ L \cdot s \\ L \cdot H \end{pmatrix} \quad (41)$$

Weil die Messpunkte tatsächlich in  $L$ - und  $V$ -Richtung gemessen werden, brauchen wir eine Berechnung der BRDF-Werte für die gegebenen  $D(x, y, z)$  und  $H$ , damit die Abbildung einer 4D-BRDF zu einer 2D-Matrix mit GSHD korrekt funktioniert.

$$\begin{aligned} L &= (D.x) \cdot t + (D.y) \cdot s + (D.z) \cdot H \\ V &= 2(H \cdot L)H - L \end{aligned} \quad (42)$$

Nun ist unsere Aufgabe,  $H'(\theta_d, \phi_d)$  und  $G'(\theta_h, \phi_h)$  zu suchen, so dass

$$f'(\theta_h, \phi_h, \theta_d, \phi_d) \cong G'(\theta_h, \phi_h) \cdot H'(\theta_d, \phi_d) \quad (43)$$

Um das Problem zu lösen, können wir dieselbe Vorgehensweise nutzen, wie oben. Im Vergleich zur ND-Zerlegung hat GSHD bei der Zerlegung der BRDF-Matrix zu zwei Vektoren einen geringeren Informationsverlust und die Differenz der BRDF-Matrix, die den Fehler der Approximation zeigt, ist auch etwas geringer. In Abbildung 3.21 wird der Zerlegungsprozess mit GSHD dargestellt.



(a) Das originale Bild der BRDF-Matrix.



(b) Faktorisierung mit GSHD.



(c) Rekonstruktion der BRDF-Matrix.



(d) Differenz zwischen (a) und (c)

Abbildung 3.21: GSHD-Zerlegung [Wynn00]

### 3.5.3.3 Singular-Wert-Zerlegung (Singular Value Decomposition, SVD)

Sollte eine Matrix  $M = [m_{i,j}]_{i=1..m, j=1..n}$  gegeben sein, so kann  $M$  durch *Singular Value Decomposition* (SVD) [Kautz99] [Andrews77] als eine Faktorisierung  $M = UDV^T$  dargestellt werden, wobei die Spalten von  $U = [u_k]$  und  $V = [v_k]$  orthonormal und die Spalten von  $D = \text{diag}(d_k)$  Eigenvektoren sind. Die Vektoren, die durch die Spalten von  $U$  und  $V$  gebildet werden, sind zueinander senkrecht und in  $D$  sind nur die Werte auf der Diagonalen ungleich Null.

Auf mathematische Weise kann das Produkt  $UDV^T$  durch eine Summe bezeichnet werden.

$$M = \sum_{i=1}^K s_i u_i v_i^T \quad (44)$$

Sowie wir vorher erwähnt haben, speichern wir die Datensätze einer BRDF in eine Matrix  $M$  mit  $[m_{i,j}] = f(L_i, V_j)$  ab, deren Spalten für die Einfallrichtung stehen und deren Zeilen der Ausfallrichtung entsprechen.

$$M = \begin{pmatrix} f(L_1, V_1) & f(L_1, V_2) & \dots & f(L_1, V_{k-1}) & f(L_1, V_k) \\ \vdots & & & \ddots & \vdots \\ f(L_k, V_1) & f(L_k, V_2) & \dots & f(L_k, V_{k-1}) & f(L_k, V_k) \end{pmatrix} \quad (45)$$

Wenn wir  $u_k$  und  $v_k$  durch SVD-Faktorisierung von  $M$  in zwei 2D-Funktionen  $u_k(L)$  und  $v_k(V)$  transformieren und nehmen nur die Komponenten an, für die gilt, dass  $n < k$ , so erhalten wir die Approximation:

$$f(L, V) \approx \sum_{k=1}^n s_k \cdot u_k(L) \cdot v_k(V) \quad (46)$$

Im Vergleich zur ND-Zerlegung ist die SVD-Faktorisierung etwas komplizierter und präziser, aber ein Nachteil ist, dass die SVD negative Werte enthalten kann, die nicht zu den positiven Reflexionswerten passen und auch Probleme in der Hardwareimplementierung bringen können. Um dieses Problem zu lösen ist [Lee99] eine gute Alternative. Zwar nimmt der Speicherbedarf von SVD mit dem Aufstieg der Auflösung schnell zu. Wenn wir 64 Messpunkte in jede Eingabe einer BRDF abtasten, wird für eine solche Matrix  $64 \times 64 \times 64 \times 64$  Bytes gebraucht. Belegt eine Float-Zahl 4 Bytes, beträgt der Speicherbedarf 64 MB. Es wird 1 GB dabei eingesetzt, wenn wir die Auflösung auf  $128 \times 128 \times 128 \times 128$  erhöhen.

### 3.5.3.4 Homomorphe Faktorisierung (Homomorphic Factorization, HF)

Im Jahr 2001 hat McCool eine neue Faktorisierungsmethode [McCool01] mitgebracht. In diesem Papier kann die Homomorphic Factorization (HF) jede BRDF in ein Produkt aus zwei oder mehreren Faktoren mit niedrigeren Dimensionen zerlegen. Jeder hängt von einem Parameter des Richtungsvektors ab. Im Vergleich zur Faktorisierung, die auf der SVD basiert, ermöglicht die HF-Technik eine bessere Faktorisierung mit positiven Werten und weniger Fehlern.

Im letzten Abschnitt haben wir schon gesehen, dass eine BRDF als eine Summe von Produkten aus Funktionen mit niedrigeren Dimensionen annähernd berechnet werden kann. Wenn wir uns vorstellen, dass es in dieser Summe nur eine einzige Komponente gibt, können wir stattdessen eine BRDF als ein einziges Produkt von verschiedenen Faktoren bezeichnen:

$$f(L, V) \approx \prod_{j=1}^J p_j(\pi_j(L, V)) \quad (47)$$

wobei  $p_j$  für unterschiedliche 2D-Funktionen und  $\pi_j: \mathbb{R}^4 \rightarrow \mathbb{R}^2$  für die Projektionsfunktionen steht, die für die 2D-Funktionen die Parameter definieren, die aus den Parametern  $L(\theta_i, \phi_i)$  und  $V(\theta_r, \phi_r)$  einer BRDFs transformiert werden.

Wenn wir die beiden Seiten der Gleichung logarithmieren, erhalten wir die folgende Gleichung:

$$\log(f(L, V)) \approx \sum_{j=1}^J \log(p_j(\pi_j(L, V))) \quad (48)$$

$$f'(L, V) \approx \sum_{j=1}^J p'_j(\pi_j(L, V)), \quad \log(f) = f' \quad (49)$$

Auf diese Gleichung mit der Summe ist entweder eine SVD- oder eine ND-Zerlegung durchführbar und gleichzeitig garantiert der Logarithmus auf beiden Seiten, dass die ausgerechnete Werte nur im positiven Wertebereich liegen kann. Aber die Werte müssen vor der Ausgabe noch exponentiell umgewandelt werden.

Der Logarithmus hat jedoch auch den Nachteil, dass wenn die Ausgabe von  $f(L, V)$

gleich Null sein kann, ein kleiner Anfangswert  $\varepsilon$  in  $f(L, V)$  eingefügt werden muß,

bevor die Gleichung logarithmiert wird. Dieser Anfangswert existiert eigentlich nicht und bringt daher eine Verzerrung mit sich.

Mit der hinzugefügten Anfangswert setzen wir die Abbildung  $\mathbb{R}^+ \rightarrow \mathbb{R}$  um:

$$f' = \log\left(\frac{f + \varepsilon A}{A}\right) \quad (50)$$

wobei  $A$  der Mittelwert der Messpunkte der BRDF ist. Wenn die Reflexionen von Obeflächen, die durch die BRDF beschrieben werden, gleichmäßig verteilt sind, kann  $A$  als Approximation der diffusen Komponente betrachtet werden.

Die Inverse der Abbildung sieht wie folgt aus:

$$f = A \cdot \exp(f') - \varepsilon A \quad (51)$$

Wenn  $\varepsilon$  sehr gering ist, ist die Komponente  $\varepsilon A$  im Vergleich zum Messfehler auch vernachlässigbar. In [McCool01] wird  $\varepsilon = 10^{-5}$  bestimmt. Weil  $f'$  im Produkt das  $A$  benutzt, wird ein  $\sqrt[3]{A}$  durchschnittlich in jede  $p_j$  eingesetzt.

Die Vorteile vom Logarithmus sind nicht nur die positiven Werte sicher zu stellen, sondern auch die großen Werte von BRDFs auf kleine Werte zu reduzieren und zu verhindern, dass sie wegen der Speicherbegrenzung überfließen können. Daher wird der Nachteil durch den Vorteil ausgeglichen.

Beim Rendern nutzen wir ein Produkt von drei Faktoren, um die BRDF zu bestimmen, so dass die Texturen-Abbildungen nicht zu viel sind. Dabei setzen wir auch den Halbvektor  $H$  als einen Faktor ein.

$$f(L, V) = p(L(\pi_j(L, V))) \cdot g(H(\pi_j(L, V))) \cdot p(V(\pi_j(L, V))), \quad \pi_j \in \{\pi_i, \pi_h, \pi_r\} \quad (52)$$

Ebenso bekommen wir eine neue Gleichung durch logarithmisches Homomorphismus auf den beiden Seiten der Gleichung.

$$f'(L, V) = p'(L(\pi_j(L, V))) + g'(H(\pi_j(L, V))) + p'(V(\pi_j(L, V))) \quad (53)$$

Wir nehmen an, die Textur ist schon mit der Koordinate  $(u, v) \in \mathbb{R}^2$  erzeugt und die

Texels (*texture elemente*) für  $p'$  und  $g'$  werden auf ein Gitter verteilt, dazu

brauchen wir eine Interpolationsfunktion über  $\mathbb{R}^2$ .

Wenn die Texels in ganzzahligen Punkten liegen, können wir durch bilineare Gewichtungen die Position der Projektion auf der Textur festlegen.

$$(u_j, v_j) = \pi_j(L, V) \quad (54)$$

$$\begin{aligned}
(U_j, V_j) &= (u_j, v_j), \\
(s_j^u, s_j^v) &= (u_j - U_j, v_j - V_j), \\
(t_j^u, t_j^v) &= (1 - s_j^u, 1 - s_j^v)
\end{aligned}
\tag{55}$$

wobei  $s$  und  $t$  Gewichtungen in zwei Achsen der Textur-Koordinate sind.

Wenn die Gewichtungen für die Projektionen in Einfalls- und Ausfallsrichtung gegeben sind, können wir durch folgende Gleichung für jeden möglichen Punkt die BRDF interpolieren.

$$\begin{aligned}
f' &= t_r^u t_r^v p' [U_r, V_r] + s_r^u t_r^v p' [U_r + 1, V_r] \\
&+ s_r^u s_r^v p' [U_r + 1, V_r + 1] + t_r^u s_r^v p' [U_r, V_r + 1] \\
&+ t_h^u t_h^v g' [U_h, V_h] + s_h^u t_h^v g' [U_h + 1, V_h] \\
&+ s_h^u s_h^v g' [U_h + 1, V_h + 1] + t_h^u s_h^v g' [U_h, V_h + 1] \\
&+ t_i^u t_i^v p' [U_i, V_i] + s_i^u t_i^v p' [U_i + 1, V_i] \\
&+ s_i^u s_i^v p' [U_i + 1, V_i + 1] + t_i^u s_i^v p' [U_i, V_i + 1]
\end{aligned}
\tag{56}$$

Dies ist eine Variante der Fatorisierung mit zwei Faktoren  $P$  und  $G$ , aber mit einem Produkt. Je mehr Produkte von Faktoren zur Berechnung angewendet werden, desto besser ist die Rendernqualität von Oberflächen. Der Vorteil ist jedoch, dass die Darstellung von BRDFs nur zwei Texturen abzubilden braucht, wenn eine der Position von Lichtquelle oder Kamera fixiert wird. Das reduziert den Speicherbedarf und erlaubt, dass viele Materialien in einer Szene gleichzeitig auf einen Rechner mit schlechter Leistung gerendert werden können.

Wie unten angegeben, zeigen wir ein Beispiel von [McCool01].

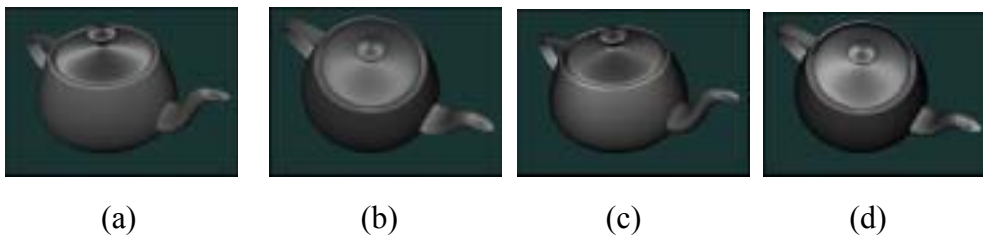


Abbildung 3.22: Rendern der Approximation mit drei Faktoren.  
(a) und (b) sind von BRDF-Modellen gerendert,  
(c) und (d) sind von HF mit drei Faktoren gerendert.

### 3.5.3.5 Verkettete Matrix-Faktorisierung (Chained Matrix Factorization, CMF)

In [Kautz99] wendet Kautz und McCool die Technik der Matrix-Zerlegung wie bei SVD an, um eine BRDF als eine Summe der Produkte von zwei 2D-Funktionen anzunähern.

$$f(L, V) \approx \sum_{k=1}^K p_k(\pi_i(L, V)) q_k(\pi_r(L, V)) \quad (57)$$

Auf diese Weise wird eine 4D-BRDF in eine Menge von 2D-Funktionen zerlegt, welche in  $UV$ -Texturen abgebildet werden können.

In [McCool01] hat McCool Homomorphic Factorization (HF) vorgestellt, in der ein einziges Produkt mit den zwei Faktoren  $P$  und  $G$  zur Approximation von BRDFs eingesetzt werden.

$$f(L, V) \approx \prod_{j=1}^J P_j(\pi_j(L, V)) \quad (58)$$

Im Vergleich zu den beiden Techniken verwendet F. Suykens *Chained Matrix Factorization* (CMF) in [Suykens03] auch zwei Faktoren zur Berechnung der Approximation von BRDFs, aber sie ermöglicht, dass es mehrere Produkte in der Matrix-Zerlegung geben kann.

$$f(L, V) \approx \prod_{j=1}^J \sum_{k=1}^{K_j} p_{j,k}(\pi_{j,i}(L, V)) q_{j,k}(\pi_{j,r}(L, V)) \quad (59)$$

Diese Gleichung entspricht der SVD, falls wir  $J = 1$  einsetzen; sie passt auch zu HF, falls wir für  $K_{j=1}$  und für  $q_{j,k} = 1$  eingeben.

Die Grundidee von CMF ist es, eine Menge unterschiedlicher Matrix-Zerlegungen mit verschiedenen Parametrisierungen zu benutzen, damit wir bei der Approximation mehrere Produkte der Faktoren erhalten können.

Dazu geben wir eine Approximation mit vier Faktoren, die in eine Faktorisierung mit drei Faktoren transformiert werden kann. Wir gehen davon aus, dass eine BRDF aus dem Produkt von Einfallsvektor, Ausfallsvektor, Halbvektor und Differenz-Vektor bestimmt wird.

$$f(L, V) \approx P_i(\pi_i(L, V)) \cdot P_r(\pi_r(L, V)) \cdot P_h(\pi_h(L, V)) \cdot P_d(\pi_d(L, V)) \quad (60)$$

wobei  $\pi_i, \pi_r, \pi_h, \pi_d$  auf die Projektionsfunktionen für Einfall- und Ausfallrichtung, Halbvektor und Differenz-Vektor verweisen.

Sollte eine BRDF durch Parametrisierung des Einfall- und Reflexionsvektors mit Projektionsfunktionen  $\pi_i$  und  $\pi_r$  dargestellt werden, so ist sie identisch mit (43)

und wir bezeichnen sie als  $M_{i,r}$ . Wenn wir eine BRDF durch Parametrisierung des Halbvektor  $H$  und des Differenz-Vektors  $D$  mit Projektionsfunktionen darstellen, wird eine Matrix  $M_{h,d}$  eingebracht, die sich als die diskrete Approximation der BRDF mit GSHD-Parametrisierung darstellen läßt.



$$M_{h,d} = \begin{pmatrix} f(H_1, D_1) & f(H_1, D_2) & \cdots & f(H_1, D_{k-1}) & f(H_1, D_k) \\ \vdots & & & \ddots & \vdots \\ f(H_k, D_1) & f(H_k, D_2) & \cdots & f(H_k, D_{k-1}) & f(H_k, D_k) \end{pmatrix} \quad (61)$$

Dann nutzen wir die SVD-Zerlegung, um  $M_{h,d}$  in ein Produkt der orthogonalen Matrizen anzunähern.

$$M_{h,d} \approx U_h V_d^T = F_{h,d}^{(h,d)} \quad (62)$$

Weil  $M_{h,d}$  nicht aus einer Folge von Produkten gebildet wird, bilden die beiden Vektoren  $U_h$  und  $V_d^T$  keine gute Approximation, aber hierbei sind fast alle Werte positiv. Eine Approximation mit mehreren Produkten enthält mit größerer Wahrscheinlichkeit negative Werte. Durch die zwei Vektoren werden die Faktoren  $P_h(\pi_h(L, V))$  und  $P_d(\pi_d(L, V))$  in (60) hergeleitet.

Der nächste Schritt, der für die Hauptidee steht, ist die Matrix  $F_{h,d}^{(h,d)}$ , die als eine Approximation von  $P_h(\pi_h(L, V))$  und  $P_d(\pi_d(L, V))$  rekonstruiert und in einer GSHD parametrisiert wird, in eine Matrix  $F_{h,d}^{(i,r)}$  durch Parametrisierung der Einfalls- und Reflexionsvektoren zu transformieren. D.h. die transformierte Approximation  $F_{i,r}^{(h,d)}$  wird in originale Einfalls- und Ausfallsvektoren, aber mit den Faktoren  $P_h(\pi_h(L, V))$  und  $P_d(\pi_d(L, V))$  dargestellt.

Als Restmatrix (*residue matrix*) bezeichnen wir die komponentenweise Division der originalen BRDF-Matrix  $M_{i,r}$  durch die neu rekonstruierte Matrix  $F_{i,r}^{(h,d)}$ . Sie definiert die relative Distanz zwischen der originalen BRDF-Matrix und der gebildeten Approximation, die durch zwei Vektoren in der GSHD-Darstellung durch die SVD-Zerlegung erzeugt wird:

$$F_{i,r}^{(res)} = \frac{M_{i,r}}{F_{i,r}^{(h,d)}} \quad (63)$$

Auf die Restmatrix führen wir wieder eine SVD-Zerlegung durch und erhalten dann:

$$F_{i,r}^{(res)} \approx U_i V_r^T = F_{i,r}^{(res;i,r)} \quad (64)$$

Auf dieselbe Weise werden  $P_i(\pi_i(L, V))$  und  $P_r(\pi_r(L, V))$  in (60) durch die zwei Vektoren  $U_i$  und  $V_r$  hergeleitet.

Bis jetzt haben wir schon alle vier Faktoren in (60) bestimmt, wenn wir sie zusammenfassen, erhalten wir die folgende Gleichung:

$$\begin{aligned}
 M_{i,r} &\approx F_{i,r}^{(res)} \otimes F_{i,r}^{(h,d)} \approx F_{i,r}^{(res;i,r)} \otimes F_{i,r}^{(h,d)} \\
 &= U_i \times V_r^T \times U_h \times V_d^T
 \end{aligned}
 \tag{65}$$

Was wir gerade gezeigt haben, ist ein Beispiel für die CMF-Faktorisierung eines Produkts mit vier Faktoren, die jeweils eine eigene Parametrisierung haben. Es können auch mehr als vier Faktoren, z.B. sechs oder acht oder noch mehrere Faktoren eingesetzt werden. Um eine höhere Genauigkeit zu erhalten, können wir sogar in jedem Schritt der Matrix in einer SVD-Zerlegung mehrere Produkte anwenden. Es kommt darauf an, welche Genauigkeit erfüllt werden soll. Die Methode mit vier Faktoren kann auch ohne Umstände auf drei Faktoren reduziert werden. Wenn wir für  $P_d(\pi_d(L,V))=1$  bei allen möglichen Kombinationen von  $L$  und  $V$  einsetzen, bleibt noch die Einfalls- und Ausfallsrichtung und der Halbvektor in (60), so dass den Differenz-Vektor ausgezogen werden kann. Für eine gegebene BRDF-Matrix mit GSHD, können wir den Differenz-Vektor durch Berechnung des durchschnittlichen Vektors von allen Spalten bekommen. In diesem Fall wird  $P_d$  in der Approximation ausgelassen und die CMF funktioniert vom Prinzip her ohne Veränderung. Unten geben wir den intuitiven Prozess von CMF [Suykens03] wieder:

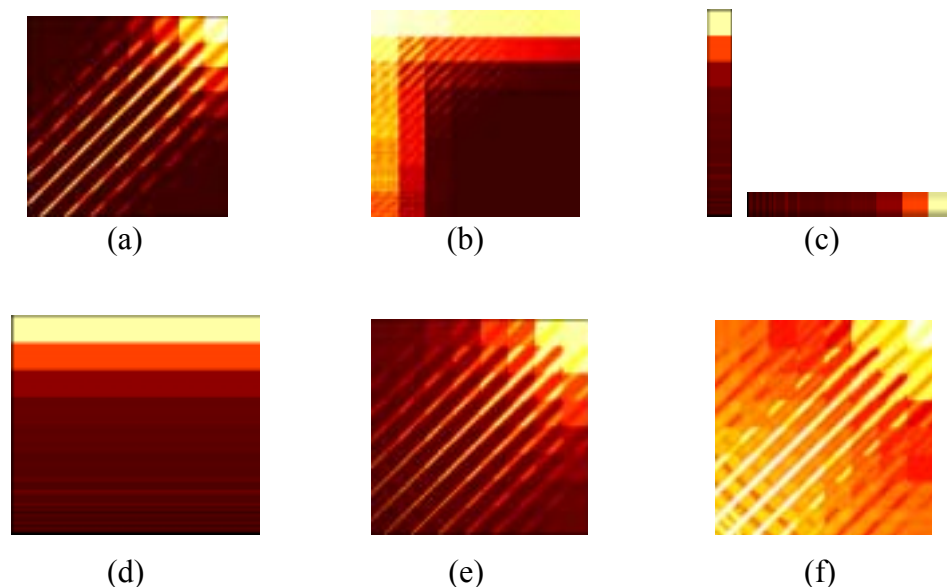


Abbildung 3.23: Chained Matrix Factorization (CMF) [Suykens03]

- (a): Die originale BRDF-Matrix mit IN und OUT,
- (b): Die originale BRDF-Matrix mit GSHD,
- (c): Zwei Vektoren  $U_h$  und  $V_d$  in GSHD,
- (d): Die von  $U_h$  und  $V_d$  gebildete Matrix in GSHD,
- (e): Die transformierte Matrix in IN und OUT von (d),
- (f): Die Restmatrix, die Division von (a) durch (e).

### 3.5.4 Anwendungen von measured-BRDFs

Das Ziel, das wir im letzten Abschnitt mit so vielen linearen Faktorisierungen vorgestellt haben, dient zur Reduzierung des Speicherbedarfs und der effektiven Implementierung von *measured*-BRDFs beim Rendern der computergraphischen Szenen, um die Oberflächen von realistischen Materialien schnell und einfach zu simulieren.

BRDFs definieren die Reflexionseigenschaft von Materialoberflächen und geben an, mit welcher Wahrscheinlichkeit die Strahlung für eine gegebene Richtung der Lichtquelle in einen bestimmten Winkel reflektiert werden sollte. Bei Rendern müssen wir den Vorgang der Strahlung nachbilden, so dass die Messdaten, die wir aus der realen Welt gesammelt haben, wieder in Bilder auf dem Monitor mit akzeptierbarem Informationsverlust transformiert werden. Zur Berechnung des Farbwerts von BRDFs sind drei Vektoren nötig: die Normale der Fläche, die Einfall- und Ausfallsrichtung. An jedem betrachteten Punkt müssen wir zuerst die Normale  $N$  der Oberfläche finden und dann zwei Vektoren, die senkrecht zueinander und zu  $N$  sind, in Abhängigkeit von verschiedenen Abbildungsarten bestimmen, so dass  $N$  und die zwei Vektoren  $s$  und  $t$  ein lokales Koordinatensystem bilden. Sollte ein Koordinatensystem für einen betrachteten Punkt festgelegt werden, können wir die Deklinations-  $\theta_i$  und Azimutwinkel  $\phi_i$  für die einfallende Strahlung, sowie die

Deklinations-  $\theta_r$  und Azimutwinkel  $\phi_r$  für die reflektierte Strahlung ausrechnen. Die vier Werte dienen als Eingaben einer 4D-BRDF.

Folgen wir der Vorgehensweise, so können wir für eine Fläche pixelweise jeden diskreten Punkt behandeln und brauchen uns nicht um die geometrische Informationen des Objektes zu kümmern. D.h. wir berücksichtigen an dem Punkt, der gerendert werden soll nur die Informationen darüber, wie die Normale dieses Punkts berechnet und die reflektierten Werte von der BRDF für eine gegebene Lichtquellen- und Kameraposition interpoliert werden kann. Dazu benötigen wir eine nützliche **Plattform**, auf der unser Shading-Programm (*Shader*) sich direkt mit der Farbtönung und Schattierung von Oberflächen einziehen kann. Es gibt Rendering-Systeme, sie lösen das Geometrieproblem, dienen zur Ermittlung der Sichtbarkeit, liefern die Normalen der Flächen und rendern die Bilder mit den diskreten Pixeln, die auf speziellen Materialoberflächen liegen und durch die interpolierten BRDF-Werte gefärbt werden. Aber solche Rendering-Systeme unterstützen jedoch allgemein nur analytische BRDFs und damit isotrope Beleuchtungsfunktionen. Wollen wir auch anisotrope Effekte erzielen, muß zusätzlich auch Funktionalität zur Unterstützung von akquirierten BRDFs integriert werden, damit sich diese Rendering-Systeme als unsere Plattform darstellen lassen.

Eine gute Alternative ist die Strahlverfolgung (*Raytracing*), die die Lichtstrahlen auf ihrem Weg von der Quelle bis zum Auge (oder Kamera) verfolgt und die Farbe eines Pixels am nächsten Schnittpunkt eines Augstrahls ermittelt. Für eine Szene sind die Positionen der Lichtquelle und der Kamera vorher schon bekannt, aber für einen

Punkt, der als der nächste Schnittpunkt eines Augstrahls mit den Objekten in der Szene betrachtet wird, sollten wir die Normale der Fläche an diesem Punkt ausrechnen. Zur Berechnung der Normalen gibt es vielseitige Methoden. Es kommt darauf an, welche geometrische Form das Objekt hat. Z. B. ist eine Kugel die einfachste zu berechnende Geometrie, weil die Normale jedes Punkts auf der Kugeloberfläche durch diesen Punkt und den Mittelpunkt der Kugel gebildet werden kann.

Beim Programmieren haben wir viele nutzbare *Raytracer* gefunden. Für die Programmiersprache C++ gibt es das erweiterbare System *Rayshade (Version 4.0)* zur Erzeugung der *Raytracing*-Bilder. In den Szenen können Lichtquellen mit verschiedenen Typen eingefügt werden, es werden auch unterschiedliche Primitive, sowie Transformationen von Objekten und Texture eingerichtet. Auf dieser Seite finden wir nicht nur den Quellcode(*source code*) des Packets, sondern auch nützliche Dokumentationen, in denen die Definitionen von *C-Headern*, die Strukturen der Klassen und die Vorschläge zur Erweiterung des Systems abgehandelt werden.

Für die verbreitete Programmiersprache Java gibt es mehrere Varianten zur Auswahl. Davon wenden wir das von Mike Murray im Jahr 2004 programmierte Rings-Raytracer an. Diese ist zu 100% in Java und beinhaltet die Farbwiedergabe-Funktion (*Color rendering*), *Anti-aliasing*, verschiedene Lichtquellen, die primitiven Flächen wie z. B die Kugel, den Kegel oder den Zylinder, sowie Transformationen von Oberflächen, damit wir bei der Entwicklung sehr einfach eine Szene erstellen und unsere eigenen Shader einfügen können.

Die Homepage von Ring-Raytracer und die anderen Quellen von Raytracer werden unten aufgelistet:

Rings-Raytracer: <http://www.cm.cf.ac.uk/Ray.Tracing/>.

POV-Ray: <http://www.povray.org>.

Radiance: <http://radsite.lbl.gov/radiance/HOME.html>

Raytracer-Sammlungen: <http://j3d.sourceforge.net/>

Aktuelle Informationen über Raytracing: <http://www.acm.org/>

In solchen Raytracern haben wir ohne Ausnahme drei Punkte zu erledigen: zuerst fügen wir eine eigene Geometrie ein und die Normale an jedem Punkt der Oberfläche sollte berechenbar sein, um das lokale Koordinatensystem zu bilden und die Transformation des Koordinatensystems zu ermöglichen. Zum zweiten Schritt müssen wir eine BRDF-Funktion für jeden Punkt in Abhängigkeit von der Kombination der Lichtquelle- und Kameraposition aufbauen, so dass die Farbwerte für beliebige Winkel interpoliert werden können. Drittens sollten wir mittels des Raytracers in der Lage sein, die Position für Lichtquelle und Kamera anpassen zu können.

Der Quellcode für die Definition einer neuen Geometrie in Java ist:

```

public interface AbstractSurface {
    abstract RGB3f getColorAt(Point3f point);
    abstract Vector3f getNormalAt(Point3f point);
    abstract boolean intersect(Ray ray);
    abstract Point3f intersectAt(Ray ray);}

```

wobei

***getColorAt***(*Vector3f* point) ein *RGB3f*-Objekt für den Farbwert des betrachteten Punkt der Oberfläche mittels der BRDF-Funktion ausgibt,

***getNormalAt*** (*Point3f* point) den Vektor der Normalen am betrachteten Punkt der Oberfläche ausgibt,

***intersect***(*Ray* ray) den booleschen Wert *TRUE* ausgibt, falls der gegebene Strahl einen Schnittpunkt mit der Fläche eines Objekts hat.

***intersectAt***(*Ray* ray) den Schnittpunkt der Oberfläche mit dem gegebenen Strahl ausliefert, falls ***intersect***(*Ray* ray) *TRUE* ist.

Dann bauen wir eine 4D-Funktion für eine BRDF auf, damit der Intensitäts- oder Farbwert an jedem Punkt für jede Kombination der einfallenden Strahlung und der reflektierten Strahlung durch Interpolation ausgerechnet werden kann.

Der Quellcode für die Speicherung einer BRDF in Java ist:

```

    double theta_i, phi_i, theta_r, phi_r;
    int ti, pi, tr, pr;
    deltat_i = ( $\pi/2$ )/ti;
    deltap_i =  $\pi$ /pi;
    deltat_r = ( $\pi/2$ )/tr;
    deltap_r =  $\pi$ /pr;

    RGB3f val;
    for (int h = 0; h < tr; h++)
    for (int i = 0; i < pr; i++)
    for (int j = 0; j < ti; j++)

```

```

for ( int k = 0; k < pi; k++ )
{
    theta_r = h * deltat_r;
    phi_r = i * deltap_r;
    theta_i = j * deltat_i;
    phi_i = k * deltap_i;
    /* Interpoliere den BRDF-Wert. */
    val = f( theta_i, phi_i, theta_o, phi_o )
    /* Speichere in 4D-Array ab. */
    BRDF[h][i][j][k] = val;
}

```

Wenn es mehr tausend Datensätze der BRDF für eine Materialoberfläche gibt, ist es kaum möglich, eine Szene mit verschiedenen Materialien in Echtzeit zu rendern. Dazu werden wir später im Kapitel von BTF effektive Techniken einbringen, mit denen die Datensätze von BRDFs mit so wenig Speicher wie möglich abgespeichert werden können, z. B. Index-Technik und RGB-Quantisation.

Unter Shader verstehen wir die verfahrenstechnischen Programme oder Recheneinheiten, die frei programmiert werden können und auch wunderschöne 3D-Effekte in Computergraphik erzeugen können. Shader definieren die Eigenschaften der Oberflächen von Materialien, z.B. die Farbe, den Reflexionsgrad und die Lichtdurchlässigkeit. Mit den verschiedenen Typen können Shader in verschiedenen Gruppen angeordnet werden. In unserer Arbeit betrachten wir Shader für Oberflächen von Materialien (*surface shader*).

Der Quellcode für die Definition des Shaders von BRDFs in Java ist:

```

public RGB3f shader(Point3f point, Vector3f cameraDirection,
                    Vector3f lightDirection
                    AbstractSurface surface)
{
    double theta_i, phi_i, theta_r, phi_r;
    Vector3f s, t, L, V;
    RGB3f colorVal;

    L = lightDirection;
    V = cameraDirection;
    Vector3f n = surface.getNormalAt(point);
}

```

```

    /* Finde die Vektoren s und t des lokalen Koordinatensystem */
    s = Transformation.GetAxisS(n);
    t = Transformation.GetAxisT(n);
    /* Berechne die vier Winkel */
    theta_i = anglesCalculation.getThetaIN(n, s, t, L, V);
    phi_i = anglesCalculation.getPhiIN(n, s, t, L, V);
    theta_r = anglesCalculation.getThetaOUT(n, s, t, L, V);
    phi_r = anglesCalculation.getPhiOUT(n, s, t, L, V);
    colorVal = interpolationCalculation(theta_i, phi_i,
                                      theta_r, phi_r);

    return (colorVal); }

```

wobei die *Transformation* eine Klasse der Funktionen für verschiedene Abbildungsarten ist. Im Prinzip ist die Abbildung von BRDFs auf Oberflächen der Materialien sehr ähnlich wie die Textur-Abbildung, die die realistische Wiedergabe von Objekten im 3D-Raum erzeugt, die die Oberflächen-Koordinaten  $(x, y, z)$  auf die Parameter-Koordinate  $(u, v)$  und weiter auf Texture-Koordinate  $(s, t)$  abbilden. Die Projektion von  $(x, y, z)$  auf  $(u, v)$  definiert, wie ein Punkt im 3D-Raum auf einen 2D-Raum projiziert werden sollte, z.B. Zylinder-, Kugel- oder Kegel-Abbildung. Die Projektion von  $(u, v)$  auf die  $(s, t)$  ist sehr flexibel und kann auch definieren, dass Abscheidung (*clipping*), Skalierung (*scaling*), Drehung (*rotating*) oder *Wrapping* der Textur einzusetzen ist. Daran arbeitet unsere Klasse *Transformation*.

Die Klasse *interpolationCalculation* dient zur Berechnung der angepassten Farbwerte, die für die vier eingegebenen Winkel  $theta_i$ ,  $phi_i$ ,  $theta_r$ ,  $phi_r$  durch die Interpolation der Datensätze von gemessenen BRDFs ausgerechnet werden. Im vorherigen Abschnitt haben wir schon die Methoden der Interpolationen für Dreieck und Viereck (oder Polygon) vorgestellt. Sie sind auch bei der Implementierung des Programms anwendbar.

Am Ende geben wir auch den Quellcode für das Hauptprogramm in Java an, das die Einstellung der Positionen der Kamera und der Lichtquelle enthält:

```

public static void main(String args[]) {
    int height = 400; // Die Höhe des Bildes
    int width = 400; // Die Breite des Bildes
    int ssWidth = 1; // Die Breite des Supersamples
    int ssHeight = 1; // Die Höhe des Supersamples

    Abstractsurface surfaces[] = {new CustomSurface()};
    Light lights[] = {new DirectionalAmbientLight()};
}

```

```
Camera camera = new Camera();

Scene scene = new Scene(camera, lights, surfaces);
RGB3f image[][] = RayTracingEngine.render(scene, width, height,
                                     ssWidth, ssHeight, null);
FileEncoder.createImageFile(image, new File("image.jpg"));}
```

Für das obige Beispiel von Raytracern haben wir noch das Problem zu lösen, dass die Normale an jedem Punkt der Oberflächen, die von uns frei definiert werden können, zum Rendern der Szene einfach und schnell berechenbar sein muß. Wenn die meisten Objekte einer Szene primitiv sind, wie z. B. Kugel, Kegel oder Quadrat, so ist das Problem nicht schwer zu lösen. Aber auf der einen Seite ist die Berechnung der Normalen beim Rendern in Echtzeit manchmal aufwendig, auf der anderen Seite gibt es in der Praxis viele Objekte, deren Geometrie sehr kompliziert ist oder von den Primitiven nur mit Überlappungen neu kombiniert werden können. In einem solchem Fall kann man die Oberflächen von komplexen Objekten durch Gitter mittels 3D-Modellierung darstellen, innerhalb jeder Zelle des Gitters werden die Normalen und Shading-Werte an allen Eckpunkten ermittelt und die anderen Punkte werden interpoliert. In dieser Arbeit werden wir über das Thema nicht weiter diskutieren und nutzen das Werkzeug *Blue Moon Rendering Tools (BMRT, Version 2.6)* von RenderMan, das die Normale an jedem Punkt von beliebigen Oberflächen bei der operativen Definition von *Shaders* ausliefern kann.

Das RenderMan ist ein Werkzeug einer von den Pixar Animation Studios entwickelten Technologie für das Rendern von Computergrafiken, welches eine Implementierung von Pixars photorealistischen 3D-Beschreibungsstandard und einen *Rendering-Interface-Standard* darstellt. RenderMan erlaubt es beim Modellieren zu spezifizieren, was und wo gerendert werden muß, ohne genau festzulegen, welcher Algorithmus benutzt werden soll.

Pixar Animation Studios entwickelte RenderMan um "*Modeler*" für geometrische Definition und "*Renderer*" für Farbwiedergabe von Oberflächen voneinander zu trennen und unabhängig voneinander zu warten. Ein *Modeler* wird dazu benutzt, Szenen zu zeichnen und Animationen zu entwerfen. Die einzige Aufgabe eines *Renderers* ist es, die mit dem *Modeler* entworfenen Szenen so realistisch wie möglich zu zeichnen. Der *Renderer* ist für die lebensechte Erzeugung von Schatten, Lichtfall und Texturen verantwortlich. Ein *Modeler* sollte sich nicht um das Rendern kümmern müssen. Ein *Renderer*, der den von RenderMan spezifizierten Standard erfüllt, kann eine Vielfalt von Methoden zum Zeichnen der Objekte benutzen, damit wir die Chance gewinnen können, unsere eigene Zeichen-Methode mit BRDFs in *Renderer* einzusetzen.

Um unser Ziel zu erreichen, werden wir mit Hilfe von *Blue Moon Rendering Tools (BMRT)* arbeiten, die als eine Alternative zu RenderMan bezeichnet wird und komplett von Larry Gritz geschrieben wurden. Das Paket von *BMRT 2.6* ist im



Internet frei herunterladbar (Im Moment eventuell nicht mehr) und die Seite mit nützlichen Informationen lautet:

<http://www.seas.gwu.edu/student/gritz/bmrt.html>.

Da wir die 3D-Beschreibung einer Szene als einen *Modeler*, der in RIB (Interface Bytestream)-Dateien von RenderMan spezifiziert wird, und einen *Renderer*, der durch Shader-Dateien von RenderMan definiert wird, ansehen können, brauchen wir nur unsere eigenen Shader in *Shading Language* zu programmieren, ohne genau zu wissen, für welche Geometrie, in welcher Szene bzw. in welchen RIB-Dateien sie benutzt werden.

In späteren Kapiteln werden wir das RenderMan für BTF näher betrachten. Hier zeigen wir nur einen Teil des Quellcodes für ein einfaches Beispiel in BRDF-Implementierung, um einen ersten Einblick in dieses Programm zu geben.

Der Quellcode für Attribute-Definition eines Walles mittels der von uns eingefügten Surface-Shader *brdfShader* in RIB-Datei ist:

```
FrameAspectRatio 1
Format 400 400 1
PixelSamples 2 2
Projection "perspective" "fov" [37]
WorldBegin
# Attribute-Definition für den Wall in Cornell-Box
AttributeBegin
Attribute "identifier" "name" "Cornell Box -- tall box"
Surface "brdfShader"
Color [ 1 1 1 ]
AttributeEnd
WorldEnd
```

Der Quellcode für *brdfShader* in *Shading Language* von RenderMan ist:

```
surface brdfShader() {
uniform float scale=1;
varying vector Vn, Nn, Ln, Sn, Tn, projector_V;

local_n = faceforward(normalize(N),I);
local_v = normalize(dPdv);
local_u = normalize(dPdu);
```

```

/* illuminance(P,Nn,PI) entire sphere */
illuminance (P, local_n, PI/2){
Ln = normalize(L);
Vn = -normalize(I);
/*
* calculate thetai, phi, thetar, phir
*/
theta_i = acos(Ln.local_n);
theta_r = acos(Vn.local_n);
projector_V = Vn - (Vn.local_n)*local_n;
phi_r = acos(projector_V.local_v);

/*
* Führe die Interpolation für den betrachteten Punkt durch
*/
color color_lowThetai = Thetai_Interpolation(lowlimit,theta_r,phi_r);
color color_upperThetai = Thetai_Interpolation(upperlimit,theta_r,phi_r);
/* Die gesamte Farb */
Ci = (color_lowThetai*(1-tt)+color_upperThetai*tt)*5*scale;
} // END-illuminance
} // END-brdfShader

```

### 3.6 Zusammenfassung

BRDFs sind gute Techniken, um die Materialoberflächen im 3D-Raum effektiv und präzise beschreiben zu können. Ab dem nächsten Kapitel kommen wir zum Hauptpunkt dieser Diplomarbeit, in dem wir eine bessere Technologie, die *Bidirectional Texture Function* (BTF) auf der Basis von BRDFs vorstellen und ermitteln.

# Kapitel 4

## BTF-Akquisition

### 4.1 Definition von BTFs

Eine der Hauptaufgaben von Computergrafik ist es, photorealistisches Material von Objekten im virtuellen System zu rendern. Im obigen Kapitel haben wir die Technik von BRDFs zur Simulation der Materialoberflächen vorgestellt. Die wichtigste Voraussetzung der Anwendung von BRDFs ist, dass wir die Textur und die Struktur einer Materialoberfläche gleichmäßig ansehen können. Jedes Element der Fläche hat dieselbe Reflexionseigenschaft, die inneren Effekte wie z. B. die Schattierung, Innerreflexion und Ausblendung der Flächenelemente voneinander werden aber vernachlässigt. Um Bilder mit noch höherer Qualität zu zeichnen, können wir solche realistische Materialien als 2D-Textur unter verschiedenen Kombinationen von Licht- und Kamerarichtungen beschreiben, die als Technologie der BTF bezeichnet wird. BTF ist eine 6D-Funktion. Sie hängt von der 2D-Textur-Koordinate  $(s, t)$ , sowie den 4D-Winkeln vom Licht und der Reflexion auf einer Hemisphäre ab. In der Praxis sollte man sich eine Menge von mehreren tausend Bildern für das Sample eines Materials bei regelmäßigen Licht- und Kamerapositionen besorgen, um die Reflexionseigenschaft dieses Materials durch BTF beschreiben zu können. Wenn wir den Mittelwert der gemessenen Intensität von jedem Bild der BTF berechnen, dann erhalten wir eine BRDF-Messung, denn die BRDF nur ein Spezialfall von BTF.

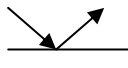


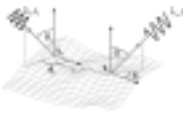
Flächen		<i>Grobe Oberfläche</i>	<i>Feine Oberfläche</i>
Licht/Kamera			
<i>fixierte Licht/Kamera</i>	Reflexion		Textur 
<i>varierte Licht/Kamera</i>	BRDF		BTF 

Abbildung 4.1: Vergleich des Erscheinungsbildes der Flächen

## 4.2 Akquisition von BTFs

### 4.2.1 BTF-Datenbank von CURet

Der Begriff BTF wurde von K. J. Dana und seinen Kollegen im Jahr 1999 [Dana99] erst eingebracht, sie haben auch das erste Messsystem für BTFs-Akquisition aufgebaut, in dem ein Roboter-Manipulator und eine CCD-Kamera zum Einsatz kamen. Statt wie bei BRDF einen Punkt auf einer Fläche zu messen, sollte eine Menge der Textur-Bilder für ein kleines ebenes Sample (10 x 10 cm) aufgenommen werden.



Abbildung 4.2: Das Messsystem und das festgehaltene Sample [Dana99]

K. J. Dana hat eine Datenbank mit mehr als 60 verschiedenen Materialien aus der Realität aufgestellt. Jedes Sample hat 205 Textur-Bilder unter unterschiedlichen Kombinationen von Licht- und Reflexionsrichtungen. Jedes aufgenommene Textur-Bild hat eine Auflösung von 640 x 480 Pixel, 24 Bits (8 Bits per R/G/B-Kanal). Bei der Akquisition werden die Messpunkte hinter einem Gitter auf die Oberfläche einer Hemisphäre verteilt, bzw. sind die Richtungen für die Lichtquelle und die Kameraposition über die Hemisphäre festgelegt. Die Richtung der Kamera variiert in den 7 verschiedenen Positionen  $22.5^\circ$ ,  $45^\circ$ ,  $67.5^\circ$ ,  $90^\circ$ ,  $112.5^\circ$ ,  $135^\circ$ ,  $157.5^\circ$  (Abstand =  $22.5^\circ$ ) für den Azimutwinkel, es werden 55 Bilder in Position 1 aufgenommen; 48 Bilder in Position 2; 39 Bilder in Position 3; 28 Bilder in Position 4; 19 Bilder in Position 5; 12 Bilder in Position 6; 4 Bilder in Position 7. Es wird für jede Position der Kamera bei der Aufnahme von Bildern verlangt, dass die Orientierung des Samples sichtbar ist und von der Strahlung der Lichtquelle beleuchtet werden kann.

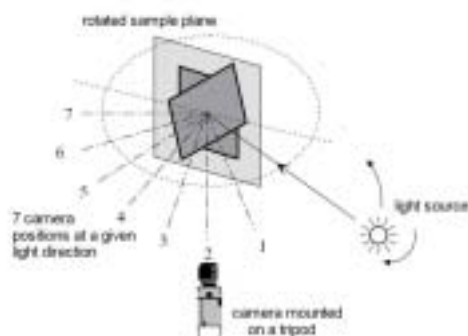


Abbildung 4.3: Die Einrichtung des Messprozesses [Dana99]

#### 4.2.2 BTF-Datenbank von Bonn

Eine der besten Datenquellen von BTFs [Müller04], die wir bei der Implementierung unserer Programme zum Rendern von 3D-Szenen häufig benutzen werden, kommt aus der Universität-Bonn. Statt Spectrometern benutzten G. Müller und seine Kollegen Gonioreflectometer mit CCD-Chips (Kodak DCS pro 14 N), um die räumliche Variation der Reflexion aufzunehmen. Aktuell haben sie die Oberflächen der Materialien *WOOL*, *WALLPAPER*, *IMPALLA*, *PROPOSTE*, *CORDUROY* aus der realen Welt gemessen. Für jedes Sample wurden die Messpunkte in 81 Richtungen für die Lichtquelle auf einer Hemisphäre bestimmt und für die jeweiligen Richtungen der Lichtquelle gibt es 81 Richtungen auf die CCD-Kamera. Von den 81 Richtungen wurde von 00 bis 750, alle 150 in Deklinationswinkel gemessen. Es werden insgesamt 6561 Bilder mit 4500 x 3000 Pixels (ohne Kompression) für ein Sample nach einer Akquisition aufgenommen. Tabelle 4.1 zeigt die Verteilung der Richtungen für Deklinations- und Azimutswinkel:

$\theta = 0^\circ :$	$\phi = 0^\circ$
$\theta = 15^\circ :$	$\phi = 0^\circ, 60^\circ, 120^\circ, 180^\circ, 240^\circ, 300^\circ$
$\theta = 30^\circ :$	$\phi = 0^\circ, 30^\circ, 60^\circ, 90^\circ, 120^\circ, 150^\circ, 180^\circ, 210^\circ, 240^\circ, 270^\circ, 300^\circ, 330^\circ$
$\theta = 45^\circ :$	$\phi = 0^\circ, 20^\circ, 40^\circ, 60^\circ, 80^\circ, 100^\circ, \dots, 260^\circ, 280^\circ, 300^\circ, 320^\circ, 340^\circ$
$\theta = 60^\circ :$	$\phi = 0^\circ, 18^\circ, 36^\circ, 54^\circ, 72^\circ, 90^\circ, \dots, 270^\circ, 288^\circ, 306^\circ, 324^\circ, 342^\circ$
$\theta = 75^\circ :$	$\phi = 0^\circ, 15^\circ, 30^\circ, 45^\circ, 60^\circ, 75^\circ, \dots, 285^\circ, 300^\circ, 315^\circ, 330^\circ, 345^\circ$

Tabelle 4.1: Die Richtungen der Messung

Nachdem die Messung durchgeführt wurde, werden die rohen Bilder in eine Menge von korrigierten und eingetragenen 2D-Textur-Bilder transformiert, um die perspektivischen Effekte zu beseitigen und den Speicherbedarf zu reduzieren. Um die perspektivischen Bilder auf eine Ebene ( $\theta = 0^\circ, \phi = 0^\circ$ ) zu projizieren, wird eine Grenzlinie als ein Marker auf der Sample-Ebene eingesetzt. Bei der Kodierung der rohen Bilder in digitale Bilder werden Pixel für Pixel auf 2D-Texture durch die Transformation, die mit Hilfe des Markers vorher ausgerechnet wird, mit angemessenem Farbwert versehen.



Abbildung 4.4: Gemessenes BTF-Sample *WALLPAPER* [Müller04]

Links: das berichtigte Bild

Rechts: die Originalaufnahme mit Perspektive

Um die Größe der Bilder zu reduzieren, werden die von der Kamera aufgenommenen 12 Bit-RGB-Bilder in 8 Bit-RGB-Bilder konvertiert und die Auflösung auf 256 x 256 verringert. Am Ende besteht jede Datenbank für ein Sample aus einer Menge von 6561 Rasterbildern mit 256 x 256 Pixels, der Speicherplatz beträgt ca. 400 MB.

#### 4.2.3 Andere BTF-Datenbanken im Internet

Außer den beiden vorgestellten Datenquellen sind im Internet noch viele weitere Datenbank-Quellen für BTF-Messungen erhältlich.

Die *PhoTex*-Datenbank ist eine Textur-Datenbank für grobe Oberflächen. Sie enthält Bilder verschiedener grober Oberflächen, die von Lichtquellen aus verschiedenen Richtungen beleuchtet wurden.

Die *VisTex*-Datenbank ist von *Vision und Modelling Group* im MIT-Media-Labor aufgebaut worden. Die wichtigste Eigenschaft dieser Datenbank ist, dass sie sich der Perspektive einer Ebene nicht anpasst und die dabei eingesetzten Lichtquellen vielseitig sind, z. B. Tageslicht. Wegen der ungenauen Richtung der Lichtquelle ist es sehr schwer, die Messdaten bei der Implementierung unserer Programme anzuwenden.

*MeasTex* ist eine Datenbank von Bildern für Textur-Analysialgorithmen. Sie ist nicht nur eine Textur-Dankbank, sondern auch eine quantitative Messung der Algorithmen der Textur. Aber leider sind die meisten angebotenen Bilder 2D-Texturen und nicht 3D-Texturen, deshalb wissen wir nicht, unter welcher Orientierung der Lichtquelle die Bilder aufgenommen wurden, weswegen sie zur Anwendung nicht geeignet ist.

*OUTex* ist eine Textur-Dankbank und gleichzeitig auch ein System für die Klassifizierung der Textur und Segmentierungsalgorithmen. Für jedes Sample werden Bilder unter drei bestimmten Beleuchtungsrichtungen, sechs räumlichen Auflösungen (100, 120, 300,360, 500 and 600 dpi) und neun Rotationswinkeln (0°, 5°, 10°, 15°, 30°, 45°, 60°, 75° and 90°), also insgesamt 162(3 x 6 x 9) verwendet, jedes Bild hat ein Format von 24Bit-RGB. Auf der Homepage werden auch erfolgreiche Anwendungsprogramme der *OUTex* und ein Vergleich dazu angegeben.

Im Vergleich der BTF-Datenbank *BonnTex* zu den anderen Datenquellen können wir feststellen, dass *BonnTex* einen überwältigen Vorteil besitzt. Die Richtungen für Lichtquelle und Kamera sind eindeutig und regelmäßig bestimmt, jedes Bild hat die passende Auflösung und der Einfluss der Perspektive wird auch korrigiert. Des

Weiteren ist noch nennenswert, dass es auf der Homepage von Bonn viele verwendbare Dokumentationen für die Kompression und die Synthese der BTFs gibt. Wir werden *BonnTex* in unserer Arbeit als Implementierungsstoff verwenden. Es folgt die Adressliste:

VisTex (*Vision Texture Database, MIT*):

<http://vismod.media.mit.edu/vismod/imagery/VisionTexture/vistex.html>

MeasTex (*Texture Database, the university of Queensland*):

<http://www.cssip.uq.edu.au/meastex/meastex.html>

OUTex (*University of Oulu, Finland*):

<http://www.outex.oulu.fi/outex.php>

PhoTex (*Texture Lab, Heriot-Watt University*):

<http://www.cee.hw.ac.uk/texturelab/>

CURet (*Columbia-Utrecht Reflectance and Texture Database, Columbia University*):

<http://www1.cs.columbia.edu/CAVE/curet/>

BonnTex (*Texture Database, Bonn University*):

<http://btf.cs.uni-bonn.de/index.html>

# Kapitel 5

## BTF-Kompression

### 5.1 Zwei Darstellungen von BTFs

Wir haben im obigen Kapitel erwähnt, dass BRDF die Reflexionseigenschaft jedes Punkts auf einer Materialoberfläche beschreibt, auf der die Punkte diskret und unter fast identischen Licht- und Kamerarichtungen sind. Aber eine BTF besteht aus einer Menge von Bildern, die unter verschiedenen Kombinationen von Licht- und Kamerarichtungen aufgenommen wurden. Die in einem solchen Akquisitionsprozess generierten Datenmengen sind sehr groß und auf solchen Daten basierende komplexe BRDF- und BTF-Modelle bereiten beim Rendering vielfältige Probleme, insbesondere im Bezug auf den notwendigen Speicherplatz. Z. B. jedes BTF-Sample der Universität-Bonn enthält eine Menge von 6561 Textur-Bildern, die Auflösung von 256\*256 Pixel haben. Der Speicherplatz beträgt ca. 400 MB zur Aufnahme eines solchen BTF-Modells. Will man eine Szene von Materialoberflächen mittels mehr als zehn BTF-Samples rendern, wird mindestens 4 GB zum Einsatz gebraucht. Aus diesem Grund ist es sehr nötig, dass BTFs mit riesigen Datenmengen durch verschiedene Verfahren komprimiert werden können.

Hierzu haben wir zwei Arten eine BTF darzustellen. Eine häufig verwendete Darstellung von BTFs ist es, dass die BTF als eine Menge von 2D-Texturen der Bilder bezeichnet wird, wobei jede 2D-Textur eine bestimmte Kombination aus Licht- und Kamerarichtung definiert.

$$\{Texture_{(L,V)}\}_{(L,V) \in M} \quad (66)$$

hierbei bezeichnet  $M$  die diskrete Menge der Kombinationen der Licht- und Kamerarichtungen. Die Größe der Menge hängt davon ab, wie genau die Durchführung der Messung eingestellt wird. Z. B. hat sie eine Höhe von 205 bei der *CURet*-Datenbank,  $81 \times 81 = 6561$  bei der *BonnTex*-Datenbank.

Die andere Darstellung ist eine BTF als eine diskrete Menge der einzelnen BRDFs zu bezeichnen, die sogenannten *Apparant BRDFs* (ABRDF) [Tong97]. Die Idee ist dabei, die 2D-Textur als eine diskrete Menge der Pixels zu zerlegen. Jedes Pixel zeigt mit derselben Position bei allen Bildern eine ABRDF an.



$$\{ABRDF_{(s,t)}\}_{(s,t) \in N^2} \quad (67)$$

wobei  $N^2$  die Koordinate der Textur bezeichnet, sie hängt von der Auflösung der 2D-Texture ab. Z. B.  $256^2$  bei der *BonnTex*-Datenbank.

BTF ist eine 6D-Funktion. Sollten wir eine BTF als eine Menge der ABRDFs darstellen, so reduzieren wir die Höhe der Dimension der BTF auf die niedrigere Dimension der ABRDF. Aber eine solche ABRDF ist eigentlich keine echte BRDF, sie kann nicht allein die Reflexionseigenschaft des Punktes, den sie bezeichnet, durch die von den Bildern ausgegebenen Datensätze beschreiben. Der Grund dafür ist, dass die Datensätze jeder ABRDF noch die Auswirkung der Abschattung, Reflexion und Abblendung von den anderen ABRDFs enthalten. Die Darstellung einer BTF als eine diskrete Menge der ABRDFs ist nur eine Approximation, die zur weiteren Anwendung wie z. B. Faktorisierung und Synthese von BTFs dient.

## 5.2 Chained Matrix Factorization (CMF) für BTF

Nachdem wir eine BTF in eine Menge der ABRDFs zerlegt haben, können wir die ABRDFs als normale BRDFs ansehen und anschließend lineare Faktorisierungen darauf durchführen. Im Kapitel 3 haben wir schon einige nützliche Methoden vorgestellt, die alle zur Faktorisierung von BRDFs einsetzbar sind. Der Unterschied dazwischen ist nur, bei der Approximation einer BTF haben wir nun eine Menge der zerlegten Faktoren, die zum Wiederaufbau der BTF abgespeichert werden müssen. Z.B. haben wir  $256 \times 256 \times 3$  Faktoren zu einer BTF, falls wir *BonnTex*-Datenbank anwenden und jede ABRDF in ein Produkt von drei Faktoren zerlegen.

Um den Speicherbedarf einer BRDF zu reduzieren, kann die BRDF durch *Chained Matrix Factorization (CMF)* als ein Produkt von drei Faktoren (zwei oder mehrere) bezeichnet werden:

$$f(L, V) \approx P(\pi_i(L, V))Q(\pi_h(L, V))R(\pi_r(L, V)) \quad (68)$$

Zur Approximation einer BTF werden wir ein Produkt von drei Faktoren für jede ihrer ABRDF produzieren. Wenn wir die Faktoren in drei **Clusters** versammeln, verweisen die drei **Clusters** auf die entsprechenden drei Faktoren von allen ABRDFs:

$$\begin{aligned} \text{Cluster}_P &= \{P(\pi_i(L, V))_k\}_{k \in N^2} \\ \text{Cluster}_Q &= \{Q(\pi_h(L, V))_k\}_{k \in N^2} \\ \text{Cluster}_R &= \{R(\pi_r(L, V))_k\}_{k \in N^2} \end{aligned} \quad (69)$$

Bei jeder der drei **Clusters** können wir ihre Elemente bzw. ihre Faktoren in eine bestimmte Skalierung normalisieren und indexieren, um die ähnliche Faktoren zusammenzufassen und weitere Speicherersparnis zu gewinnen. Nun transformieren wir ein Problem der Abspeicherung von BTFs in ein Problem der Kodierung und Indexierung, zu denen es schon mehrere Verfahren gibt. Da die Faktoren der drei **Clusters** normalisiert werden, sind wir in der Lage, die Faktoren  $P$ ,  $Q$  und  $R$  in die entsprechende Farbwerte  $R$ ,  $G$  und  $B$  des RGB-Farbsystems(oder XYZ- und HSV-Farbsystem) zu konvertieren, so dass eine 2D-Texture für die drei **Clusters** erzeugt werden kann. Diese Texture ist relative günstiger abzuspeichern und für die Anwendung durch Interpolation.

Für viele BTFs sehen die unterliegende kleinen Strukturen (*mesostructure*) der Oberflächen in der realen Welt repetitiv aus. Viele ABRDFs, die zu einer BTF gehören, sind sehr ähnlich, obwohl sie vielleicht in verschiedene Positionen der 2D-Texture liegen können, dazu sind ihre Faktoren sicher auch ähnlich. Wir entwickeln eine Vergleichsmethode von BRDFs und komprimieren die Menge der ABRDFs zu einer kompakteren Menge.

Um zwei Vektoren in 3D-Raum zu vergleichen, benutzen wir normalerweise die zweite Norm:

$$\begin{aligned} \vec{V}_1 &= (x_1, y_1, z_1), \vec{V}_2 = (x_2, y_2, z_2) \\ Abstand(\vec{V}_1, \vec{V}_2) &= \sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \end{aligned} \quad (70)$$

Wenn wir eine ABRDF als einen Vektor mit so vielen Dimensionen wie Anzahl der Kombinationen von Licht- und Kamerarichtungen zur Messung ansehen können, dann ist der Vergleich zwischen den ABRDFs im Prinzip genau wie der Abstand zwischen den Vektoren.

$$\begin{aligned} ABRDF_i &= (a_i^1, a_i^2, a_i^3, \dots, a_i^M) \\ ABRDF_j &= (a_j^1, a_j^2, a_j^3, \dots, a_j^M) \\ Abstand(ABRDF_i, ABRDF_j) &= \sqrt{(a_i^1 - a_j^1)^2 + (a_i^2 - a_j^2)^2 + (a_i^3 - a_j^3)^2 + \dots + (a_i^M - a_j^M)^2} \end{aligned} \quad (71)$$

Durch das Vergleichsverfahren von ABRDFs können wir die ähnlichen ABRDFs mit demselben Index verzeichnen, um das Ziel der Speicherersparnis zu erreichen.

Da eine BTF durch lineare Faktorisierung ihrer ABRDFs komprimiert werden kann, kann sie auch durch eine Menge der anpassenden *Analytical*-BRDF-Modelle wie z. B. Phong-, Blinn-, Ward-, oder Lafortune-Modell pixelweise ersetzt werden. Beim Rendern der 3D-Szenen haben *Analytical*-BRDF-Modelle großen Vorteil, sie brauchen nicht die riesigen gemessenen Datensätze in Echtzeit in Speicher einzuladen und benötigen nur ein Paar Parameter zur Anwendung. Aber weil die meisten *Analytical*-BRDF-Modelle für spezielle Klassen von gleichartigen Material-

oberflächen entwickelt werden und im Prinzip nur zur Approximation der Reflexionseigenschaft der Materialien durch physikalische Theorie dienen, verhindert die Darstellung einer BTF durch Ersetzung von *Analytical*-BRDF-Modelle die Qualität der gerenderten Bilder und die realistische Aussicht der virtuellen Objekte.

### 5.3 Principal Component Analysis(PCA)

Statt die diskrete Menge der ABRDFs zu einer BTF anzunähern, bezeichnen wir die BTF als eine kompakte Datenbank der Datensätze und versuchen, die lineare Zerlegung direkt darauf durchzuführen.

Wie wir oben bereits erwähnt haben, kann eine BRDF als eine Matrix, deren Spalten und Zeilen auf Licht- und Reflexionsrichtungen (oder umgekehrt) verweisen, dargestellt werden. Auf dieselbe Weise kann man eine BTF auch als eine riesige Matrix darstellen. Die Spalten dieser Matrix entsprechen der Auflösung der 2D-Textur und jede Spalte wird durch die Pixelwerte einer 2D-Texture gebildet, die Zeilen dieser Matrix entsprechen den eingestellten Einfalls- und Ausfallsrichtungen der Messung und jede Zeile indiziert eine ABRDF der BTF. Dazu nehmen wir auch das Beispiel von *BonnTex*:

$$M_{BTF} = \begin{pmatrix} f_1(L_1, V_1), f_1(L_1, V_2), \dots, f_1(L_i, V_{j-1}), f_1(L_i, V_j) \\ f_2(L_1, V_1), f_2(L_1, V_2), \dots, f_2(L_i, V_{j-1}), f_2(L_i, V_j) \\ \vdots \\ f_k(L_1, V_1), f_k(L_1, V_2), \dots, f_k(L_i, V_{j-1}), f_k(L_i, V_j) \end{pmatrix} \quad (72)$$

wobei  $k \in N^2 = 256 \times 256$  und  $(i, j) \in M$  mit  $i = j = 81, \#M = 81 \times 81$ .

Die Hauptkomponentenanalyse (*Principal Components Analysis, PCA*) [Jolliffe92] ist eine Methode der multivarianten Verfahren in der Statistik und dient oft zur Vereinfachung von großen Datenmengen. Sie ist mit der Faktorenanalyse eng verwandt und auch eine lineare Transformation, die die Daten der Datenmenge in ein Koordinatensystem mit niedrigerer Dimension projizieren kann. Das Ziel ist es, die größte Varianz jeder Projektion der Datenmenge an die erste Hauptachse (*Principal Component*) zu legen, die zweitgrößte Varianz an die zweite Hauptachse zu legen, usw..... Zum Ende wählen wir die ersten  $c$  Hauptachsen zur Darstellung der Datenmenge mit möglichst geringem Informationsverlust aus. PCA kann zur Reduzierung der Dimension der Datenmenge angewendet werden und dabei die Charakteristika der Datenmenge noch zu erhalten.

Bei der BTF-Kompression müssen wir die angepassten  $c$  Hauptachsen für eine BTF-Matrix bzw. eine unabhängige Basis für den BTF-Raum finden, so dass alle Daten der Datenmenge in die Hauptachsen projiziert werden können und die entsprechenden Parameter für jeden Datensatz auszurechnen sind. Beim Beispiel von

BonnTex führen wir die PCA auf die Spalten der BTF-Matrix durch und finden die ersten  $c$  Hauptkomponenten, die sogenannten Eigennormtexturen (*eigen Textures*), damit jedes Bild der BTF eines Samples durch die  $c$  Eigennormtexturen mit angepassten Parametern dargestellt werden kann.

Wir zeigen ein Beispiel von *WOOL*-BTF des *BonnTex*:



Abbildung 5.1: Das originale Bild des *WOOL*-Materials [Sattler03]

Durch PCA kann die ganze Menge der Bilder  $B_{(L,V) \in M}$  für *WOOL*-BTF durch z. B.  $c=7$  Eigennormtexturen  $E_c = E_0, E_1, E_2, E_3, E_4, E_5, E_6$  mit Parametern dargestellt werden:

$$B_{(L,V)} = \sum_{c=0}^6 p_{(L,V)}^c \cdot E_c \quad (73)$$

wobei  $p_{(L,V)}^c$  die entsprechende Parameter sind.



Abbildung 5.2: Die Eigennormtexture [Sattler03]

(von links) Eigennormtexture  $E_0, E_1, E_2, E_3, E_4, E_5, E_6$

## 5.4 Singular Value Decomposition (SVD)

Im Prinzip ist Singular Value Decomposition (SVD) eine Methode zur Bestimmung von Principal Components (PC).

Wir haben die SVD-Zerlegung bereits im obigen Kapitel über die Faktorisierung von BRDFs betrachtet. Eine Matrix  $M$  mit  $m \times n$  Elementen kann durch

$M = UDV^T$  zerlegt werden.  $U$  und  $V$  sind orthogonale Matrizen, die durch Spalten bzw. die Eigenvektoren gebildet werden.  $D$  ist eine Diagonalmatrix, ihre von Null verschiedenen Werte liegen auf der Diagonalen und sind Eigenwerte der Matrix.

Im Vergleich zu zur obigen Faktorisierung bei BRDF werden die durch SVD

zerlegten Eigenvektoren der BTF-Matrix Eigennormtexturen, die wir gerade gezeigt haben.

Der Nachteil des SVD sowie des PCA ist, dass negative Werte in den Eigenvektoren oder in den Hauptkomponenten auftreten können, wenn wir mehrere Eigenvektoren oder Hauptachsen zum Wiederaufbau der Matrix einsetzen wollen. Das eignet sich nicht so gut zur graphischen Hardware-Implementierung, weil man bei der Anwendung häufig die Werte in einer Textur abspeichern will und für das RGB-Farbsystem positive Werte benötigt.

## 5.5 Per-view Matrix Factorization (PVMF)

Obwohl die oben vorgestellten Faktorisierungsmethoden für BRDF zur Approximation der ABRDFs einer BTF auch einsetzbar sind, haben sie ein Problem bezüglich des Speicherbedarfs, wenn die Anzahl der ABRDFs bzw. die Pixel der 2D-Bilder von BTFs zu hoch ist. Z. B. die BTF mit  $128^2$  Pixels bei *CUReT* enthält  $128^2=16k$  ABRDFs, bei *BonnTex* sogar  $256^2=64k!$  Auf der anderen Seite schöpft die Faktorisierung der ABRDFs die räumliche Kohärenz zwischen den Pixeln der BTF nicht aus. Die unlinearen Effekte wie z. B. eigene Abschattung, Verdeckung und Ablendung voneinander werden nicht berücksichtigt.

Aber wenn wir die Faktorisierungsmethoden wie PCA oder SVD direkt auf eine Matrix der ganzen BTF durchführen, tritt beim Speicherbedarf ein erhebliches Problem auf. Eine solche Matrix, die von allen Datensätzen der BTF gebildet wird, kann leicht die Menge von ein Paar hundert Megabytes in Float-Präzision erreichen. In einem solchem Fall benötigt die Berechnung der Hauptkomponenten und Rekonstruktion der BTF-Matrix eine lange Zeit. Deshalb passt die Faktorisierung auf die ganze BTF nur zu den BTF-Samples mit wenigen Datensätzen oder zu einer Teilmenge der BTF.

Um den räumlichen Zusammenhang zwischen den ABRDFs zu erhalten, benutzen wir die Darstellung der BTF als eine Menge von 2D-Bildern (z. B.  $81 \times 81$  2D-Bilder eines Samples bei *BonnTex*). Nun ist es unsere Aufgabe, ein passendes Verfahren zur Zerlegung der Menge einer BTF zu finden, bzw. wie die Bilder einer BTF in Teilmengen verteilt werden sollten. In [Müller04] wird PCA für jede Kamerarichtung (*Per-view Matrix Factorization*) bei *BonnTex* durchgeführt. Die 6561 Bilder eines BTF-Samples werden in 81 Teilmengen klassifiziert, jede Teilmenge steht für eine bestimmte Kamerarichtung und die PCA ist auf jede Teilmenge unabhängig voneinander einsetzbar. Zum Ende werden die ersten  $c$  Hauptkomponenten der jeweiligen Teilmengen als die Approximation der BTF bezeichnet.

$$M_{V_i} = (\text{Texture}_{L_0,V}, \text{Texture}_{L_1,V}, \dots, \text{Texture}_{L_i,V}) \quad (74)$$

wobei  $M_{V_i}$  die Bezeichnung einer Teilmenge der Bilder eines BTF-Samples

und  $L_{V_i}$  die Menge der eingestellten Lichtrichtungen für eine gegebene Kamerarichtung ist. Hierbei nehmen wir das Beispiel von *BonnTex*,  $\#L_{V_i} = 81$ .

Nachdem wir die Hauptkomponenten bzw. die Eigentextur für eine bestimmte Kamerarichtung ausgerechnet haben, können wir mit ihnen die BTF mittels angepasster Parameter rekonstruieren.

$$BTF(L, V_i, X) \approx \sum_{j=1}^c p_{V_i, j}(L) \cdot h_{V_i, j}(X), \quad j = 0, \dots, 81 \quad (75)$$

wobei  $p_{V_i, j}$  die Parameter und  $h_{V_i, j}$  die ausgerechneten Hauptkomponenten bezeichnet. Da die PCA in jede Kamerarichtung durchgeführt wird, kann für  $c$  eine kleine Zahl wie z.B. 4 oder 7 eingesetzt werden.

## 5.6 Per-cluster Matrix Factorization (PCMF)

Das Ziel, dass wir eine pixelweise Faktorisierung der ABRDFs indizieren oder die BTF-Bilder nach der Kamerarichtung gruppieren, ist um die Datenmenge der BTF mit höherer Dimension in feste Teilmengen mit niedrigerer Dimension aufzuteilen und die Teilmengen durch einen linearen Unterraum mittels PCA anzunähern. Bei der Faktorisierung der Menge der ABRDFs vergleichen wir die Abstände der ABRDFs mittels der Norm-Berechnung. Bei der Matrix-Faktorisierung nach Kamerarichtung (oder per Lichtrichtung) klassifizieren wir die Bilder entsprechend der Kamerarichtung (oder Lichtrichtung). Eine weitere Variante wird von G. Müller in [MMK03] vorgestellt. Die Idee, die Teilmengen in Abhängigkeit von den Datensätzen der BTF festzulegen, ist die sogenannte *Local PCA*-Methode.

Das Verfahren baut zuerst eine BTF-Matrix auf, deren Spalten der ABRDF entsprechen. Jeder Vektor, bzw. jede ABRDF dieser Matrix enthält  $81 \times 81$  Elemente für die Kombinationen der Licht- und Kamerarichtungen.

Im zweiten Schritt werden  $k$  Zentralvektoren  $r_i$  für  $k$  Cluster initialisiert, die zufällig aus den ABRDF-Vektoren ausgewählt werden. In jedem Cluster wird eine Menge der  $c$  Eigen-ABRDFs zur Approximation des Cluster-Raumes zugeordnet.

Im dritten Schritt werden alle ABRDF-Vektoren in den nächsten Cluster mit kleinstem Abstand zu seinem Zentralvektor aufgeteilt, gleichzeitig muß jeder Zentralvektor als Mittelwert seines Clusters neu aktualisiert werden.

Am Ende wird die Menge der Eigen-ABRDFs in jedem Cluster mittels PCA neu ausgerechnet, damit ihre  $c$  Komponenten den Cluster-Raum darstellen können. Wenn die Qualität der Rekonstruktion der BTF nicht gut ist, können wir den ersten bis letzten Schritt wiederholen, so dass der Fehler der Approximation so stark wie möglich reduziert wird. Im Vergleich zur Faktorisierung auf die ganze BTF, benötigt jede Cluster-Faktorisierung nicht erheblich mehr Speicherbedarf.

## 5.7 Die Vergleichliste der Methoden

Name	BTF-Repräsentation	Kompressionsrate	Genauigkeit	HW-Anwendung
CMF	ABRDFs	Jede ABRDF durch zwei Vektoren	hoch	sehr gut
		$r = (L+V)/L*V$	(L=81;V=81)	
PCA	2D-Textur	Matrix durch c Komponenten	niedrig	gut
		$r = 1/c$	(c=16)	
SVD	2D-Textur	Matrix durch c Eigenvektoren	niedrig	gut
		$r = 1/c$	(c=16)	
PVMF	2D-Textur	c Bilder jeder View-Richtung	mittel	gut
		$r = c/V$	(c= 4)	
PCMF	ABRDFs	c Hauptkomponenten jedes Clusters	hoch	sehr gut
		$r = c*k/N*N$	(k=32; N=256)	

Tabelle 5.1: Die Vergleichliste der Methoden

# Kapitel 6

## BTF-Synthese

### 6.1 Definition und Bedeutung der BTF-Synthese

Um die Oberfläche des Materials von 3D-Szenen realistisch zu beschreiben, werden häufig die Techniken *Image-Based-Rendering* zum Einsatz kommen. Sie ist einfach, schnell und effektiv, aber die Größe der gemessenen Bilder ist wegen immer der Begrenzung von Messgeräten limitiert. Man muß eine neue Textur ohne Beschränkung generieren, so dass die ganze Oberfläche des Objekts bedeckt werden kann. Auf der anderen Seite sind die kleinen Strukturen der Oberflächen (*mesostructure*) von den meisten Materialien in der Realität repetitiv. Wir können durch Approximation des stochastischen Prozesses eine neue Textur aus dem gemessenen Sample erzeugen, wobei die generierte Textur und die originalen Bilder möglichst ähnlich aussehen sollten. Eine Synthese der Textur ist sehr wichtig für Anwendungen in Computer-Graphik und Bildverarbeitung.

### 6.2 Algorithmen der Synthese von 2D-Texturen

Zur Synthese von 2D-Texturen können wir das Problem wie folgt formulieren: Wir bezeichnen eine Textur als visuelles Muster auf einer unbegrenzten 2D-Ebene, die eine stationäre Distribution in Mesostruktur hat. Wir nehmen an, dass ein endliches Sample, das groß genug zum Adaptieren der optischen Eigenschaften dieser Textur (z. B ein Bild) ist, als Eingabe aufgenommen wird. Unser Ziel ist, eine angepasste Methode zu finden, die unendliche Ebene durch die Muster der endlichen Textur zu bedecken.

#### 6.2.1 Tiling der Texture

Eine der einfachsten Methoden ist das Fliesen (*tiling*) des Textur-Samples auf der Ebene. Wir setzen die Sample-Textur direkt nebeneinander auf die zusynthetisierende Ebene ein. An den Lücken an den Grenzlinien müssen die Sample-Texturen eventuell abgeschnitten werden, um die Lücken auszufüllen.



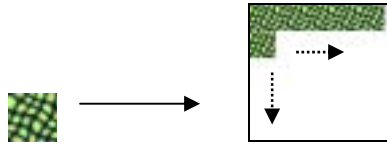


Abbildung 6.1: Texture-Tiling  
Links: die Sample-Texture.  
Rechts: die synthetisierte Texture

Aus Abbildung 33 können wir erkennen, dass die Methode der Synthese von komplexen Mustern nicht gut passt und die Verzerrung klar sichtbar ist. Die Schwäche der Methode *Tiling* liegt darin, dass die Beziehung zwischen den Mustern der Sample-Texture nicht berücksichtigt wird. Bei einem solchen Verfahren muß das Problem *Hole-Filling* gut gelöst werden.

## 6.2.2 Pixelweise Synthese der Textur

Eine oft verwendete Methode zur Synthese der 2D-Texture wird von Efros in [Efros99] vorgestellt. Die Grundidee des Verfahrens ist es, eine Textur als ein *Markov Random Field (MRF)* zu beschreiben und jedem Pixel der Textur den entsprechende Intensitätswert (bei uns RGB-Farbwert) zuzuordnen. Es wird angenommen, dass die Verteilung der Wahrscheinlichkeit des Pixelwerts für ein Pixel unabhängig von den anderen Pixeln ist, wenn die Farbwerte all seiner räumlichen Nachbarn schon bekannt sind. Der Nachbarbereich eines Pixels wird als ein rechteckiges Fenster modelliert, in dem alle Nachbarn rund um dieses Pixel stehen. Bei der Definition der Größe des Nachbarbereichs kommt es darauf an, wie stochastisch die Mesostruktur der Textur behandelt werden soll. D. h., falls die Textur relativ regelmäßige Verteilung hat, kann das Fenster ein bisschen kleiner dargestellt werden, umgekehrt ist es genauso.

Der Algorithmus von Efros wird Pixel für Pixel auf der Ebene, die durch die Sample-Textur synthetisiert werden soll, durchgeführt. Von dem Sample aus wird die synthetisierte Textur nach außen immer größer erzeugt. Zuerst wird ein einzelnes Pixel als Anfangszustand so ausgewählt, dass die Information der Sample-Texture so viel wie möglich eingefangen wird. Während der Prozess vorgeht, wird der Pixelwert jedes Pixels  $p$  folgendermaßen bestimmt: die Texels, die zu den Nachbarn rund um  $p$  im rechteckigen Fenster ähnlich sind, werden in der Sample-Textur gesucht. Dann werden die Texelwerte der Texels, die in den gefundenen Nachbarschaften innerhalb verschiedenen Fenstern auf der Sample-Textur liegen, entsprechend ihrer Gewichtung zur Berechnung des Farbwerts für  $p$  eingesetzt. Um die Geschwindigkeit des Algorithmus zu erhöhen und Speicherplatz zu sparen, kann auch eine Nachbarschaft zufällig ausgewählt werden und der Mittelwert der Texelwerte in dieser Nachbarschaft als Ausgabe ausgeliefert werden. Das Verfahren kann pixelweise neue Bilder mit beliebigen Größen aus der eingegebenen Sample-Textur erzeugen.

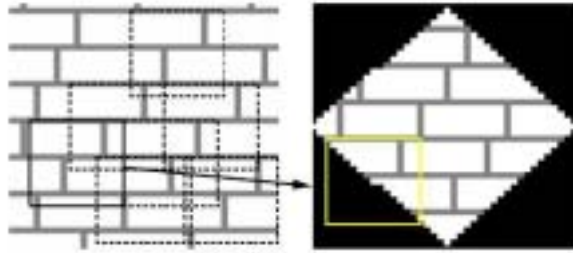


Abbildung 6.2: Überblick der Methode [Efros99]  
 Links: die Texture eines Samples wird vorgegeben  
 Rechts: die neue Bild wird Pixel für Pixel synthetisiert.

Wir werden unten durch mathematische Formeln den Algorithmus formulieren.

Wir bezeichnen die Textur des gemessenen Samples als  $I_{smp}$  und die synthetisierte

Texture aus dem Sample als  $I$ ,  $I_{real}$  als die Texture in der realen Welt ohne

Begrenzung mit  $I_{smp} \subset I_{real}$ . Wir setzen für  $W \in I$  ein Pixel der synthetisierten

Texture und für  $N_W$  alle Nachbarpixel im Textur-Fenster von  $W$  ein und definieren

$d(N_{W_1}, N_{W_2})$  als die Distanz zwischen zwei verschiedenen Nachbarschaften von

Pixeln und wählen eine kleine Zahl als die Fehlergrenze  $\delta$ .

Wie oben erwähnt betrachten wir eine Textur als eine Realisation eines gleichmäßigen *Markov Random Field*, die *Probability Density Function (PDF)* des Textur-Fensters wird durch seine kausalen, regionalen Nachbarn gekennzeichnet.

$$P(W | N_W) = P(W | I) \quad (76)$$



Abbildung 6.3: ein Pixel  $W$  und seine  
 Nachbarn im Fenster [Yakov03].

Um den Pixelwert von  $W$  zu bestimmen, suchen wir eine Menge  $\Omega(W)$  der

Nachbarschaften in der Sample-Texture, die ähnlich zu  $N_W$  ist.

$$\Omega(W) = \{N_Q \subset I_{smp} : d(N_Q, N_W) \leq \delta\} \quad (77)$$

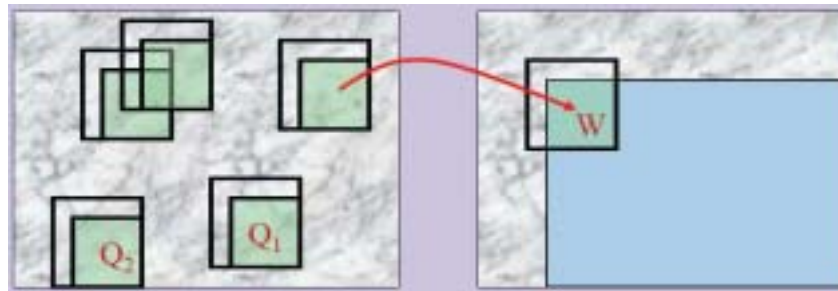


Abbildung 6.4: Es wird eine der ähnlichen Nachbarschaften ausgewählt, um den Wert von  $W$  zu bestimmen [Yakov03].

Dazu sollten wir darauf achten, dass die Nachbarn im Fenster an den Grenzlinien der synthetisierten Textur auf verschiedene Arten behandelt werden müssen. (Siehe Abbildung 6.5)

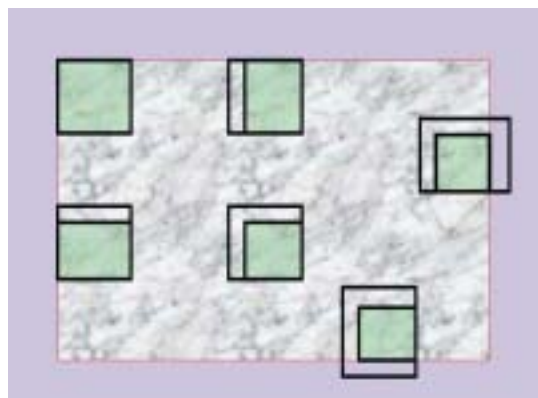


Abbildung 6.5: Die Grenzlinien des Fensters sind unterschiedlich zu behandeln [Yakov03].

Beim obigen Verfahren gehen wir immer davon aus, dass in der Sample-Textur für jedes Pixel in der synthetisierten Textur ähnliche Nachbarn auffindbar sind. Aber da wir als Eingabe nur eine endliche Textur des Samples zur Verfügung haben, kann der Fall auftreten, dass fast keine Ähnlichkeit der Nachbarschaft in der Sample-Textur zur Nachbarschaft eines Pixels gefunden wird. In einem solchen Fall können wir mittels heuristischen Methoden den Algorithmus verbessern, so dass wir ein weitere  $\Omega'(W) \approx \Omega(W)$  in der Sample-Textur feststellen. Bei der Implementierung von [Efros99] wird dieses Problem so gelöst, dass die nächste Übereinstimmung  $N_{W_{best}} = \operatorname{argmin}_W d(N_P, N_W) \subset I_{smp}$  zuerst gefunden werden soll, und dann alle Nachbarn  $N_W$  in  $\Omega'(W)$  eingefügt werden, sobald folgende Bedingung erfüllt wird:

$$d(N_w, N_p) < (1 + \varepsilon)d(N_p, N_{w_{best}}) \text{ mit } \varepsilon = 0.1. \quad (78)$$

Nun haben wir eine passende Distanz-Funktion  $d$  festzulegen, die den Grad der Ähnlichkeiten von zwei Nachbarschaften angibt. Eine Alternative wäre, dass wir einen Vektor durch die Farbwerte aller Pixel, die zusammen zum Bereich der Nachbarschaft im Fenster gehören, aufbauen können. Dann ist das Problem des Vergleichs von Nachbarschaften in das Problem des Vergleichs von Vektoren transformiert. Zur Bestimmung des Abstands von Vektoren ist die 2-Norm oder  $n$ -Norm einsetzbar, d. h. wir brauchen nur die Summe der Quadrate der komponentenweisen Differenzen zwischen den Vektoren zu berechnen. Die Faktoren in der Summe können auch mit den Gewichtungen, die die Distanzen zwischen den Mittelpunkten von Pixeln in der Nachbarschaft zu dem zu synthetisierenden Pixeln bezeichnen, multipliziert werden.

Wir geben zwei Beispiele für die pixelweise Synthese der 2D-Textur aus [Efros99].

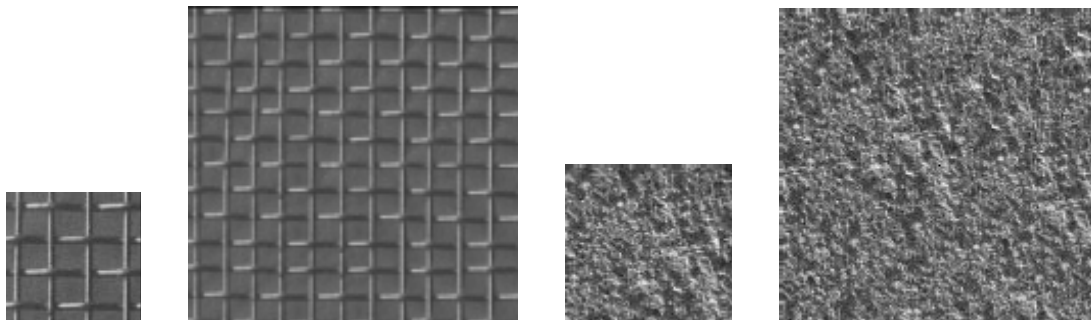


Abbildung 6.6: Synthese der 2D-Texture [Efros99]

Links: die originale Texture;

Rechts: die synthetisierte Texture

### 6.2.3 Schnelle pixelweise Synthese der Textur

Die Grundidee der pixelweisen Synthese ist es, dass eine Sample-Textur als ein *Markov Random Field* behandelt wird. Für jedes Pixel der synthetisierten Textur ist sein Pixelwert durch seine Nachbarn bestimmt, deren Pixelwerte in der Sample-Textur vorher schon zugeordnet werden. Aber beim obigen Algorithmus von Efros ist leicht zu erkennen, dass die Suche nach der ähnlichsten Nachbarschaft in der Sample-Textur sehr lange dauert und die Berechnung sehr aufwendig ist, da die Verteilung der Eintrittswahrscheinlichkeiten auf einem *Markov Random Field* gleichmäßig ist. Zur Bestimmung des Pixelwerts eines Pixels wissen wir nie, welche Nachbarschaft in der Sample-Textur als die beste Lösung ausgewählt werden sollte, bevor das letzte Pixel der Sample-Textur geprüft wird. Mit anderen Worten müssen wir für jedes Pixel der synthetisierten Textur alle Pixel auf der Sample-Textur überprüfen, um die ähnlichste Nachbarschaft dieses Pixels zu finden.

Für eine Synthese-Methode liegt der Hauptteil der Berechnungszeit in der Suche nach der ähnlichen Nachbarschaft, d. h. wenn wir die Zeit der Suche nach der Nachbarschaft der synthetisierten Pixel verkürzen können, sind wir auch in der Lage, den Algorithmus der Textur-Synthese zu beschleunigen. In [Wei00] wurde eine verwendbare Beschleunigungsmethode vorgestellt, die mit der sogenannten *Tree-structured vector quantization (TSVQ)* funktioniert. Die Idee dabei ist, dass wir die Nachbarschaften der Pixel auf der Sample-Textur als die Punkte in einem Raum mit mehreren Dimensionen bezeichnen und das Problem der Übereinstimmung der Nachbarschaften in ein Problem der Suche nach dem nächsten Punkt transformieren [Nene97].

Ein Problem der Suche nach dem nächsten Punkt kann wie folgt spezifiziert werden: wird eine Menge  $M$  von ein paar Punkten und ein beliebiger, zu suchender Punkt  $X$  in einem Raum mit mehreren Dimensionen eingegeben, ist es die Aufgabe einen Punkt  $Y$  aus dieser Menge  $M$  zu finden, so dass die Distanz zwischen  $X$  und  $Y$  minimal ist. Da die Anzahl der Punkte aus  $M$  riesig sein kann, müssen wir eine angepasste Datenstruktur aufbauen, die die verschiedenen Punkte aus  $M$  auf einer effizienten Weise abspeichert, um eine schnelle Abfrage und schnelle Besuche der Punkte aus  $M$  zu ermöglichen.

Eigentlich ist *Tree-structured vector quantization (TSVQ)* eine Methode zur Datenkompression. Sie nimmt zuerst eine Menge  $S$  von Punkten als Eingabe und berechnet dann den Mittelpunkt der Punkte aus dieser Menge. Der Mittelpunkt wird in die Wurzel eines binären Baumes abgespeichert. Alle Punkte, die unter dem Mittelpunkt liegen, werden in einem Teilbaum der Wurzel klassifiziert. Ebenso werden alle anderen Punkte, die über dem Mittelpunkt liegen, in einem anderen Teilbaum der Wurzel genommen. Innerhalb der zwei Teilbäume wird der Prozess weiter durchgeführt, bis eine Begrenzung der Tiefe getroffen wird. Zum Ende werden alle Punkte aus  $S$  in Blätter eingefügt und die Knoten speichern nur die Mittelwerte ihres Teilbaumes ab. Jedem Punkt wird eine Kodierung entsprechend seiner Position im Baum zugeordnet. Die Menge der Kodierungen wird als ein Kodebuch bezeichnet. Zur Abfrage der Punkte können wir dieselbe Methode der Besuche von Knoten im Baum nutzen oder direkt auf das Kodebuch verweisen. Die Laufzeit beträgt  $O(\log \#S)$ , schneller als  $O(\#S)$ , falls alle Punkte linear durchsucht werden sollen.

Um *TSVQ* in die Synthese-Methode [Wei00] einzusetzen, wird eine Menge  $\{N(P_i)\}$  der Nachbarschaften von allen Pixeln  $P_i$  in der Sample-Textur erzeugt. Jede

Nachbarschaft davon wird als ein Vektor bezeichnet, dessen Dimension in der Höhe der Anzahl der Pixel in der Nachbarschaft ist. Durch die Mengen von solchen Vektoren kann eine Baum-Struktur bei der Vorarbeitungsphase (*offline*) erstellt werden und alle Vektoren werden in die Blätter des Baumes entsprechend ihrer Komponente eingefügt. Während der Prozess der Synthese durchgeführt wird, wird der nächste Vektor bzw. die ähnlichste Nachbarschaft für die Nachbarschaft jedes Pixels der synthetisierten Textur im binären Baum gesucht. Gehen wir davon aus, dass  $N$  die Auflösung bzw. die Anzahl der Pixel der eingegebenen Sample-Textur bezeichnet, hat der binäre Baum eine Größe in Höhe von  $O(N)$  und eine

durchschnittliche Laufzeit der Such-Operation von  $O(\log N)$ .

Ein Nachteil, den wir beachten müssen, ist der Speicherbedarf des Baumes von Nachbarschaften. Wenn  $d$  die Anzahl der Pixel in einer Nachbarschaft bezeichnet, müssen wir mit  $O(d*N)$  den Baum abzuspeichern. Zum Glück haben Texturen überlicherweise eine sich wiederholende Struktur. Deshalb müssen wir nicht unbedingt die Nachbarschaften für alle Pixel, sondern können, um Speicher zu sparen, nur einen Teil der Pixel bei der Sample-Textur in den Baum einfügen. Aber der reduzierte Baum hat selbstverständlich eine Senkung der Qualität der synthetisierten Textur zur Folge. Abbildung 38 zeigt die synthetisierten Texturen mittels *TSVQ* verschiedener Größen.

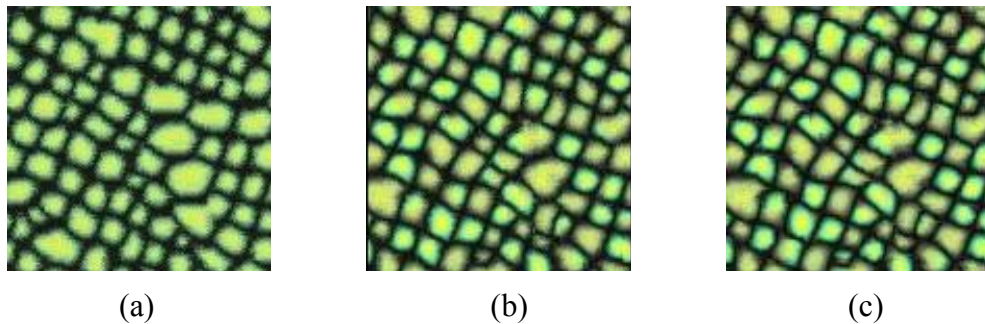


Abbildung 6.7: *TSVQ* verschiedener Baum-Größen. [Wei00]

Die originale Sample-Textur 64 x 64,

die alle synthetisierten Texture 128 x 128.

Die Größen: (a) 64, (b) 512, (c) 4096 (alle Pixel)

Zur Erhöhung der Qualität der synthetisierten Textur wird noch eine interessante Methode, die sogenannte *Multiresolution Synthesis* in [Wei00] eingebracht. Die Idee hierbei ist, da wir eine große Nachbarschaft für Texturen mit großer Skala-Struktur brauchen und eine große Nachbarschaft auch eine lange Berechnungszeit bedeutet, können wir die große Skala-Struktur durch weniger Pixel in einer Textur mit niedrigerer Auflösung darstellen [Burt83]. Bei der Implementierung werden zwei Pyramiden verschiedener Stufen in unterschiedlichen Auflösungen für die Sample-Textur und die zu synthetisierende Textur aufgebaut. Der Synthese-Prozess wird von der Unterstufe in niedrigerer Auflösung zur Oberstufe in höherer Auflösung durchgeführt. Wird ein Pixelwert einem Pixel der synthetisierten Textur auf einer Stufe der Pyramide zugeordnet, muß seine Nachbarschaft nicht nur mit der Nachbarschaft in der Sample-Texture auf derselben Stufe übereinstimmen, sondern muß sich seine entsprechende Nachbarschaft auf unterer Stufe an der Nachbarschaft in niedrigerer Auflösung der Pyramide für die Sample-Texture anpassen. Mit anderen Worten, für jedes Pixel der synthetisierten Textur werden wir versuchen, eine Nachbarschaft in der Sample-Texture zu finden, die entweder auf großer Struktur oder detailliert in kleiner Skalierung ähnlich zur Nachbarschaft dieses Pixels ist.

Wir geben zwei Beispiele mittels *Multiresolution Synthesis* in *TSVQ* [Wei00].



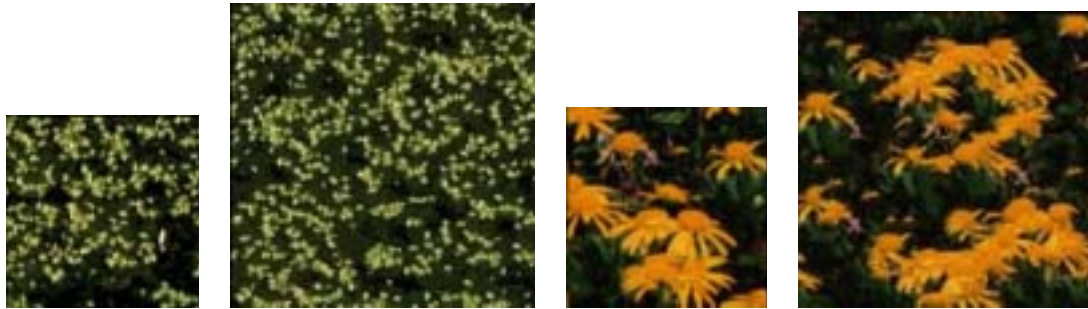


Abbildung 6.8: Die synthetisierten Texturen mittels 2-Stufen-Pyramide [Wei00]  
 Links: die originale Sample-Texture in 128 x 128,  
 Rechts: die synthetisierten Texture in 200 x 200

#### 6.2.4 Blockweise Synthese der Texture

Die Methode von [Efros99], die eine Textur beliebiger Größe Pixel für Pixel synthetisiert, ist sehr aufwendig, weil die ganze Sample-Texture für jedes Pixel der synthetisierten Texture durchsucht werden muß. In [Wei00] wurde die Methode mittels *TSVQ* beschleunigt. Aber die pixelweisen Methoden sind beide etwas unzuverlässig, da die Pixel, die innerhalb der Nachbarschaft eines zu bestimmenden Pixels liegen und als entscheidender Faktor zur Bestimmung des Pixelwerts dieses Pixels zur Verfügung stehen, sind selbst auch synthetisiert! Wenn die Pixel in den ersten Paar Zeilen oder Spalten der synthetisierten Textur nicht in Ordnung sind, dann ist die gesamte zu bestimmende Textur verzerrt. Auf der anderen Seite ist die pixelweise Synthese in manchen Fällen unnötig. Stellen wir uns vor, die Sample-Textur enthält ein spezielles Muster, das nur durch einen kleinen Teil schon erkennbar ist. Sobald der Suchprozess einige Pixel innerhalb eines Musters in der Sample-Textur gefunden hat, sind die anderen Teile dieses Musters tatsächlich auch bestimmt. D.h. viel Suchzeit wird für Pixel eingesetzt, deren Pixelwerte vorher festgelegt werden können. Durch die Beschreibung der Vorgehensweise wird aufgezeigt, dass wir vielleicht nicht nur ein Pixel, sondern in jeder Runde des Suchprozesses nur Pixel eines Stückes der Textur verwenden könnten.

Um das Problem zu lösen, wird die Methode in [Efros01] und [Liang01] verbessert. Die Grundidee, dass ein Pixel immer von seiner Nachbarschaft abhängt, bleibt unverändert. Der Unterschied ist nur, dass ein Block von Pixeln der Sample-Textur in Abhängigkeit seiner Nachbarschaft, in der einige Blöcke liegen, direkt in die zu synthetisierende Textur kopiert wird. Der überlappende Regionalbereich zwischen den Nachbarblöcken wird entweder mit Vermischung (*alpha blending*) oder mit optimaler Grenz-Beschneidung (*optimal boundary cutting*) behandelt, so dass die unmittelbare Nachbar-Blöcke voneinander nahtlos bleiben. Dieses Verfahren heißt blockweise Synthese der Texture. Es ist sehr einfach, ist aber überraschend leistungsstark. Es ist so ähnlich wie ein Puzzelspiel, wir nehmen ein Paar Stücke von einer eingegebenen Textur und versuchen sie zusammenzulegen, weswegen es auch als *Image Quilting* bezeichnet wird.

Beim blockweisen Algorithmus wird zuerst eine Menge  $S_b$  von Blöcken  $B_i$ , deren Größe vom Benutzer frei definiert werden darf (die Blöcke sollten mindestens so groß sein, dass sie die Informationen des Musters und der Struktur einer Sample-Textur enthalten können), aus der Sample-Textur aufgebaut. Es gibt insgesamt drei verschiedene Arten an Synthese-Strategien. Fangen wir mit der ersten Strategie an, die auch am einfachsten bei der Implementierung des Programmierens ist. Wir wählen die Blöcke aus der Menge  $S_b$  zufällig aus und legen sie in die passenden Positionen der zu synthetisierenden Textur ein. Abbildung 6.9 (a) zeigt deutlich, dass die Grenzlinien zwischen den unmittelbaren Blöcken klar sichtbar sind und die Regionalbereiche nicht geglättet sind. Dabei kann die Qualität der synthetisierten Textur nicht akzeptierbar sein.

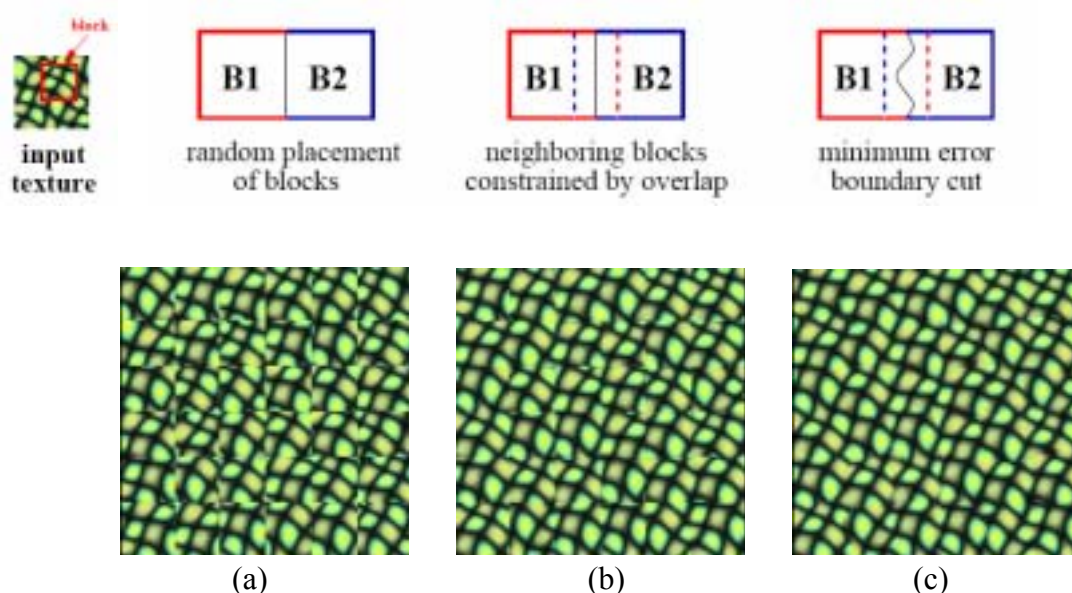


Abbildung 6.9: *Image Quilting* mittels der drei Strategien [Efros01]

- (a) Zufällige Auswahl von Blöcken
- (b) Auswahl in Abhängigkeit von Nachbarn
- (c) Grenz-Beschneidung von Blöcken

Bei der zweiten Strategie versuchen wir die Beziehung zwischen einem Block und seinen Nachbarn zu nutzen, wie wir es oben bei der pixelweisen Synthese der Textur betrachtet haben. Statt einen zufälligen Block aus der Menge  $S_b$  auszuwählen, versuchen wir einen Block aus  $S_b$  zu finden, der mit seinen Nachbarn entlang des Bereichs der Überlappung übereinstimmt. Abbildung 6.9(b) beweist die Effektivität der Strategie und zeigt die Verbesserung der Qualität der synthetisierten Textur an. Die Kanten zwischen den Blöcken sind jedoch auch hier noch deutlich erkennbar. Auf der Basis der zweiten Strategie bewerten wir die dritte Strategie als beste Lösung, den sogenannten *minimum error boundary cut*.



Das Ziel der Strategie ist den Bereich der Überlappung zwischen zwei Blöcken abzuschneiden, so dass die zwei Blöcke am Rand nahtlos anknüpfen können bzw. der überlappter Fehler am kleinsten wird. Dazu ist dynamischer Programmieren oder Dijkstra's Algorithmus verwendbar.

Der minimale Kosten-Pfad des überlappten Bereichs kann durch die folgende Schritte berechnet werden. Sollten  $B_1$  und  $B_2$  zwei Blöcke sein, die Überlappung entlang ihrer vertikalen Kanten mit Regionen  $B_1^{overlap}$  und  $B_2^{overlap}$  (Abbildung 6.9(c)). Dann

wird der Fehler der Überlappung definiert als  $e = (B_1^{overlap} - B_2^{overlap})^2$ . Um den vertikalen Pfad mit kleinstem Fehler durch das Region zu finden, muß man nur mit  $e_{i,j}(i, j = 2, \dots, N)$  durchlaufen und den gesammelten Minimum-Fehler  $E$  für alle mögliche Pfade berechnen.

$$E_{i,j} = e_{i,j} + \min(E_{i-1,j-1}, E_{i-1,j}, E_{i-1,j+1}) \quad (79)$$

Davon kann man sich vorstellen, dass beim jedem Schritt drei Wahlen vor Augen stehen: ein Schritt nach unten und ein Schritt nach links, ein Schritt nach unten und ein Schritt nach rechts sowie einfach nur ein Schritt nach unten. Der Prozess versucht immer, die beste aus den drei Wahlen zu bestimmen, so dass die Summe der Fehler minimal bleibt. Wenn man am Ende des Prozesses jeden Schritt auf dem Pfad zurückverfolgt, wird der Pfad der beste Aufschnitt des überlappten Bereichs der beiden Blöcke. Übrings funktioniert das nicht nur entlang der vertikalen Kanten, sondern auch entlang der horizontalen Kanten, falls zwei Blöcke Überlappung entlang ihrer horizontalen Kanten haben. Abbildung 6.9(c) zeigt deutlich an, dass *minimum error boundary cut* als beste Strategie im Vergleich zu den anderen zwei Strategien überragende Leistung bringt. Der Nachteil des Verfahrens ist, dynamisches Programmieren ist ein bißchen kompliziert bei Anwendungen zu implementieren und braucht auch mehr Laufzeit.

### 6.3 BTF-Synthese mittels 2D-Algorithmen

Wir bezeichnen die Synthese einer BTF als eine Texture-Synthese von BRDFs statt der Texture-Synthese der Pixelwerte. Dazu läßt sich jede BRDF einer BTF als einen Vektor höherer Dimension ansehen. Deshalb sind die Synthese-Methoden, die wir in obigen Abschnitten betrachtet haben, alle bei BTF-Synthese verwendbar, wenn wir auf einer effektiven Weise die Dimension von BRDFs reduzieren können. Eine Möglichkeit wurde von Koudelka [Koudelka03] eingebracht. Er baut eine riesige Matrix von BTF-Daten auf und führt anschließend eine SVD-Zerlegung darauf durch, um die erste  $k$  Eigenvektoren mit größten Eigenwerten zu erhalten. Die Menge von den Eigenvektoren bildet eigentlich eine Basis eines Raumes mit  $k$  Dimensionen, so dass jede BRDF durch eine lineare Kombination von den Eigenvektoren und

angemessenen Parametern repräsentiert werden kann. Damit wird die Dimension der BRDFs auf  $k$  reduziert. Der Nachteil der Methode ist, dass die Laufzeit sehr lang ist, um solch eine Matrix zu zerlegen, wenn BTF eine riesige Datenmenge enthält.

Unsere Idee ist, die BRDFs einer BTF durch Quantisation-Algorithmen wie z. B. Median-Cut-Algorithmus in einzelne Blöcke aufzuteilen. Wählen wir einen Repräsentanten aus jedem Block und tragen seinen Index auf ein Textur-Bild ein, das mit selber Auflösung wie die originale BTF-Bilder sein sollte. Dann analysieren wir das Bild der Indizes, um die statistische Muster-Information auszuziehen und die Möglichkeit zu eröffnen, dass die kleine Textur eine große Oberfläche bedecken kann. Mit anderen Worten bauen wir größere BTF nicht direkt durch die BRDFs auf, sondern ihre Indizes werden synthetisiert. Beim Rendern von Objekten braucht man natürlich eine anpassende Datenstruktur zum Speichern der BRDF-Daten, so dass Shader einfach, schnell und präzise die benötigten Daten durch den entsprechenden Index einladen kann.

## 6.4 Andere Methoden bei BTF-Synthese

In [Liu01] wurde eine Synthese-Methode der BTF eingebracht. Sein Verfahren läßt sich in drei Schritte teilen. Zum ersten wird ein annäherndes Höhe-Feld (*height field*) aus dem gemessenen Sample mittels *Shape-from-shading* [Leclerc91] entdeckt. Dann wird ein neues Höhe-Feld, der ähnliches Muster und statistische Eigenschaften wie das originale Höhe-Feld hat und auch in beliebiger Größe sein kann, mittels Synthese-Methode der 2D-Textur wie z. B. *non-parametric sampling method* [Efros99] generiert, gleichzeitig erzeugt es auch ein graues Bild in Bezug auf bestimmte Kamera- und Licht-Richtungen. Beim letzten Schritt erstellt es ein neues Bild durch Kombination von dem grauen Bild und dem indizierten BTF-Bild mit derselben Kamera- und Lichtrichtungen.

Der Durchlauf sieht wie folgt aus:

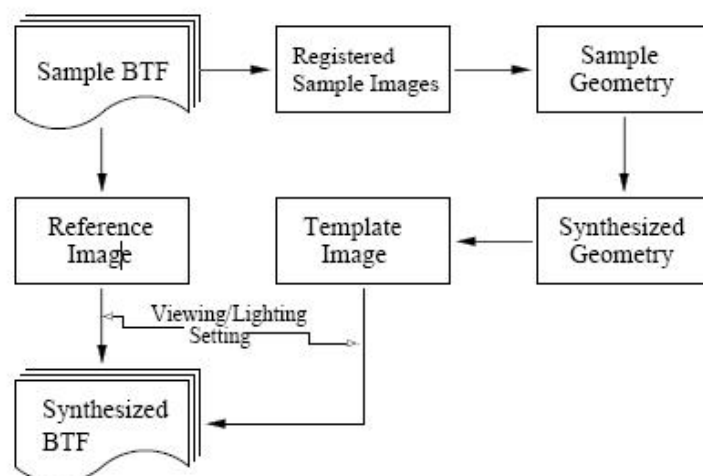


Abbildung 6.10: Der Durchlauf des Verfahrens von Liu [Liu01]

Eine weitere Alternative ist von Tong [Tong02] im Jahr 2002 vorgestellt. Sein Algorithmus basiert auf *Surface-Textons* [Leung99], die erforderliche Informationen aus dem BTF-Sample zur Synthese von BTFs extrahieren und behalten können. Zur Textur sind Textons in der Tat die Texels (*textur elements*) mit den sich versammelten charakteristischen Eigenschaften. Synthese wird später auf die Surface-Textons in Bezug auf die kleine Struktur (*mesostructure*) der Oberjetoberfläche pixelweise durchgeführt.

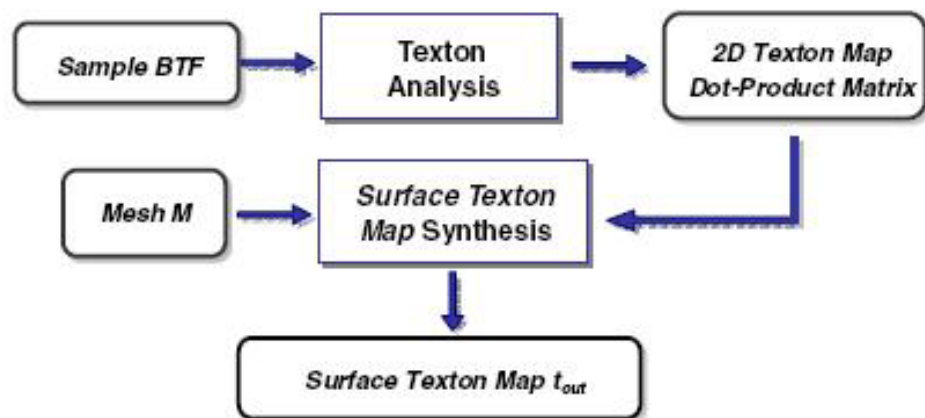


Abbildung 6.11: Der Durchlauf des Verfahrens von Tong [Tong02]

# Kapitel 7

## Implementierung von BTFs bei Anwendungen

### 7.1 Das Konzept der Implementierung

In obigen Kapiteln haben wir die Darstellungen der Kompression, sowie der Synthese von BTF betrachtet. Auf dieser Basis sind wir in der Lage, die BTF in Anwendungen einzubringen, um die komplexe Oberfläche realistischer Materialien bei der Erzeugung von 3D-Szenen zu simulieren. Dazu wählen wir zuerst einen angepassten Datenträger aus, der die Informationen von Farbwerten und der geometrischen Strukturen von BTFs beinhaltet und auf einer günstigen Weise wiederaufbauen kann. Als zweites nutzen wir wieder den Shader von RenderMan, sowie andere Anwendungen, wie z. B. OpenGL, so dass die eingelesenen Daten der BTFs die Menge von Bildern rekonstruieren können. Dabei werden wir mehr über die Arbeitsweise und Abtastung von Texturen für Shader von RenderMan diskutieren und die Implementierung der *Surface*-Sprache von RenderMan behandeln.

### 7.2 Der anpassende Datenträger von BTFs

Bei unserer Arbeit ist die Textur von Bildern (Image) eine der besten Alternativen, denn fast alle Anwendungen, die sich auf Computer-Graphik beziehen, können Texturen schnell und exakt einlesen. Sonst dienen Texturen auch ganz gut zur Hardware-Implementierung. In der Praxis haben wir mit vielen Arten von digitalen Rasterbildern zu tun, wie z. B. BMP, GIF, TIFF, PNG, JPEG,..., usw. BMP ist ein einfaches und vor allem unter Windows weit verbreitetes Dateiformat für Vollfarbender. Es benötigt aber mehr Speicherplatz, denn jedes Pixel wird als ein ganzes Byte gespeichert. JPEG ist eigentlich ein Standard, der ein Verfahren zur Kompression von Farbbildern definiert. Wir haben es ausprobiert und fanden heraus, dass die Daten, die wir in einigen JPEG-Bildern abgespeichert haben, nicht exakt mit den wieder aus den Bildern ausgelesenen Daten übereinstimmen, obwohl die kleinen Abweichungen nur eine sehr geringe Auswirkung auf die optische Wahrnehmung des menschlichen Auges haben. Aus diesen zwei Gründen läßt sich TIFF bei unserer Arbeit als beste Wahl ansehen. Es ist ein universelles und flexibles Dateiformat, das es erlaubt, Daten mit Gleitkommawerten (*float*) problemlos zu speichern, so dass es

möglich ist einen Datenträger zu finden.

Wegen des riesigen Speicherbedarfs von BTF gehen wir immer davon aus, dass Shader verschiedener Anwendungen die komprimierten Daten von BTF einladen werden. Sollten die originalen BTFs ohne Kompression eingesetzt werden, damit die Qualität der Bilder beim Shading ohne Informationsverlust ist, schlagen wir vor, Anwendungen mit den gut organisierten und sortierten Bildern in Höhe von ca. 400 MB direkt zu einer BTF zu programmieren. Man muß also nicht einmal alle Bilder in den RAM-Speicher laden, sondern zuerst eine Index-Tabelle auf die Positionen der Bilder einlesen. Nachdem man in Abhängigkeit von Kamera- und Lichtrichtungen festlegt, welche Bilder momentan benötigt werden, dürfen sie in den RAM-Speicher kommen (Es funktioniert leider nicht in Echt-Zeit!). Eine solche Index-Tabelle kann ebenso in ein TIFF-Bild gespeichert werden.

Im Kapitel 4 haben wir die Kompressionsmethode SVD per View (PVMF) vorgestellt. Die Idee ist es, dass wir zuerst die Bilder von BTF in Pixelwerte transformieren und anschließend die Pixelwerte in einer Matrix speichern, die Zeilen enthalten die Stellen der diskreten Pixel und die Spalten die R, G, B-Kanäle in allen gemessenen Einfallrichtungen der Strahlung. Nachdem wir eine SVD darauf durchgeführt und die Hauptkomponenten mit kleinen Eigenwerten vernachlässigen, brauchen wir nur die ersten  $c$  Hauptkomponenten ( $c$  kann 4, 7, 9 oder mehr pro View-Richtung sein) mit den größten Eigenwerten, die Matrizen der entsprechenden Parameter und deren Eigenwerte zu behalten. Zum Beispiel gibt es, für eine der View-Richtungen insgesamt 81 verschiedene Bilder. Dazu bauen wir eine  $(256 \times 256) \times (81 \times 3)$ -Matrix auf. Nach der SVD-Zerlegung mit 4 Hauptkomponenten haben wir noch zwei Matrizen abzuspeichern, wovon eine  $(256 \times 256) \times 4$  beträgt und die andere  $4 \times (81 \times 3)$ . Um die originale Matrix wiederaufzubauen, müssen wir nur die zwei Matrizen multiplizieren, wobei die rekonstruierte Matrix lediglich eine Approximation ist.

Auf diese zwei Matrizen sind ihre Komponenten alle positive oder negative *Float*-Zahlen. Zum Speichern von Gleitkommawerten nutzen wir den Standard IEEE754 [Parker01], in dem jeder Gleitkommawert durch eine binäre Folge von 32 Bits dargestellt werden kann.

Bei einem 32-bit TIFF-Bild mit Vollfarben gibt es die vier Kanäle Alpha, Rot, Grün und Blau pro Pixel. Jeder Farbkanal belegt ein Byte und liegt im Wertebereich von 0 bis 255. Die vier Kanäle ARGB bilden eine Folge von 32 Bits. Jeder Gleitkommawert kann durch folgende Formel genau wiedergegeben werden:

$$X = base^{exponent} \times significand \times (-1)^{sign} \quad (80)$$

wobei das erste Bit das Vorzeichen darstellt, die Bits 23 bis 31 den Exponent wiedergeben und die Bits 0 bis 22 den Significand anzeigen.

Zum Abspeichern eines Gleitkommawerts in einen Pixelwert müssen wir eine Folge von 32 Bits aus dem Vorzeichen, dem Exponent und dem Significand bilden und dann in ARGB transformieren. Beim Auslesen eines Gleitkommawerts aus dem Pixelwert geht der Prozess genau umgekehrt.

Der Quellcode für den Wiederaufbau des Gleitkommawerts aus einem Pixel in Java ist:

```
int s = ((bits >> 31) == 0) ? 1 : -1;
int e = ((bits >> 23) & 0xff);
int m = (e == 0) ?
        (bits & 0x7fffff) << 1 :
        (bits & 0x7fffff) | 0x800000;
return s * m * 2e-150
```

### 7.3 Wiederaufbau der BTF-Bilder

Der Prozess der Rekonstruktion von BTF-Bildern ist genau umgekehrt zum Prozess der Kompression von BTF-Bildern mittels SVD-Zerlegung. Im Prinzip brauchen wir nur die Pixelwerte von den  $c$  TIFF-Bildern auszulesen, in denen die Werte für die  $c$  Hauptkomponenten abgespeichert wurden, und bauen dann die Werte in einer Matrix auf. Auf die selbe Weise bauen wir eine weitere Matrix mit den zugehörigen Parametern auf. Nachdem wir die Matrix der Hauptkomponenten mit der Matrix der Parameter multiplizieren, erhalten wir eine größere Matrix für die rekonstruierten BTF-Bilder, die im Vergleich zu den originalen Bildern selbstverständlich ein wenig an Information verlieren.

Nun zeigen wir die rekonstruierten BTF-Bilder an. Alle Bilder haben die selbe Kamera-Richtung (Theta = 15, Phi = 60), zu jedem Wert von  $c$  haben wir vier verschiedene Licht-Richtungen ausgewählt: (Theta = 0, Phi = 0), (Theta = 60, Phi = 342), (Theta = 75, Phi = 105), (Theta = 75, Phi = 345).



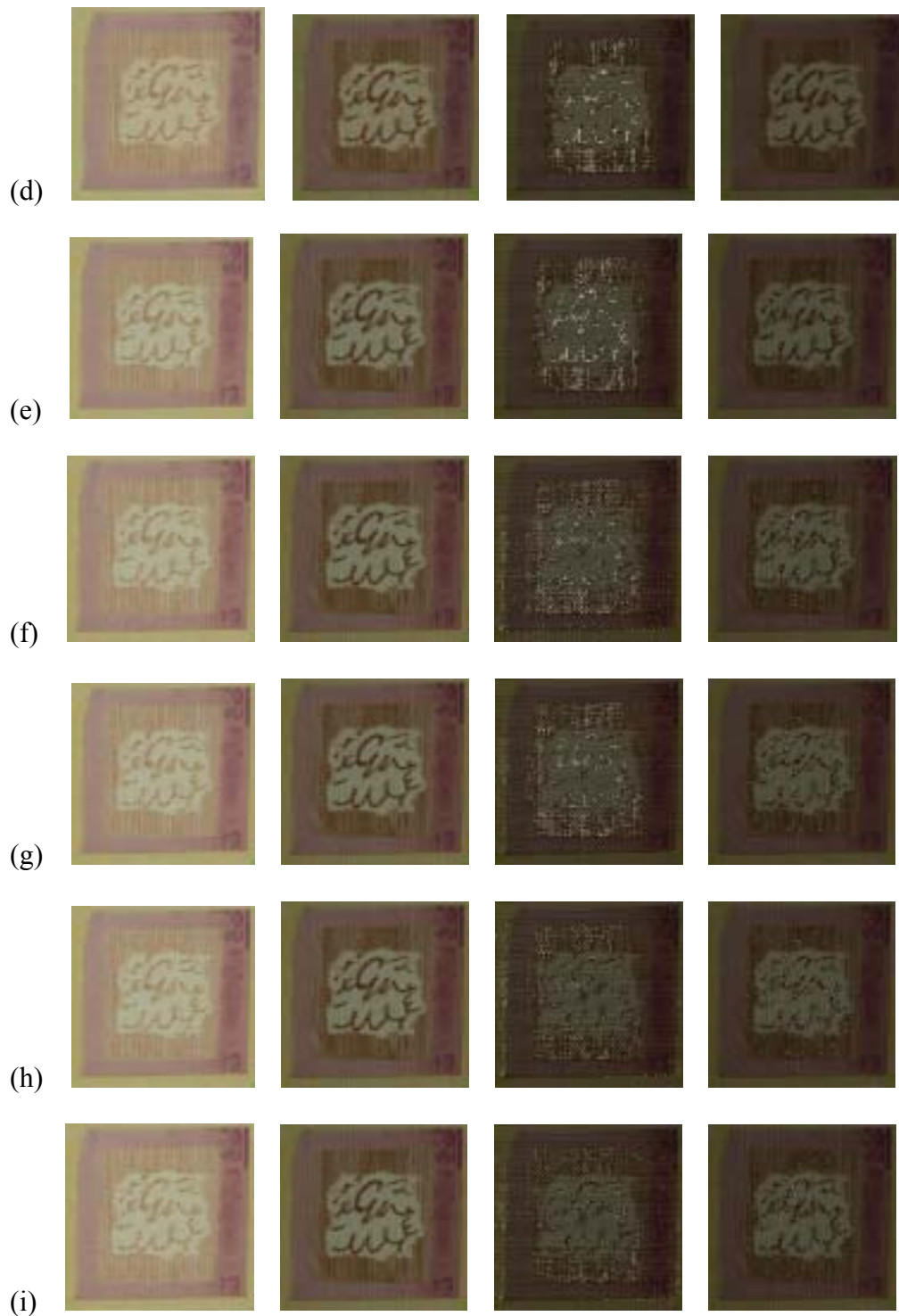


Abbildung 7.1: Die rekonstruierten BTF-Bilder

(a) die originalen Bilder

(b)  $c = 2$ ; (c)  $c = 4$ ; (d)  $c = 7$ ; (e)  $c = 9$ ;

(f)  $c = 15$ ; (g)  $c = 17$ ; (h)  $c = 22$  (i)  $c = 30$

Aus den wiederaufgebauten Bildern ist ganz deutlich zu sehen, dass je größer  $c$  eingesetzt wird, desto besser die Qualität der Bilder ist. Aber es ist ein Problem, dass ein TIFF-Bild mehr Speicherplatz als ein JPEG-Bild mit selber Auflösung braucht. So belegt zum Beispiel ein TIFF-Bild mit 256 x 256 Pixel 256KB Speicherplatz, während ein JPEG-Bild mit 256 x 256 nur ca. 60 KB belegt. Aus diesem Grund macht die SVD-Kompression keinen Sinn mehr, wenn wir für  $c$  eine größere Zahl als 17 einsetzen. Wir geben in folgender Tabelle die Kompressionsrate an:

Number Of PCA	RGB-Avg. Error(/255)	Compression Rate
2	0.0405	8.5237
4	0.0344	4.4001
7	0.0272	2.5300
9	0.0247	1.9766
15	0.0203	1.1906
17	0.0189	1.0520
22	0.0164	0.8148
30	0.0136	0.5981

Tabelle 7.1: Das Kompressionrate der SVD-Zerlegung

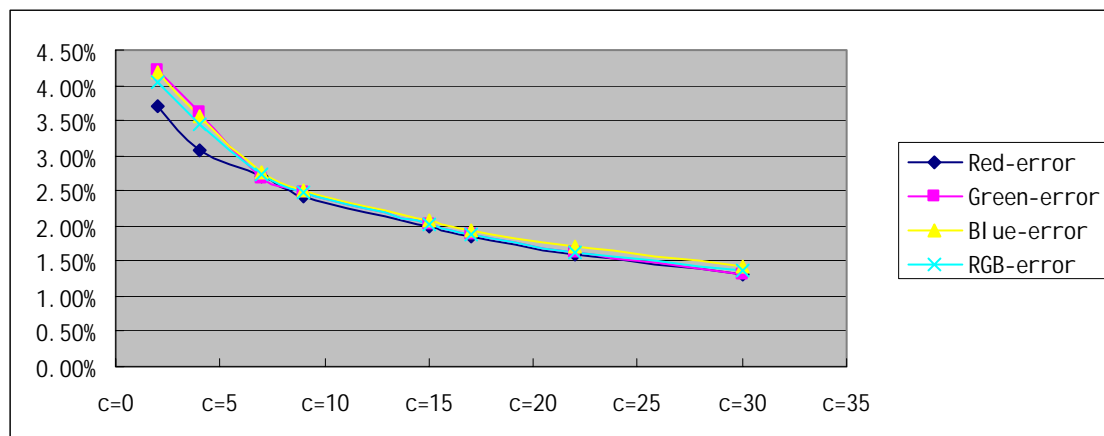


Abbildung 7.2: Die Error-Analyse der SVD-Zerlegung

## 7.4 Kompression der TIFF-Bilder mittels LZW-Algorithmen

Obwohl wir schon einen großen Faktor bei der BTF-Kompression mittels SVD-Zerlegung gewinnen können, müssen wir immer noch einen weiteren Faktor zur Speicherersparnis finden, so dass weniger TIFF-Bilder für die komprimierten



BTF-Matrizen abgespeichert werden müssen. Um die in den TIFF-Bildern gespeicherten Gleitkommawerte präzise und unverändert zu behalten und trotzdem den Speicherbedarf zu verringern, verwenden wir das einfache, schnelle und verlustfreie Kompressionsverfahren LZW [Welch84], das von Terry A. Welch im Jahr 1984 eingebracht wurde.

Das Konzept bei LZW ist es, dass wir mit einem „Wörterbuch“ mit allen möglichen Pixelwerten mit Indizes von 0 bis 255 (8 Bits) anfangen. Beim Durchsuchen eines ganzen Bildes werden die neuen „Wörter“, die noch nicht im Wörterbuch vorhanden sind und neu erscheinen, mit einer zugeordneten Kodierung zum Wörterbuch hinzugefügt.

Wir können es uns so vorstellen, dass am Anfang alle Wörter einzelnen Pixeln entsprechen und danach neue Wörter mit zwei unmittelbar nebeneinander liegenden Pixel eingefügt werden. Sobald wieder ein weiteres Wort mit zwei Pixeln auftaucht, wird nur seine Kodierung eingetragen und so komprimiert. Später werden die Wörter mit drei, vier Pixel,..., usw. eingetragen. Je länger das Wort der Kodierung wird, desto höher ist die Kompressionsrate.

LZW funktioniert schnell und ist auch einfach in allen Programmierungssprachen implementierbar. Ein weiterer Vorteil dieses Kompressionsverfahrens ist, dass das bei Kompression erzeugte Wörterbuch muß zur Dekompression nicht gesendet werden, denn die ersten 256 Wörter des Wörterbuchs sind vorher schon bekannt und das erste Pixel wird auf jeden Fall ankommen. D. h. bei der Dekodierung wird ein Wörterbuch mit den selben Wörtern erzeugt wie bei der Kodierung,

Zu unserer Implementierung sind auf den TIFF-Bildern alle Pixelwerte 32 Bit groß. Für ein TIFF-Bild mit 256 x 256 Pixels in 32 Bit reicht in den meisten Fällen ein Wörterbuch mit ca. 76000 Wörtern in 17 Bit aus. Mit anderen Worten haben wir mittels LZW die 256 x 256 x 4 ganzen Zahlen mit 8 Bit (Jedes Pixel zerlegen wir in vier Werte für A, R, G und B) in die ca. 76000 ganzen Zahlen mit 17 Bit verlustfrei komprimiert. Die Kompressionsrate beträgt ca. 1.623.

Der Algorithmus des Kompressionsverfahrens von LZW lautet:

```
set w = NIL
loop
read a character k
if wk exists in the dictionary
    w = wk
else
    output the code for w
    add wk to the dictionary
    w = k
endloop
```

Der Algorithmus des Dekompressionsverfahrens von LZW ist:

```
    read a character k
    output k
    w = k
    loop
        read a character k
        entry = dictionary entry for k
        output entry
        add w + first char of entry to the dictionary
        w = entry
    endloop
```

## 7.5 Spezifikation im Modeler von RenderMan

Wie wir vorher schon im Kapitel 2 erwähnt haben, ist das RenderMan ein Werkzeug einer Technologie zum Rendern von Computergrafiken. RenderMan erlaubt es beim Modellieren zu spezifizieren, was und wo gerendert werden muß, ohne genau festzulegen, welcher Algorithmus benutzt werden sollte. Bei den Implementierungen von 3D-Szenen nutzt man "*Modeler*" für die geometrische Definition und "*Renderer*" für die Farbwiedergabe von Oberflächen, im Prinzip sind beide voneinander getrennt programmierbar. Ein *Modeler* wird dazu benutzt, Szenen zu zeichnen und Animationen zu entwerfen. Ein *Renderer* wird so definiert, dass die entworfenen Szenen so realistisch wie möglich aussehen. Der *Modeler* spezifiziert eine Szene mit einer *RIB*-Datei und das *Shading*-Modell wird mit der *SL*-Datei in Abhängigkeit von unterschiedlichen Illuminationen in *Renderer* definiert.

Da es schon eine große Menge vom *Modeler* entworfene Modelle gibt, ist es möglich, dass wir uns jetzt nur auf das Rendern der Materialoberflächen konzentrieren können, um unsere eigene Zeichen-Methode mit BTF in *Renderer* einzusetzen.

Weil unsere BRDFs oder BTFs keine einfache mathematische Formel mit angepassten Parametern sind, müssen wir immer mit einer großen Menge von Datensätzen arbeiten und mit der Beschränkung des Speicherbedarfs von Software kämpfen. Die Daten von BTFs können leicht die von *Modeler* definierten Grenzen von Standardwerten (*default value*) [BMRT00] überschreiten, obwohl sie mittels SVD schon komprimiert wurden. Deshalb müssen wir einige Optionen neu einstellen und die entsprechend mehr Werte in die *RIB*-Dateien einsetzen, damit unsere Shader für BTF gut funktionieren.

RenderMan geht davon aus, dass beim Standardfall ein einziges Bild als Textur mit „*texturename*“ in ein Shader eingeladen wird. Wenn man weitere Texturen in Shader benutzen will, müssen die Namen der Textur-Dateien in einer *RIB*-Datei deklariert werden, damit den Texturen ihre Indizierungen vor dem Rendern zugeordnet werden können .

Die Anweisung der Deklaration sieht aus, wie folgt:

```
Declare "texturename0" "uniform string"  
Declare "texturename1" "uniform string"  
...  
Surface "wallpaper_bonnBTF"  
  
"texturename0" ["PCA-Image000.tiff"]  
"texturename1" ["PCA-Image001.tiff"]  
...
```

Die originalen Bilder einer BTF betragen ca. 400MB. Nach der SVD-Kompression können wir einen Faktor von ca. 5 gewinnen, d. h. die komprimierten Daten betragen nur noch etwa 82 MB. Dazu müssen wir „*texturememory*“ [BMRT00] in der RIB-Datei von RenderMan erhöhen.

```
Option "limits" "texturememory" [100000] #100M  
WorldBegin  
...  
WorldEnd
```

Ähnlich zu „*texturememory*“ [BMRT00] definiert „*geommemory*“ eine Begrenzung des Speicherplatzes, der zum Zeichnen der Geometrien von 3D-Szenen dient.

```
Option "limits" "geommemory" [100000] #100M  
WorldBegin  
...  
WorldEnd
```

Eine bestimmte Menge von Speicher ist dabei auch nötig, um es zu ermöglichen, dass der Interpreter der *Shading*-Sprache von RenderMan den Derivativ-Speicher (*derivmemory*) [BMRT00] korrekt berechnen kann.

```

Option "limits" "derivmemory" [100]      #100k
WorldBegin
...
WorldEnd

```

## 7.6 Surface-Shader von RenderMan

Wir stellen zuerst das einfache Beispiel „plastic“-Shading-Model von RenderMan [Ebert98] detailliert vor, um die Arbeitsweise und Struktur der *Shading*-Sprache zu erklären.

```

surface
plastic (float Ka = 1, Kd = 0.5, Ks = 0.5;
        float roughness = 0.1;
        color specularcolor = color (1, 1, 1))
{
    normal Nf = faceforward(normalize(N), I);
    normal V = normalize(-I);
    Oi = Os;
    Ci = Os * (Cs * (Ka * ambient() + Kd * diffuse(Nf)) +
              specularcolor * Ks * specular(Nf, v, roughness));
}

```

Die Parameter des Shading-Modells sind die Koeffizienten von *Ambient*, *Diffuse* und *Specular*-Reflexionskomponenten und die „*roughness*“ kontrolliert die Stufe der spiegelnden Reflexion. Farben sind im RGB-Farbraum definiert und alle drei RGB-Kanals sind zwischen [0, 1] normiert, *color(1, 1, 1)* ist Weiß.

Ein *Surface*-Shader von RenderMan bezeichnet eine Oberfläche von Geometrien als eine Menge von diskreten Punkten, die auf die Oberfläche liegen. Bei der Erzeugung von 3D-Szenen wird jeder Punkt im 3D-Koordinatensystem zuerst geprüft, ob er in Abhängigkeit der Position der Kamera sichtbar ist oder nicht. Soll der Punkt im Monitor oder im Bild ausgegeben werden, werden die Informationen des Punkts wie z.B. Position  $P$  im 3D-Raum und die Normale  $N$  an diesem Punkt an den *Surface*-Shader, der als Shading-Model aufgerufen wird, geliefert. Der *Surface*-Shader entscheidet den Farbwert oder die Intensität der reflektierten Strahlung.

Die Normale steht senkrecht zur Tangentenebene der Oberfläche am Punkt  $P$  und geht von der Oberfläche nach außen. Da eine Oberfläche zwei Seiten hat, ist es möglich, die innere Seite zu sehen. In einem solchem Fall wollen wir natürlich, dass die Normale gegen die Position der Kamera geht, nicht umgekehrt. Die von RenderMan angebotene Funktion *faceforward(vector1,vector2)* wird die Richtung von  $I$  und die Richtung der Normale  $N$  vergleichen.  $I$  ist der Vektor von der Position der Kamera zum Punkt  $P$ . Falls die zwei Vektoren  $I$  und  $N$  in der selben Richtung liegen, dann gibt *faceforward()*  $-N$  statt  $N$  aus.

Der globale Parameter  $O_s$  bezeichnet die Opazität der Oberfläche.  $O_s$  muß weniger als 1 sein, wenn die Oberfläche ein bißchen transparent ist. Der Farbwert  $C_i$ , der am Ende vom *Surface*-Shader als Reflexion ausgegeben wird, wird durch die Summe vom Ambient-Term, der Multiplikation vom diffusen Koeffizient und der Farbe der Oberfläche  $C_s$  und dem spiegelnden Term berechnet. Die Funktionen *ambient()*, *diffuse()* sowie *specular()* sind alle global und in RenderMan frei benutzbar.

Im Kapitel 2 haben wir schon ein Beispiel für die Implementierung einer gemessenen BRDF in Shader von RenderMan gegeben. Nun gehen wir einen Schritt weiter und programmieren eine BTF mit viel mehr Datensätzen in der Shader-Sprache von RenderMan. Um das Ziel zu erreichen, müssen wir uns um drei Punkte kümmern:

1. Je ein Winkelpaar  $(\theta, \phi)$  beschreibt eine Richtung im dreidimensionalen Raum.

Dieser Richtung entspricht genau ein Vektor im  $IR^3$ , der in die gleiche Richtung zeigt. Für eine gegebene Kamera- und Licht-Richtung müssen wir Einfallswinkel  $\theta_i$ , Einfallswinkel  $\phi_i$ , Ausfallswinkel  $\theta_r$  und Ausfallswinkel  $\phi_r$  im beliebigen Koordinatensystem berechnen.

2. Da all unsere komprimierten Daten von gemessenen BTFs in TIFF-Bilder gespeichert werden, ist es sehr wichtig, die Bilder in Shader von RenderMan richtig einzuladen und dabei die Pixelwerte aus den Texturen fehlerfrei abzutasten, damit die Dekompression der Daten ohne Verlust durchgeführt werden können.

3. BTF ist eine sechsdimensionale Funktion, die zusätzlichen zwei Dimensionen werden im Unterschied zu BRDF im Textur-Koordinatensystem spezifiziert. Um eine BTF mit verschiedenen Arten wie eine Textur in der Oberfläche von Geometrien abzubilden, müssen wir die Transformation vom 3D-Raum  $(x, y, z)$  nach dem Texturraum  $(s, t)$  definieren.

### 7.6.1 Die Berechnung von vier Winkeln

Die vier Winkel  $\theta_i$ ,  $\phi_i$ ,  $\theta_r$  und  $\phi_r$  beschreiben den Vektor  $V = -I$ , der die View-Richtung bezeichnet, und den gegebenen Vektor  $L$ , der die Licht-Richtung bezeichnet. Aber verschiedene Koordinatensystem ergeben auch unterschiedliche Berechnungsergebnisse. Deshalb ist es die erste Aufgabe, ein lokales Koordinatensystem an jedem Punkt der Oberfläche zu definieren. RenderMan zerlegt die Oberfläche einer Geometrie als eine Menge von kleinen Zellen. Die beiden

globalen Vektoren  $dPdu$  und  $dPdv$  lassen sich als zwei Richtungen die nach vorne verlaufen ansehen, was durch Abbildung 40 verdeutlicht wird:

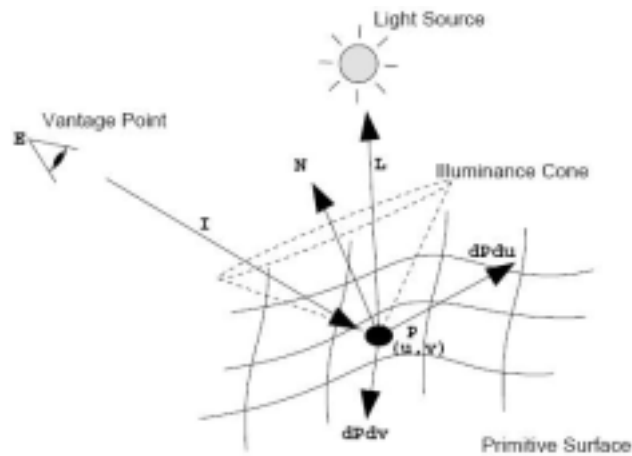


Abbildung 7.3: *Surface*-Shader-Modell aus [RI00]

An jedem Punkt, der gerendert werden sollte, können wir ein lokales Koordinatensystem aus  $N$ ,  $dPdu$  und  $dPdv$  bilden, die aufeinander senkrecht stehen. Nachdem wir  $L$  und  $V$  in dieses Koordinatensystem projiziert haben, benutzen wir folgendes Verfahren, um zwei Zenitwinkel zu berechnen:

$$\begin{aligned}\theta_i &= \cos^{-1}(N \cdot L) \\ \theta_r &= \cos^{-1}(N \cdot V)\end{aligned}\tag{81}$$

Die Berechnung des Azimutwinkles ist etwas komplizierter, da hierzu  $L$  und  $V$  in die Ebene von  $dPdu$  und  $dPdv$  projiziert werden müssen. Hierzu ziehen wir einfach den Anteil des Vektors  $L$  und  $V$  ab, der parallel zum Vektor  $N$  verläuft. Dadurch erhalten wir die zwei Projektionen  $projector\_L$  von  $L$  und  $projector\_V$  von  $V$  auf der Ebene von  $dPdu$  und  $dPdv$ .

$$\begin{aligned}projector\_L &= L - (L \cdot N) \cdot N \\ projector\_V &= V - (V \cdot N) \cdot N\end{aligned}\tag{82}$$

Mit den beiden so entstanden Vektoren können wir den Azimutwinkel berechnen.

$$\begin{aligned}\phi_i &= \cos^{-1}(dPdv \cdot projector\_L) \\ \phi_r &= \cos^{-1}(dPdv \cdot projector\_V)\end{aligned}\tag{83}$$

Der Quellcode für die Berechnung der vier Winkel in Shader-Sprache von RenderMan ist:

```

varying vector local_n = faceforward(normalize(N),I);
local_n = normalize(local_n);

varying vector local_v = normalize(dPdv);
varying vector local_u = normalize(dPdu);

varying vector Ln = normalize(L);
varying vector Vn = normalize(-I);

varying float theta_light, theta_view, phi_view, phi_light;
varying normal projector_V, projector_L;

theta_view = round(180*(acos(Vn.local_n))/PI);
theta_light = round(180*(acos(Ln.local_n))/PI);

projector_V = Vn - (Vn.local_n)*local_n;
projector_L = Ln - (Ln.local_n)*local_n;

phi_view = round(180*(acos(projector_V.local_u))/PI);
phi_light = round(180*(acos(projector_L.local_u))/PI);

```

Dazu gibt es noch einen Punkt zu bemerken: Die Funktion *arccos*() gibt einen Winkel nur im Intervall von  $[0^\circ, 180^\circ]$  aus. Wenn die Oberfläche anisotrop ist, muß genau überprüft werden, ob die Winkel  $\phi_i$  und  $\phi_r$  im Intervall von  $(180^\circ, 360^\circ)$  liegen oder nicht.

```

if (acos(projector_V.local_v)>PI/2) phi_view = 360 - phi_view;
if (acos(projector_L.local_v)>PI/2) phi_light = 360 - phi_light;

```

## 7.6.2 Repräsentation von BTFs im Shader

Gehen wir davon aus, dass die Anzahl der Hauptkomponenten bei SVD-Kompression mit 4 eingesetzt werden soll, so haben wir 4 TIFF-Bilder (sogenannte PCA-Bilder) mit einer Auflösung von 256 x 256 Pixel für 4 Hauptkomponenten und ein TIFF-Bild (ein sogenanntes Parameter-Bild) mit einer Auflösung von 4 x (81\*3)Pixel, um die zugehörigen Parameter der View-Richtung abzuspeichern, um die Matrix der Daten

von BTF später wiederaufzubauen. Da es insgesamt 81 View-Richtungen zu einer BTF-Messung gibt, müssen wir  $81 \cdot (4+1)$  TIFF-Bilder in Shader von RenderMan als Texturen einbringen. Wegen der niedrigeren Auflösung der Parameter-Bilder können wir alle 81 Parameter-Bilder in einem TIFF-Bild zusammenstellen, das  $(81 \cdot 4)$  Zeilen und  $(81 \cdot 3)$  Spalten hat, je vier Zeilen stehen für eine View-Richtung und je drei Spalten stehen für eine Licht-Richtung. Auf der anderen Seite stellen wir je vier PCA-Bilder in ein großes TIFF-Bild mit einer Auflösung von  $1024 \times 1024$  Pixel zusammen, das in der Tat alle Daten der Hauptkomponenten zu einer View-Richtung enthält. Auf dieser Weise entstehen insgesamt 82 TIFF-Bilder, die als Texturen in *Surface*-Shader eingeladen werden müssen und deren Speicherbedarf ca. 85 MB beträgt. Abbildung 41 zeigt eines der 81 PCA-Bilder und das Parameter-Bild aller View-Richtungen.

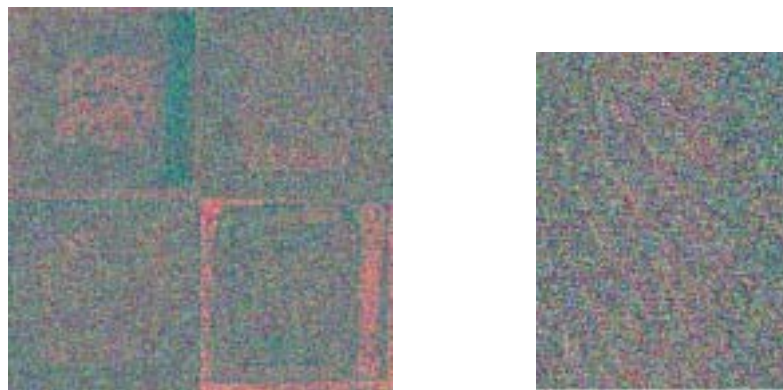


Abbildung 7.4: Zwei TIFF-Bilder für *Wallpaper*-BTF mittels SVD-Kompression (4 PCAs)

Links: das PCA-Bild der View-Richtung ( $\Theta = 0, \Phi = 0$ ).

Rechts: das Parameter-Bild aller View-Richtungen.

Wenn man **BMRT** mit der alten Version benutzt, müssen die TIFF-Bilder in Textur-Dateien (\*.tex) umgewandelt werden, damit der Shader von RenderMan auf die Pixelwerte der Texturen zugreifen kann. Die Anweisung von RenderMan [RI00] lautet:

```
MakeTexture imagename texturename parameterlist
```

Unser aktuelles **BMRT** mit Version 2.6 muß diese Anweisung nicht unbedingt durchführen, denn die TIFF-Bilder können direkt vom Shader eingelesen werden.

Die TIFF-Bilder sind zweidimensionale Texturen und werden daher indiziert durch ein Paar  $(s, t)$  der horizontalen und vertikalen Abstände von obenlinker Ecke im Intervall von  $[0, 1]$ , was die Textur-Koordinate bezeichnet. Die Anweisung der Abtastung von Texturen in Shader ist:

```
texture (filename ,q, r)
```



Wenn zwei Gleitkommawerte ( $q, r$ ) als Parameter der Textur-Koordinate eingegeben werden, wird das Render-Modell von RenderMan den nächsten Bereich von  $q$  und  $r$  auf die Shading-Einheiten (Pixel oder Micropolygon oder Sample) durchsuchen und berechnet den durchschnittlichen Wert des betroffenen Bereichs im Textur-Raum, d. h. die Ausgabe ist ein gefilterter Wert des Besuchs der Textur.

Bei der Abtastung der Textur haben wir noch ein interessantes Problem gefunden: die Auflösung der Texturen sollte möglichst 2.Potenz sein, damit die vom Shader ausgelesenen Werte mit den originalen Werten ganz genau übereinstimmen. Sonst weicht der abgetastete Pixelwert immer ein wenig ab. So hat zum Beispiel, das Pixel, das wir in einem TIFF-Bild gespeichert haben, die Farbwerte (Alpha = 255, Red = 176, Green = 27, Blue = 98), aber der Shader von RenderMan liest das Pixel als (Alpha = 253, Red = 179, Green = 25, Blue = 97) aus. Wenn man den Pixelwert nur als Farbwert behandelt oder die TIFF-Bilder nur als Texturen in Oberflächen von Materialien abbilden will, ist der Unterschied mit dem menschlichen Auge nicht so deutlich erkennbar. Aber leider bringt es ein großes Problem mit sich, wenn die TIFF-Bilder als Datenträger eingegeben werden. Um einen solchen Fall zu vermeiden, ergänzen wir unsere TIFF-Bilder immer mit Nullpixeln (A, R, G, B = 0 oder A, R, G, B = 255) bis die Höhe und die Breite der TIFF-Bilder beide 2.Potenz sind.

Die eingebaute Funktion der *Shading*-Sprache `texture()` kann zwei unterschiedliche Typen ausgeben, wobei eine Ausgabe ein Farbwert *Color* ist und eine andere eine *Float* ist.

```
c = color texture(texturename, q, r);  
float alpha = texture(texturename[3], q, r);  
float red = texture(texturename[0], q, r);  
float green = texture(texturename[1], q, r);  
float blue = texture(texturename[2], q, r);
```

Wenn es viele Texturen gibt die in Shader eingelesen werden müssen, kann man die Texturnamen in einem String-Array speichern und dann die Elemente des Arrays mit zugehörigen Indizes besuchen.

```
c = color texture(texturename[index], q, r);  
float alpha = texture((texturename[index])[3], q, r);  
float red = texture((texturename[index])[0], q, r);  
float green = texture((texturename[index])[1], q, r);  
float blue = texture((texturename[index])[2], q, r);
```

### 7.6.3 Textur-Abbildung (texture mapping)

In den vorherigen Abschnitten haben wir immer angenommen, dass die Texturen auf der  $(s, t)$ -Koordinate von  $[0, 1]$  indiziert werden und die Abtastung der Pixelwerte von

Texturen mit dem Besuch auf der Oberfläche von Geometrien fast identisch sind. Aber es passt eventuell nicht zu allen Modellen und Texturen, weil  $s$  und  $t$  möglicherweise nicht gleichmässig auf die Oberfläche distributiv sind und das Muster der Textur verzerrt aussieht. Ausserdem ist es manchmal nicht so einfach, dass die separaten und aneinander grenzende primitiven Geometrien eine kontinuierliche  $s, t$ -Abbildung am Rand haben. Eine typische Methode ein solches Problem zu bewältigen ist es, eine geeignete Funktionen der Transformation zwischen dem 3D-Raum von Geometrien und Texturraum zu definieren, damit ein Benutzer die Erscheinung der Oberflächen mit der Abbildung von Texturen selbst entscheiden kann. Eine solche Textur-Abbildung (*texture mapping*) [Foley90] [Möller99] kann kugelförmig, zylinderförmig oder planar, sowie perspektivisch sein. Wir zeigen ein Beispiel [Anthony99] in der *Shading*-Sprache von RenderMan an, dass jeden Punkt  $P$  im 3D-Raum durch eine einfache kugelförmige Abbildung in den  $(ss, tt)$ -Parameterraum projiziert.

```

vector V = normalize(vector P);
float ss = (-atan (ycomp(V)), xcomp(V) + PI) / (2*I);
float tt = 0.5 - acos(zcomp(V)) / PI;

```

Auf der anderen Seite können wir durch das Transformieren des lokalen Koordinatensystems die vielseitigen Abbildungen der Texturen auf Oberflächen von Geometrien definieren, wenn wir eine BTF als eine Textur bezeichnen, bei der jedes „Pixel“ tatsächlich eine BRDF ist.

Die *Shading*-Sprache von RenderMan erlaubt es auch, dass man durch eine 4x4-Matrix die Punkte und die Vektoren zwischen verschiedenen Koordinatensystemen transformiert. Was müssen nur die 16 Komponenten mit unterschiedlichen Werten bestimmen.

## 7.7 Die gerenderten Bilder einer Glühbirne

Wir haben hier eine RIB-Datei von RenderMan genommen, die für ein Model ein Glühbirne definiert wird. Um unsere BTF statt die *Glass*-Oberfläche auf den Kopf der Glühbirne abzubilden, haben wir das vorhandene *Surface*-Shader durch unser eigenes Shader mit verschiedenen Parametern ersetzt, so dass die verschiedenen Bilder mit Material „*Wallpaper*“ aus Bonn gerendert werden können. Da die BTF-Daten durch SVD vorher komprimiert wurden, kann man aus den gerenderten Bildern deutlich sehen, dass es ein Paar falsch gerenderte Pixel am Rand der Glühbirne gibt (Siehe Abbildung 7.5), denn bei Abtastung der Textur-Bilder sind Shader von RenderMan häufig ein bißchen anders als andere Programmiersprachen wie z. B. JAVA. Jeder Pixelwert der eingeladenen Texture wird später in eine *Float*-Zahl transformiert und dient zur Berechnung der rekonstruierten BTF-Daten, deshalb bringt die kleine Verzerrung großem Fehler beim Rendern.

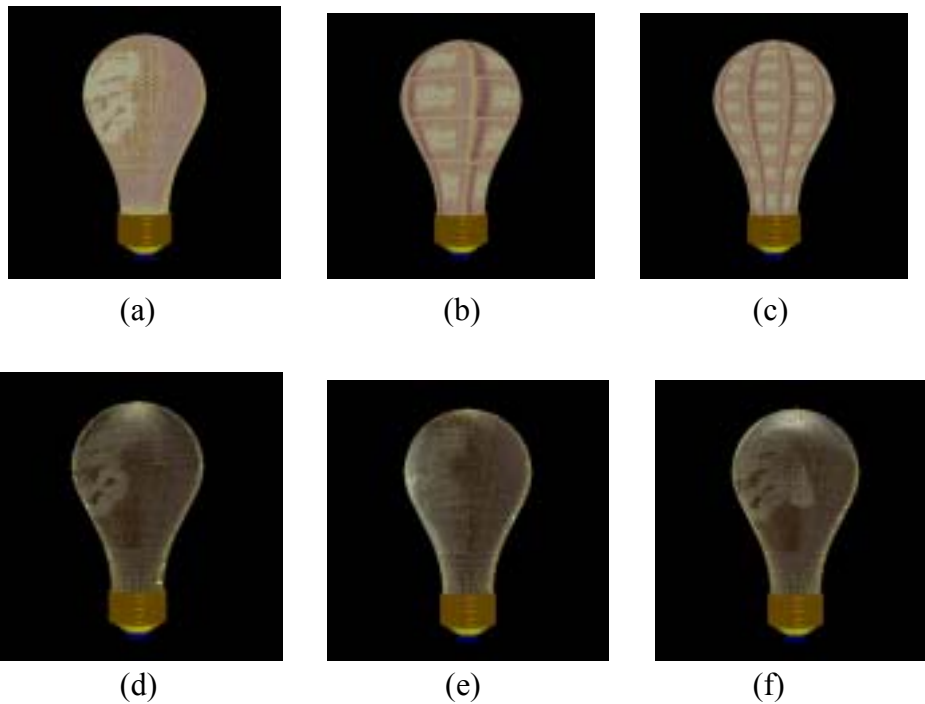


Abbildung 7.5: Die gerenderten Bilder einer Glühbirne der Abbildung

- (a) mit 2D-Textur der Skalierung = 1.0
- (b) mit 2D-Textur der Skalierung = 0.4
- (c) mit 2D-Textur der Skalierung = 0.2
- (d) mit 6D-BTF der Skalierung = 1.0
- (e) mit 6D-BTF der Rotation =  $-\pi/2$  um Oberflächennormale
- (f) mit 6D-BTF der Rotation =  $\pi*3/4$  um Oberflächennormale

## 7.8 Rendern mit den originalen BTF-Bildern

Im obigen Abschnitt haben wir die Organisation der komprimierten BTF-Daten sowie die Einladung der Daten in Shader von RenderMan vorgestellt. Mit den komprimierten Daten die Materialoberfläche von 3D-Szenen zu rendern ist wichtig und einfach gegenüber zu den Anwendungen in Echt-Zeit, aber der Verlust der Informationen bei Kompression ist immer unvermeidbar und manchmal bringt er sogar große Verzerrung bei Bildern. In Bezug auf verschiedene Ziele wollen wir eine Methode finden, mit der unser Shader direkt auf die originalen BTF-Bilder zugreifen kann. Die erste Schwierigkeit mit den originalen Bildern zu arbeiten ist der Speicherplatz. Die von der Universität Bonn aufgenommen Bilder sind alle in JPEG-Format und betragen ca. 400 MB für ein BTF-Sample. Aber Shader von RenderMan können nur Bilder in TIFF-Format einlesen. Wenn wir die originalen BTF-Bilder in TIFF-Format konvertieren, brauchen sie sogar ca. 1.7 GB! Einmaliges Durchführen von Shader mit der riesigen Datenmenge ist unmöglich, wir müssen die zu rendernde Szene aufteilen.

Stellen wir vor, dass wenn eine Materialoberfläche gerendert werden soll, ein Shader von RenderMan die Oberfläche so zeichnen wird wie eine Menge von diskreten Pixel gerendert wird, die in Richtung auf Kamera sichtbar sind. Die BTF-Bilder wurden auch in Bezug auf Kamera- und Lichtrichtung aufgenommen. Deshalb werden die BTF-Bilder mit großem Blickwinkel beim Rendern eines Punktes überhaupt nicht gebraucht, falls dieser Punkt der Oberfläche in Bezug auf Kamerarichtung kleinen Blickwinkel hat, umgekehrt ist der Fall auch genauso. D. h. zu diesem Zeitpunkt muß Shader die BTF-Bilder mit großem Blickwinkel gar nicht einladen, obwohl sie zu späterem Zeitpunkt noch benötigt werden. Aus diesem Grund können wir unser Shader in eine Menge von Shader zerlegen, die jeweils für einen Bereich der Kamerarichtung verantwortlich sein sollten und nur eine Teilmenge von BTF-Bildern einladen müssen. Bei unserer Implementierung teilen wir alle Pixel in Abhängigkeit von Deklinationswinkel der Kamerarichtung in drei Gruppen auf: von 0 bis 45 Grad, von 45 bis 60 Grad und von 60 bis 90 Grad. Die originalen BTF-Bilder werden entsprechend in drei Mengen (ca. 600MB) verteilt und dienen zum Rendern von den drei Shadern, die nur die innerhalb ihres Gebietes liegenden Pixel zeichnen und die anderen Pixel außerhalb ihres Gebietes mit Nullwert setzen. Zum Schluß fassen wir die drei gerenderten Bilder in ein Bild zusammen.

Ein Teil der Quellcode von den drei Shader sieht so aus wie unten:

```

if ((0<=theta_view)&&(theta_view<45)) { // if viewer can see in [0, 45).
    rendern(pixel);
}
else
    pixle = color(0,0,0);

```

```

if ((45<=theta_view)&&(theta_view<60)) { // if viewer can see in [45, 60).
    rendern(pixel);
}
else
    pixle = color(0,0,0);

```

```

if ((60<=theta_view)&&(theta_view<90)) { // if viewer can see in [60, 90).
    rendern(pixel);
}
else
    pixle = color(0,0,0);

```

Zeigen wir nun die gerenderten Bilder von RIB-Datei einer Teekanne mit Material-*Wallpaper* und Material-*Impalla*, wobei wir die Dreieck-Interpolation eingesetzt haben, die der Aussage der Universität Bonn nach und durch unsere eigene Erfahrungen besser als Viereck-Interpolation ist.

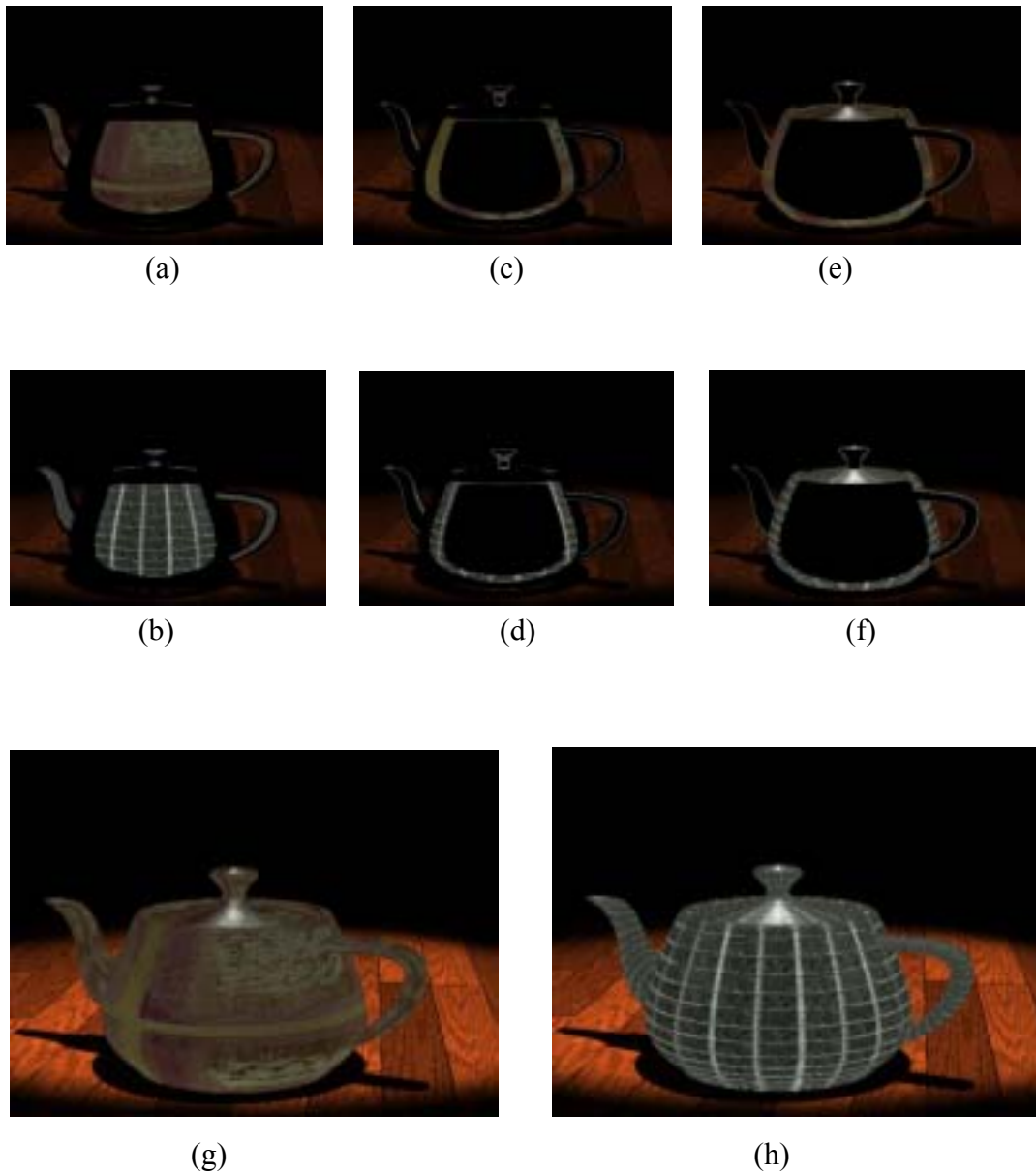


Abbildung 7.6: Die von RenderMan gerenderten Bilder von BTF-*Wallpaper* und BTF-*Impalla*  
 (a), (b): *Theta\_View* in  $[0, 45)$  Grad  
 (c), (d): *Theta\_View* in  $[45, 60)$  Grad  
 (e), (f): *Theta\_View* in  $[60, 90)$  Grad  
 (g), (h): vollständige Bilder

# Kapitel 8

## Generierung von BTFs

### 8.1 Bedeutung der Generierung von BTFs

Die Akquisition BTFs auf Grundlage realer Material-Muster ist sehr aufwendig und bedarf spezieller und technisch komplexer Aufnahmegeräte. Das ist auch der Grund, warum es nicht so viele gute Datenbanken von BTFs im Internet gibt wie in der Universität Bonn. Auf anderer Seite lassen sich einmal erstellte Modelle nicht parametrisieren oder variieren.

Sollten wir eine BTF als eine Texture von pixelweisen ABRDFs bezeichnen, wollen wir eine neue BTF aus vorhandenen BRDFs und BTFs so generieren wie die Muster von 2D-Texturen aus Farbwerten erzeugt werden können. Unterschied der beiden Verfahren ist nur, dass sich eine BRDF als ein Pixel ansehen läßt, obwohl sie viel mehr Dimensionen hat als ein dreidimensionaler (RGB) oder vierdimensionaler (ARGB) Pixelwert. Wenn wir wissen, welche BRDFs in einer BTF am häufigsten vorkommen, können wir ja einfach die BRDFs ihrer Häufigkeit nach sortieren und die BRDFs, die am meisten vorkommen, setzen. Das ist machbar, der Qualitätsverlust ist nicht besonders groß.

Auf dieser Weise der Quantisierung sind wir in der Lage, die Anzahl von BRDFs einer BTF sowie den Speicherbedarf zu verringern und die Verteilung von BRDFs zu ermitteln, damit unser Ziel, die Generierung von neuen BTFs, ermöglicht werden kann.

### 8.2 Median-Cut-Algorithmus

Median-Cut-Algorithmus wurde von Heckbert in 1980 [Heckbert82] eingebracht, dieser dient zur Quantisierung von Farbwerten und Kompression von Farbbildern.

Die Idee ist, zuerst jede Pixelfarbe im Quellbild als Punkt im RGB-Kubus aufzufassen und danach die Pixelfarben des Quellbildes Punktwolke im RGB-Kubus zu bilden. Die Aufgabe ist, die Punktwolke wiederholt in Quader zu teilen, bis eine Grenze der Anzahl der kleinen Quadern erreicht wird. Bei der Bestimmung dieser Quader sollen die Häufigkeit und die Entfernung der Farben voneinander, also die Ähnlichkeit der Farben, eine Rolle spielen. Damit möchte man auch eine

gleichmäßige Verteilung der Farben erreichen können. Am Ende wird ein Repräsentant aus jedem der Quader ausgewählt, auf den alle Pixelfarben des Quaders abgebildet werden. Dazu wird der durchschnittliche Pixelwert (*centroid*) aller Pixelfarben aus dem Quader oft als eine gute Wahl benutzt. Der Algorithmus sieht so aus wie unten:

```

Color_quantization(Image, n){
  For each pixel in Image with color C, map C in RGB space;
  B = {RGB space};
  While (n-- > 0) {
    L = Heaviest (B);
    Split L into L1 and L2;
    Remove L from B, and add L1 and L2 instead;
  }
  For all boxes in B do
    assign a representative (color centroid);
  For each pixel in Image do
    map to one of the representatives;
}

```

Da bei jeder Iteration des Prozesses ein Quader ausgewählt und danach in zwei Quader mit der selben Anzahl der Punkte geteilt wird, können wir den Durchlauf durch einen binären Baum (*no balance*) abbilden. Die Wurzel des Baums enthält alle Pixelfarben des Quellbildes und seine zwei Kinder teilen diese Pixelfarben. Die Repräsentanten der Blätter bezeichnen die neu abgebildete Farbpalette.

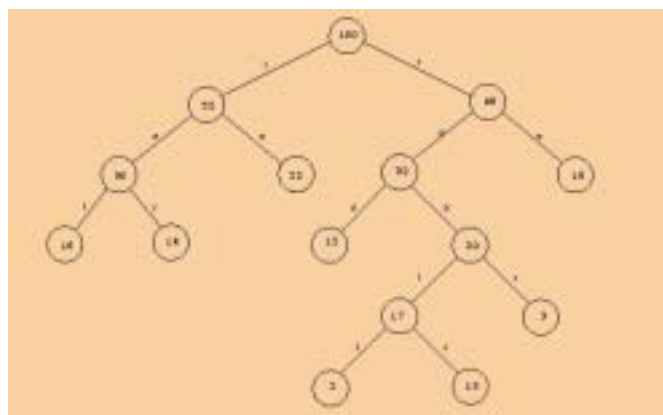


Abbildung 8.1: Ein Beispiel des Median-Cut-Baumes [Osnabrueck]

Im Prinzip funktioniert Median-Cut-Algorithmen genauso bei Verwendung von BTF wie von 2D-Rasterbildern, nur bringt BTF wegen ihrer riesigen und komplexen Datenmengen ein großes Problem bei der Laufzeit. Wir wissen, die Auflösung einer BTF entspricht der Anzahl von BRDFs. Bei BTF-Datenquelle der Universität Bonn ist jedes BTF-Bild in Auflösung mit 256 x 256 Pixels, die auf 256 x 256 BRDFs

verweisen. Bei der Implementierung bilden wir die BRDFs in einen Raum mit  $81 \times 81 \times 3$  Dimensionen ab, da BTF in  $81 \times 81$  verschiedene Kombination der Kamera- und Licht-Richtungen gemessen wurden und jeder Pixelwert in RGB-Raum zerlegt wird. Bei der Bestimmung der Verteilung von Quadern müssen wir in jede Iteration die Dimension mit größter Distanz aus  $81 \times 81 \times 3$  suchen, um zu entscheiden, in Bezug auf welche Dimension die BRDFs sortiert werden und weiter in zwei Quader mit selber Anzahl verteilt werden sollten. Bei meinem Entwicklungsrechner mit CPU Pentium IV (1.5GHZ) und 512 RAM kostet jede Durchsuchung der längsten Kante in Quader ungefähr 5 Minuten. Man kann sich vorstellen, dass wenn wir die BRDFs in 256 (8 Bits) Quader abbilden wollen, dieser Prozess ungefähr  $256 \times 5$  Minuten  $\approx 22$  Stunden dauert. Eine Möglichkeit zur Verkürzung der Laufzeit wäre, in jeder Iteration des Prozesses statt des Quaders mit längster Kante ein Quader höchster Stufe in einem so entstandenen binären Baum (Siehe Abbildung 8.1) zu nehmen, damit der Baum ausgeglichen (*balance*) ist und die Tiefe nicht zu hoch wird. Beim Programmieren muß man also nicht die größten Distanzen von allen Quader vergleichen, sondern ein Quader höchster Stufe, das noch keine Kinder hat, nehmen. Damit wird die Zeit der Durchsuchung gespart. Solchen Methoden beschleunigen den Prozess, bringen aber selbstverständlich die Senkung der Qualität der Quantisation.

Nachdem wir die BRDFs einer BTF in eine Menge von Quadern quantisiert haben, wählen wir ebenso einen Repräsentanten aus jedem Quader aus und bilden alle BRDFs innerhalb dieses Quaders auf den Repräsentanten ab. Wir berechnen die durchschnittliche BRDF mit jeder durchschnittlichen Komponente:

$$\begin{aligned}
 & BRDF^{Quader\_i}_0(Pixelwert^0_0, Pixelwert^0_1, \dots, Pixelwert^0_{81 \times 81}) \\
 & \quad \vdots \\
 & BRDF^{Quader\_i}_n(Pixelwert^n_0, Pixelwert^n_1, \dots, Pixelwert^n_{81 \times 81}) \quad (84) \\
 & \Downarrow \\
 & BRDF^{Quader\_i}_{centroid} \left( \frac{1}{n} \sum_{k=1}^n Pixelwert^k_0, \frac{1}{n} \sum_{k=1}^n Pixelwert^k_1, \dots, \frac{1}{n} \sum_{k=1}^n Pixelwert^k_{81 \times 81} \right)
 \end{aligned}$$

Einen Punkt müssen wir hier besonders erwähnen, die Anzahl der Quader bei der Quantisation sollte stark von dem BTF-Material abhängen. Für ein normales 2D-Farbbild gibt es nur drei Dimensionen, wenn wir die Farbpunkte in Bezug auf eine der drei Dimensionen aufteilen, werden die Farbpunkte mit Ähnlichkeit bei den anderen zwei Dimensionen im selben Quader liegen. Aber eine BRDF hat viel mehr Dimension als ein Pixel, was natürlich hoch kompliziert wird. Stellen wir vor, ein Material soll mit starker Veränderung in verschiedene Richtungen auf Kamera oder Lichtquelle aussehen. D. h. die BTF, die aus solchem Material akquiriert wird, hat eine Menge von Textur-Bildern mit großen Unterschieden voneinander. Falls wir die BTF als eine Menge von BRDFs bezeichnen, haben die Pixelwerte jeder BRDF ebenfalls große Unterschiede dazu. Obwohl wir durch Quantisation die BRDFs mit größter Differenz in verschiedene Quader aufteilen können, kann man auch nicht garantieren, dass die BRDFs innerhalb desselben Quaders ähnlich sind. Die einzige Lösung dazu ist die Anzahl der Quader zu erhöhen, was aber mehr Kosten verursacht.



### 8.3 Indexbilder und BRDF-Block

Um die Aufteilung von BRDFs einer BTF auf einen Datenträger zu speichern, bringen wir den Begriff „Indexbild“ von BTFs ein, damit die aufgeteilten BRDFs bei Anwendungen eingesetzt werden können.

Zu einer BTF-Textur bezeichnet Indexbild die Aufteilung der BRDFs, die als Repräsentanten aus jedem Quader gewählt werden. Jeder Pixelwert des Indexbildes verweist auf eine BRDF, die ebenso in einem sogenannten „BRDF-Block“-Bild abgespeichert wird. Auf diese Weise brauchen die Shader von Anwendungen nur zwei Textur-Bilder einzuladen, was bei unterschiedlichen Programmiersprachen sehr leicht implementierbar ist.

Wir nutzen immer die BTF-Datenquelle von Universität Bonn wie oben. D. h. unsere Index-Bilder haben dieselbe Auflösung von 256 x 256 Pixels wie die gemessenen Textur-Bilder der Materialien. Am Anfang erzeugen wir ein 2D-Rasterbild (Anfangsbild), dessen jedes Pixel einen BRDF-Index bezeichnet (Siehe die Abbildung 53). Beim Lauf des Programms kann der Zwischenzustand auch in Bilder mit selber Auflösung gespeichert werden.

*Pixel(0), Pixel(1), ....., Pixel(255)*  
*Pixel(256), Pixel(257), ....., Pixel(511)*  
 ...  
 ...  
*Pixel(256\*255), Pixel(256\*255+1)....., Pixel(256\*256-1)*

Nachdem die Quantisation zu den BRDFs durchgeführt wurden, wird der Index jeder BRDF durch den Index eines Quader, zu dem die BRDF gehört, ersetzt. Es sieht z. B. wie unten aus.

(Siehe die Abbildung 51)

*Quader(4), Quader(0),.....,Quader(m),....., Quader(k)*  
*Quader(9), Quader(7),....., Quader(i)*  
 ...  
 ...  
*Quader(m), Quader(i),....., Quader(7)*

wobei  $0 \leq m, k, i \leq n$  und  $n$  die Anzahl der Quader ist.

Das BRDFs-Block-Bild speichert die Pixelwerte von allen BRDFs, die der Reihenfolge der Quader nach sortiert werden:  
 (Siehe die Abbildung 52)

$$\begin{array}{l}
 BRDF_{\text{Quader}_0}(Pixelwert^0_0, Pixelwert^0_1, \dots, Pixelwert^0_{81 \cdot 81}) \\
 BRDF_{\text{Quader}_1}(Pixelwert^1_0, Pixelwert^1_1, \dots, Pixelwert^1_{81 \cdot 81}) \\
 \vdots \\
 BRDF_{\text{Quader}_{n-1}}(Pixelwert^{n-1}_0, Pixelwert^{n-1}_1, \dots, Pixelwert^{n-1}_{81 \cdot 81})
 \end{array}$$

Wir zeigen die Indexbilder und BRDF-Block-Bilder sowie das Anfangsbild unten:

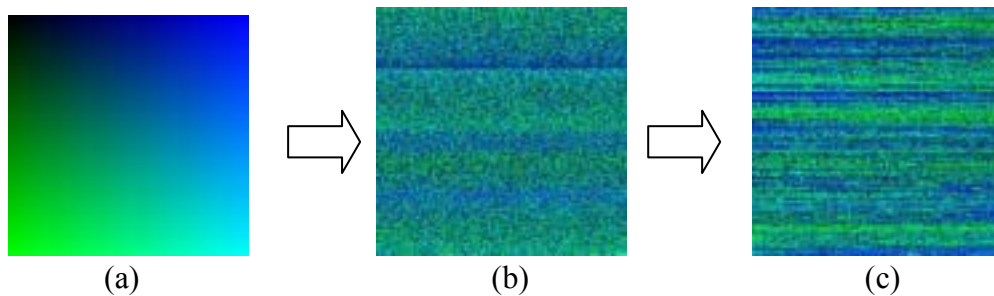


Abbildung 8.2: Der Durchlauf der Quantisation der BRDFs-Indizes  
 (a) Das Anfangsbild  
 (b) Quantisation mit 8 Quadern  
 (c) Quantisation mit 4096 Quadern

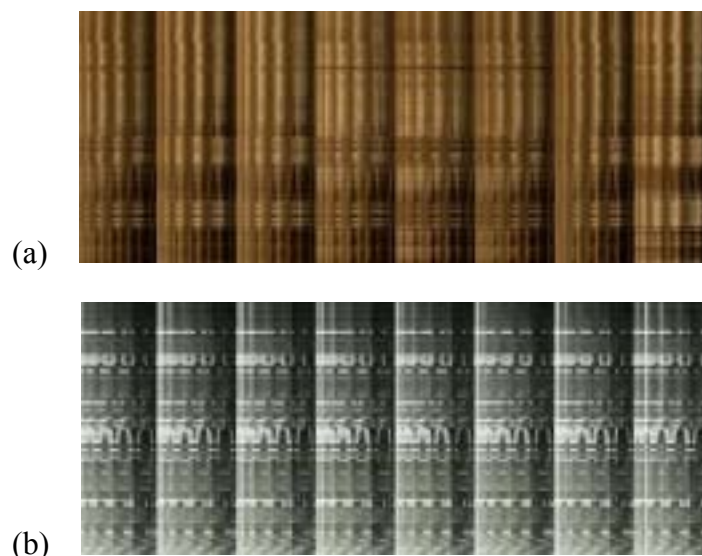


Abbildung 8.3: Die BRDFs-Block-Bilder  
 (a) Material-Coduroy  
 (b) Material-Impalla

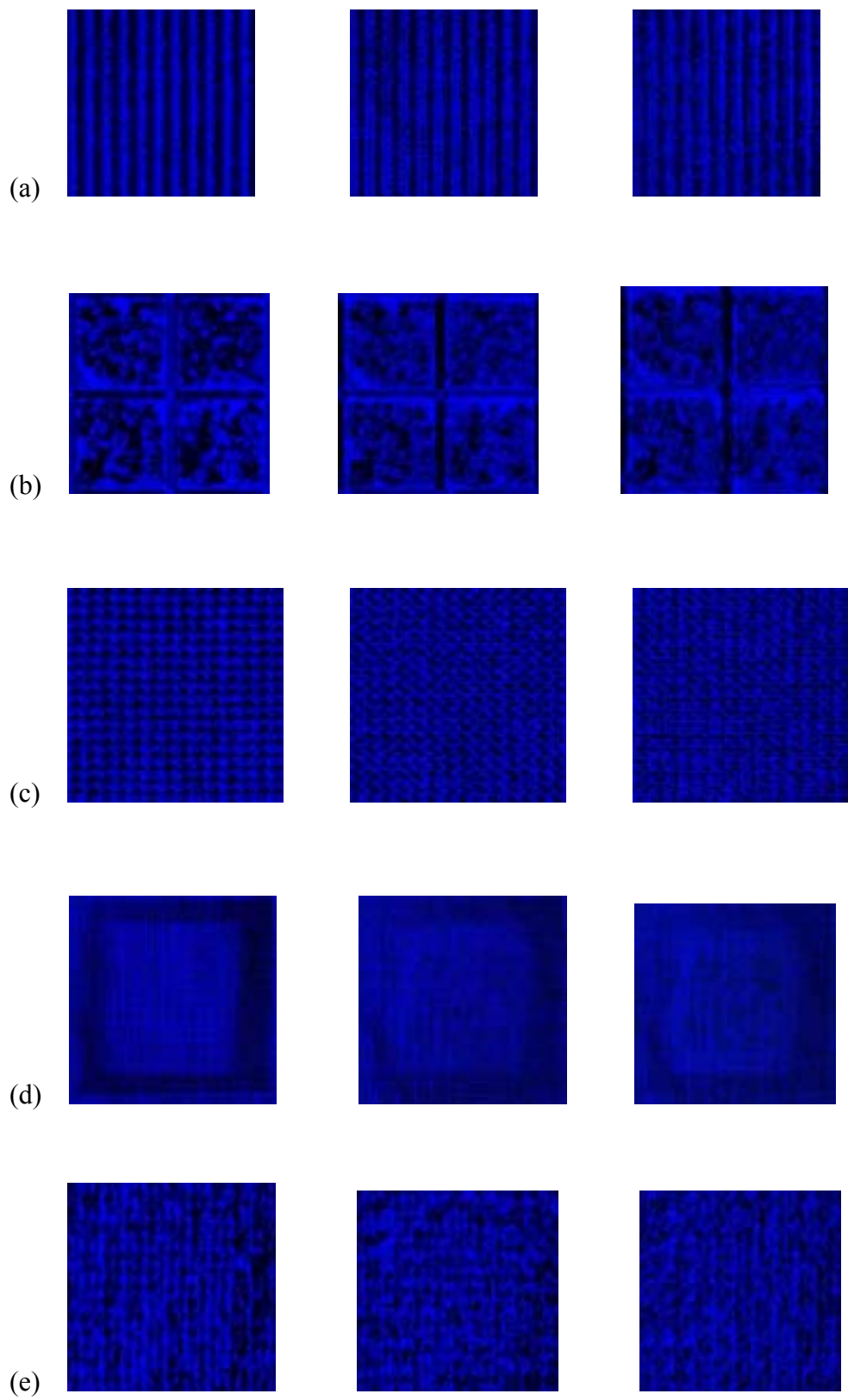


Abbildung 8.4: Die Indexbilder von (a) *Corduroy*, (b) *Impalla*, (c) *Proposte* (d) *Wallpaper*, (e) *Wool* mit 256, 512 und 1024 Quadern.

Um den Einsatz des Median-Cut-Algorithmus zu beurteilen, zeigen wir zuerst die rekonstruierten BTF-Bilder aus den Index- und BRDF-Block-Bildern und die originalen BTF-Bilder zusammen, jedes der Textur-Bilder ist betrachtet unter Kamerarichtung  $(\theta = 0^\circ, \phi = 0^\circ)$  und Lichtrichtung  $(\theta = 0^\circ, \phi = 0^\circ)$ .

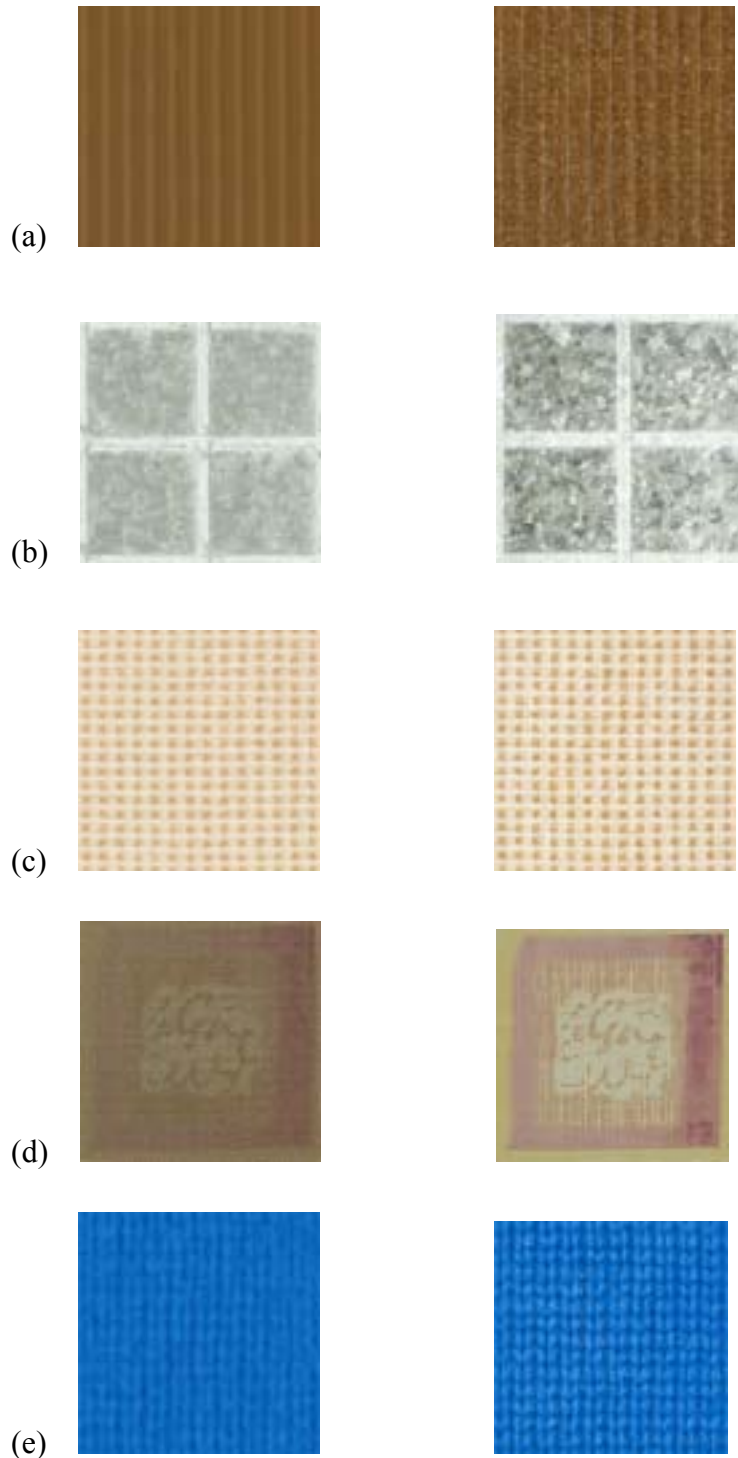


Abbildung 8.5: die rekonstruierten BTF-Bilder  
 Links: die Bilder aus Median-Cut (256 Quader)  
 Rechts: die originalen BTF-Bilder

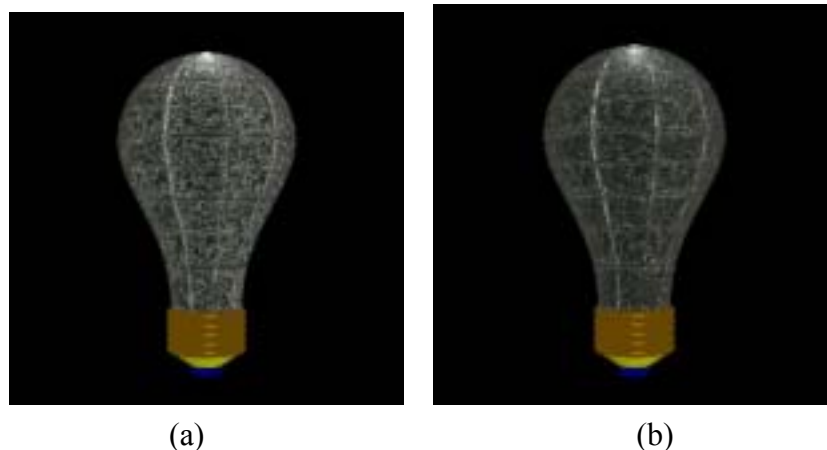
Um die Qualität der rekonstruierten Bilder mittels Median-Cut-Algorithmus mit den Bildern mittels SVD-Kompression zu vergleichen und den Error zu analysieren, stellen wir wieder eine Tabelle wie folgt auf:

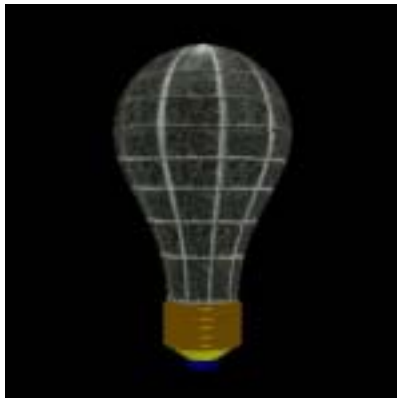
Level of Median-Cut	RGB-Avg. Error(/255)	Max-Error(/255)	Compression Rate
256 (balanced)	0.097	0.162	192.308
256 (no balance)	0.075	0.131	161.031
400 (no balance)	0.062	0.116	80.000

Tabelle 8.1: Die Kompressionsrate des Median-Cut-Algorithmus

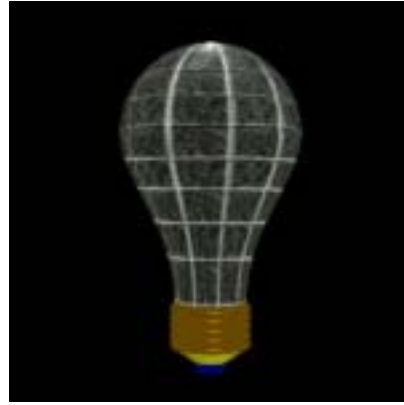
## 8.4 Im Vergleich zu vorherigen Rendern-Methoden

Bis jetzt können wir schon die Methoden zum Rendern von BTFs in zwei Gruppen klassifizieren. Der Idealfall, der die Bilder bester Qualität erzeugt, ist selbstverständlich direkt auf die originalen BTF-Bilder zu programmieren, um den Verlust der BTF-Daten zu verhindern. Bei der zweiten Gruppe ist zuerst die riesigen BTF-Datenmenge mittels verschiedener Kompression- oder Zerlegungsverfahren zu komprimieren, die entweder auf die BRDFs (wenn wir eine BTF als eine Menge von BRDFs bezeichnen) oder direkt auf die BTF durchgeführt werden, und dann auf die komprimierten Daten zu programmieren. Nun können wir eine dritte Gruppe einbringen, die BRDFs einer BTF durch Quantisation wie z. B. Median-Cut-Algorithmus zu quantisieren, damit die Anzahl von BRDFs einer BTF verringert werden kann und gleichzeitig der benötigte Speicher auch eingespart wird. Unten zeigen wir zuerst die gerenderten Bilder aus den drei verschiedenen Gruppen an.





(c)



(d)

Abbildung 8.6: Die gerenderten Bilder von Material-*Impalla* von  
(a) Median-Cut-Algorithmus (*balance*, 256 Quader)  
(b) Median-Cut-Algorithmus (*no balance*, 256 Quader)  
(c) SVD-Kompression (9 PCAs per View)  
(d) den originalen BTF-Bildern



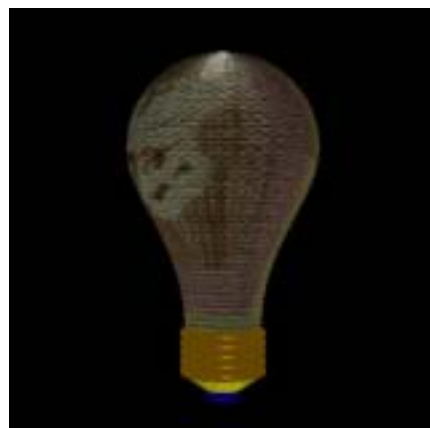
(a)



(b)



(c)



(d)

Abbildung 8.7: Die gerenderten Bilder von Material-*Wallpapera* von  
 (a) Median-Cut-Algorithmus (*balance*, 256 Quader)  
 (b) Median-Cut-Algorithmus (*no balance*, 256 Quader)  
 (c) SVD-Kompression (9 PCAs per View)  
 (d) den originalen BTF-Bildern

Aus den gezeigten Bildern können wir deutlich sehen, dass nicht nur die rekonstruierten BTF-Bilder relative große Verzerrung im Vergleich zu den originalen Bildern haben, sondern auch die daraus gerenderten Bilder sehen nicht so gut aus wie die anderen zwei Methoden. Aber Median-Cut-Algorithmus bringt trotzdem Vorteile, die die anderen zwei Methoden nicht haben. Der erste Vorteil ist Speicherplatzeinsparung. Für eine BTF in Höhe von ca. 400MB kann nur durch zwei Bilder in Höhe von ca. 2MB ( 256 Quader ) oder ca.10MB (1024 Quader) repräsentiert werden. Dazu hat Kompression mittels SVD nur ein Kompressionsrate gleich  $4/81$ , wenn wir die niedrigste Anzahl von Hauptkomponenten mit 4 einsetzen. Außerdem enthalten die durch SVD-Zerlegung komprimierten Daten noch negative Werte, die bei der Einladung von Texturen oder Hardware-Implementierung Probleme bringen können. Die Laufzeit der Implementierung mittels Median-Cut-Algorithmus ist auch viel kürzer als der Implementierung mittels SVD, denn die von Median-Cut-Algorithmus erzeugten Bilder enthalten Pixelwerte, die direkt von Shader beim Rendern benutzt werden können. Aber die von SVD erzeugten Bilder sind eigentlich Datenträger, deren Pixel *Float*-Zahlen speichern. Die Transformation von den Pixelwerten nach *Float*-Zahlen kostet viel Zeit, und die Berechnung des Wiederaufbaus der originalen Daten von den Hauptkomponenten ist auch sehr aufwendig. Ein weiterer Vorteil, der allein von Median-Cut-Algorithmus eingebracht wird, stellen wir im nächsten Abschnitt vor.

## 8.5 Einsatz neuer BTFs beim Rendern

### 8.5.1 Erzeugung von BTFs verschiedener Farben

Im obigen Abschnitt haben wir mittels Median-Cut-Algorithmus die Information über das Muster einer BTF ausgezogen und sie in Indexbilder abgespeichert. Jedes Pixel des Indexbildes verweist auf eine BRDF mit  $81 \times 81$  Dimension. Mit anderen Worten wurde die Verteilung von ähnlichen BRDFs einer BTF auch ermittelt. Stellen wir nun vor, dass wir ein 2D-Textur-Bild zur Hand haben, auf dem ein Mädchen mit einem gelben Hut gemalt wird, und führen dann eine Quantisation darauf durch. Die Pixels mit ähnlichen gelben Pixelwerten werden in das selbe Quader genommen. Wenn wir den Repräsentanten dieses Quaders durch ein Pixel mit rotem Pixelwert ersetzen, dann wird das Mädchen einen neuen roten Hut setzen.

Ebenso wie bei 2D-Textur sind wir schon in der Lage, die BRDFs einer BTF durch Median-Cut-Algorithmus ihren Ähnlichkeiten nach zu quantisieren. Wir sind selbstverständlich neugierig, was passieren kann, wenn wir jetzt statt einiger oder

allen BRDFs weitere BRDFs einsetzen. Was wir auf dieser Weise machen ist in der Tat die Farbe einer BTF auszutauschen, ohne dass die Eigenschaften der Struktur und das Muster einer BTF verändert werden.

Unser Test war ein Indexbild mit den anderen BRDF-Block-Bildern zu verbinden bzw. die BRDF-Blöcke mit verschiedenen Indexbild zu indizieren, damit eine Menge von BTFs mit selber Struktur, aber mit unterschiedlichen Farben vorkommt. Wir zeigen ein Paar BTF-Bilder wie folgt:

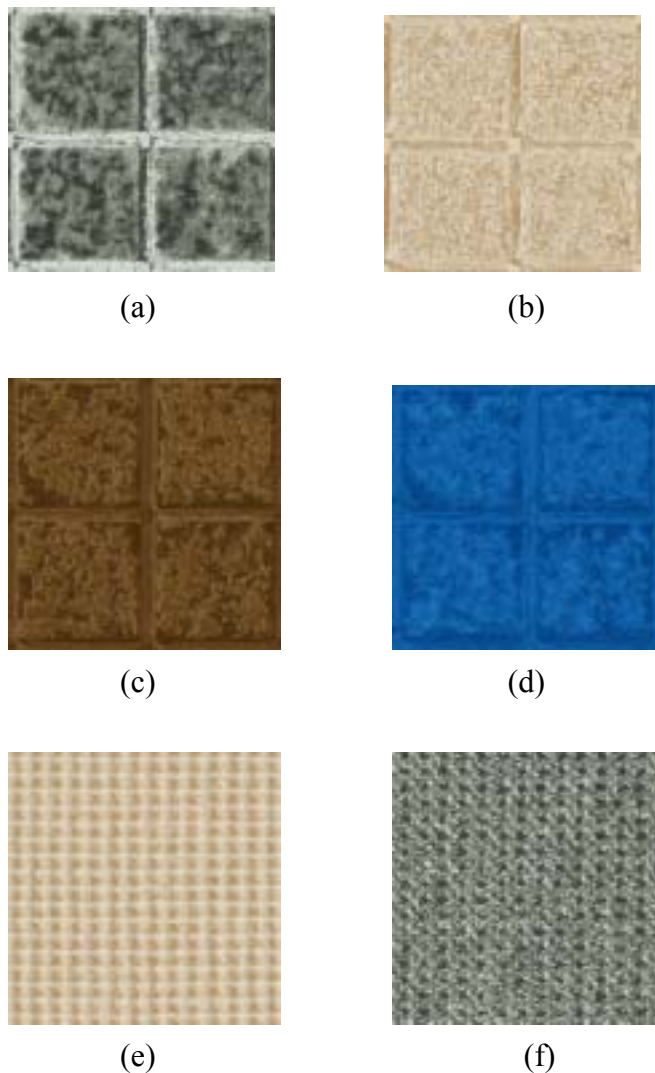


Abbildung 8.8: BTFs verschiedener Farben

- (a) Das originale Bild aus *Impalla*
- (b) Das erzeugte Bild von *Impalla*-Muster und *Proposte*-BRDFs
- (c) Das erzeugte Bild von *Impalla*-Muster und *Corduroy*-BRDFs
- (d) Das erzeugte Bild von *Impalla*-Muster und *Wool*-BRDFs
- (e) Das originale Bild aus *Proposte*
- (f) Das erzeugte Bild von *Proposte*-Muster und *Impalla*-BRDFs



## 8.5.2 Rendern der neuen BTFs in RenderMan

Wenn wir die durch Median-Cut-Algorithmus verarbeiteten BTF-Daten in Shader von RenderMan einsetzen, ist es ziemlich einfach zu implementieren. Die Textur-Bilder, die unser Shader einlesen müssen, sind nur ein Indexbild und ein BRDF-Block-Bild. Wir haben im letzten Kapitel schon gezeigt, dass, wenn wir direkt die originalen BTF-Bilder beim Rendern in RenderMan verwenden wollen, wir die originalen BTF-Bilder in drei Mengen aufzuteilen brauchen und jedes Shader ca. zwei tausende Bilder einladen lassen müssen; wenn wir die mittels SVD komprimierten BTF-Daten in Shader einbringen, haben wir das selbe Problem, eine riesige Menge von Textur-Bildern in Shader einzuladen. Sonst hat jedes IndexBild die Auflösung von  $256 \times 256$  (in 2-Potenz) Pixels, was beim Abtasten von Shader kaum Fehler macht, die beim Rendern mittels SVD häufig auftauchen können.

Der Quellcode beim Ausgeben der Pixelwerte in Shader von RenderMan ist:

```
float index = 256 * round ( texture ( indexImage[1], s, t) * 255)  
                + round ( texture ( indexImage[2], s, t) * 255);  
  
color Ci = color texture( brdfBlockImage,  
                (pca_position * 81 + parameter_position) / width, index / height);
```

Wir zeigen hierbei ein Paar gerenderte Bilder mit verschiedenen BTFs. Wegen der langen Laufzeit des Median-Cut-Algorithmus haben wir die Anzahl von Quadern bei der Quantisation mit 256 eingesetzt.



(a)



(b)



(c)



(d)



Abbildung 8.9: Die gerenderten Bilder aus verschiedenen BTFs

- (a) *Impalla*-Muster aus *Impalla*-BRDFs
- (b) *Impalla*-Muster aus *Proposte*-BRDFs
- (c) *Impalla*-Muster aus *Corduroy*-BRDFs
- (d) *Proposte*-Muster aus *Proposte*-BRDFs
- (e) *Proposte*-Muster aus *Impalla*-BRDFs
- (f) *Proposte*-Muster aus *Corduroy*-BRDFs

## 8.6 Offene Probleme

Obwohl die rekonstruierten BTF-Bilder und die gerenderten Bilder mittels Median-Cut-Algorithmus zurzeit nicht so perfekt aussehen, darf man nicht vergessen, dass wir nur ca. 2 MB (256 Quader) Speicherplatz statt ca. 400 MB für ein BTF-Sample beim Rendern brauchen. Auf andere Seite hat die Verwendung des Median-Cut-Algorithmus bei BTF eine Möglichkeit geweckt, dass neue BTFs aus vorhandenen BRDFs oder BTFs, die durch spezieller und technisch komplexer Aufnahmegerate gemessen wurden, generiert werden können. Dadurch findet man einen weiteren Weg, vielartige BTFs zu erhalten, die aber nicht viel kosten.

Nun bleiben uns noch einige Probleme zu lösen, die wir später auch in andere Literaturen berücksichtigen werden.

- Wie kann man die Qualität der Quantisation von BRDFs verbessern? Außer der Kantenlänge der Quader spielt die Anzahl der BRDFs innerhalb jedes Quaders auch eine wichtige Rolle?
- Was bringt das, wenn man verschiedene Strategien des Median-Cut-Algorithmus der 2D-Texture auf BTF einsetzt?
- Wie kann die Quantisation von BRDFs trotz ihrer riesigen Dimension beschleunigt und die Laufzeit verkürzt werden? Mit welcher Datenstruktur wird die Quantisation effektiver?

- Wie können die inneren Effekte zwischen den BRDFs wie z. B. die Schattierung, Innerreflexion (*interreflections*) und Ausblendung der Flächenelemente beim Austausch einiger BRDFs berücksichtigt werden?
- Mit welchen Methoden kann man die richtige BRDFs so austauschen, dass die Struktur und das Muster einer BTF möglichst unverändert bleiben?

# Kapitel 9

## Zusammenfassung und Ausblick

### 9.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurden Konzepte für eine effektivere Nutzung von BTFs in der photorealistischen Bilderzeugung entwickelt und prototypisch umgesetzt. Der Fokus liegt dabei auf einer vereinfachten Synthese von BTFs aus vorhandenen BRDF- und BTF-Daten sowie in einer effizienten Nutzbarmachung dieser Informationen für Rendering-Prozesse.

Heute spielt BTF eine immer wichtigere Rolle bei der Computergrafik. Der Grund liegt darin, dass die realistische Visualisierung der Material- und Objektoberfläche eine der anspruchvollsten Aufgaben bei den Anwendungen der graphischen Datenverarbeitung darstellt. Auf der einen Seite kann die Verwendung der BTF echte Materialien mit komplexen Reflexionsverhalten erzeugen. Auf der anderen Seite behebt BTF das Problem ärmlicher Texturen und schwacher Lichteffekte in traditionellen 3D-Modellen. Aber die Beschreibung von Oberflächen mittels BTFs bringt verschiedene Probleme mit sich.

- Akquisition von BRDFs und BTFs bedarf spezielle Aufnahmegeräte und der Akquisitionsprozess dauert sehr lange.
- Einmal erstellte Modelle lassen sich nicht parametrisieren. Zur Erzeugung von 3D-Szenen mit vielfältigen Materialien stehen uns nicht so viele BTF-Samples zur Verfügung.
- Die in einem Akquisitionsprozess generierten Datenmenge sind sehr groß, auf solche Daten basierende BTF-Modelle bereiten vielfältige Probleme beim Rendering.

Nachdem die BTF-Datenbank aus der Universität-Bonn unterladen worden ist, sollte eine Kompression wegen der Begrenze des Speicherplatzs durchgeführt werden. Die folgenden Techniken sind bei der Kompression von BTFs anwendbar:

- Singular-Wert-Zerlegung (SVD)
- Normalisierte Zerlegung (ND)
- Gram-Schmidt-Halbwinkel-Differenz (GSHD)
- Verkettete Matrix-Faktorisierung (CMF)
- Homomorphe Faktorisierung (HF)

Beim Rendern von 3D-Szenen wird eine Oberfläche als eine diskrete Menge der Pixel bezeichnet, die in Richtung auf eine eingestellte Kamera oder das menschliche Auge sichtbar sind. Zu jedem Betrachtungspunkt müssen seine Normale und sein lokales Koordinatensystem in Bezug auf Abbildungsart der Textur vorher bestimmt werden. Dabei dient das von Pixar entwickelte Werkzeug „RenderMan“ zu einer guten Plattform, auf der nur beim Modellieren spezifiziert werden soll, was und wo gerendert werden muß, ohne genau festzulegen, welcher Algorithmus eingesetzt werden soll. In RenderMan werden "Modeler" für geometrische Definition und "Renderer" für Farbwiedergabe von Oberflächen voneinander getrennt, deshalb muß man sich nur auf *Shader* konzentrieren können, um die Oberflächen der Objekte von 3D-Szenen mittels BTF realistisch zu beschreiben.

Die Akquisition BTFs auf Grundlage realer Material-Muster ist sehr aufwendig und bedarf spezieller und technisch komplexer Aufnahmegeräte. Mittels Median-Cut-Algorithmus wird die Möglichkeit eröffnet, dass neue BTFs aus vorhandenen BRDFs oder BTFs generiert werden können. Die Idee des Verfahrens ist, die Information des Textur-Musters einer BTF durch Quantisation ihrer BRDFs auszuziehen und die zugeordneten Indizes auf andere BRDFs zu verbinden, so dass die neu erzeugten BTFs dieselbe Muster-Struktur wie die originale BTF haben, aber mit vielartigen Farben aussehen.

## 9.2 Ausblick

Das in dieser Diplomarbeit vorgestellte Verfahren zur BTF kann als Ausgangspunkt für weitere Anwendung und Arbeiten entwickelt werden. Dabei wären folgende Ansätze denkbar:

Die meisten Shader von RenderMan sind sehr einfach implementiert und benutzen die vorhandenen Funktionen, die in *Shading*-Programmiersprache von RenderMan eingepackt werden. Dagegen sind die für BTF aufgebauten Shader sehr schwer zu machen, da sich die Entwicklungsumgebungen, die kostenlos von Internet heruntergeladen sind, für das Programmieren von komplexen Shader nicht eignen. In dieser Arbeit wurden Shader teilweise von Java generiert. Auf diesen Weg könnte ein komplettes Java-Paket so entwickelt werden, dass es eingegebene Parameter übernimmt und unterschiedliche Shader in Bezug auf verschiedene Ansprüche erzeugt.

Bei der Generierung von BTFs gäbe es noch mehr Möglichkeiten. Außer Median-Cut-Algorithmus könnten weitere Quantisation-Techniken mit verschiedenen Strategien eingesetzt werden. Sonst ist es auch ganz wichtig, die Laufzeit zu verkürzen. Eine Idee wäre, die Quantisation müsste nicht direkt auf BRDFs durchgeführt werden, sondern auf ihre zugeordneten Parameter nach einer Durchführung der PCA-Zerlegung, die die Dimension der BRDFs reduziert.

Interessant wäre auch die Anbindung des in dieser Arbeit entwickelten Shader von RenderMan an den Anwendungen auf verschiedenen Plattform wie z. B. C<sup>++</sup> und OpenGL.

## **Anhang A    Anleitung zum Anwendungsprogramm**

### **1   Die Funktionen des Anwendungsprogrammes**

Wie wir im Kapitel 1 erwähnt haben, werden wir zum Abschluss ein Anwendungsprogramm anbieten, das zur Anzeige der originalen BTF-Bilder, der Komprimierung von den BTF-Daten sowie Quantisierung der BRDFs von BTFs dient. Wir erstellen unsere Anwendung im Prinzip in drei Teile, die jeweils eine Funktion davon erfüllen. Damit kann man die Möglichkeit haben, auf einer graphische Oberfläche die BTF-Daten zu behandeln. In den nächsten Abschnitten werden wir die drei Panel und ihre zugehörige Bedienelemente auf der Oberfläche detailliert vorstellen.

### **2   Anzeige der originalen BTF-Bilder**

Für jeden Benutzer, der sich für BTF interessiert und weiter mit BTF bei Erzeugung graphischer Szenen arbeiten will, ist der erste Schritt, die BTF-Bilder in verschiedene Richtungen auf Kamera und Lichtquelle anzuschauen. Im Moment gibt es fünf verschiedene Materialien aus der Welt, die von der Universität Bonn als BTF-Sample gemessen wurden, später kommen eventuell weitere BTF-Sample. Wer mehr über die Eigenschaften der Materialien wissen und die darauf bezogene Literaturen lesen möchte, kann die Homepage der Universität Bonn [BonnTex06] besuchen, was sich immer lohnt. Bei unserer Arbeit organisieren wir die aufgenommenen Textur-Bilder von den BTF-Sample auf derselben Weise wie die Universität Bonn. Jedes BTF-Sample hat 81 Kamerarichtungen, die jeweils 81 Lichtrichtungen haben, also insgesamt 6561 Bilder. Auf Panel unserer Anwendung (Siehe Abbildung 10.1) gibt es drei Bedienelemente, die ein Benutzer auswählen kann, um das BTF-Sample in Bezug auf bestimmte Kamerarichtung und Lichtrichtung zu definieren. Jede bestimmte Richtung ist in der Tat ein Paar von zwei Winkeln ( $Theta \theta, Phi \phi$ ). Die 81 Kamerarichtungen sind nach der Reihenfolge von  $Theta \theta_{view}$  sortiert. Zur jeder Kamerarichtung sind die 81 Lichtrichtungen ebenso nach der Reihenfolge von  $Theta \theta_{light}$  sortiert.

Die Oberfläche der Anwendung sieht wie folgt aus:

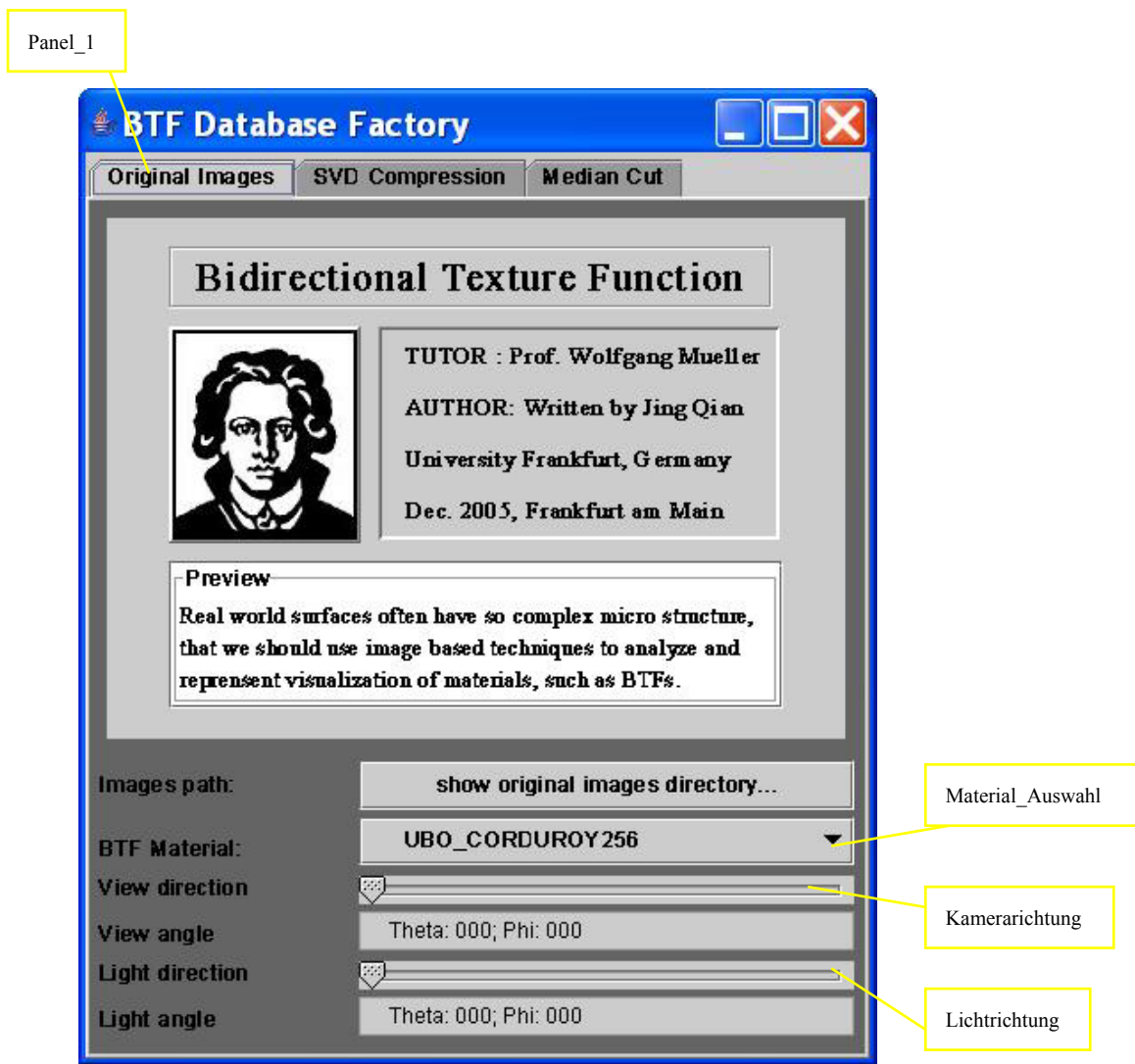


Abbildung 10.1: Das erste Panel der Anwendung

Wenn man den richtigen Pfad eingibt, wo die originalen BTF-Bilder abgespeichert sind, wird das entsprechende Textur-Bild in dem kleinen Fenster „original image“ angezeigt. Das Bild ändert sich sofort bei Veränderung der Kamera- oder Lichtrichtung.

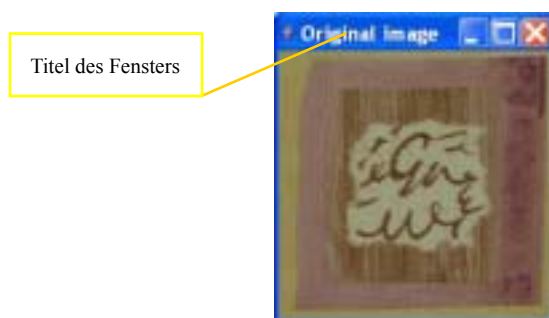


Abbildung 10.2: Der Fenster des originalen Bildes

### 3 Panel zur SVD-Kompression

Das Panel der SVD-Kompression (Siehe Abbildung 10.3) lässt sich in zwei Teile darstellen. Ein Teil dient zur Komprimierung von den originalen BTF-Bildern und erzeugt mittels des Colt-JAVA-Packages die *PCA*-Bilder und ihre zugehörige *Parameter*-Bilder in TIFF-Format. Ein anderer Teil dient zur Rekonstruktion der BTF-Bilder.

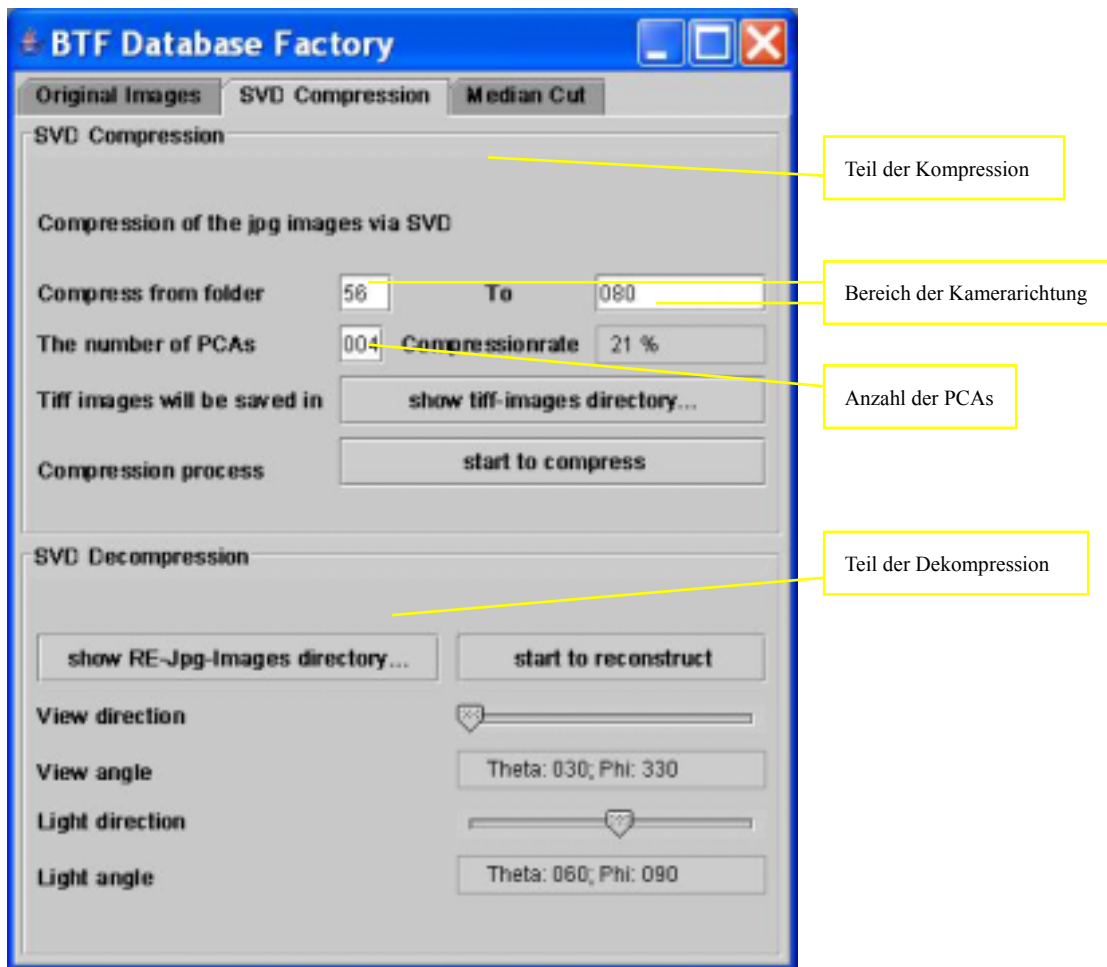


Abbildung 10.3: Das zweite Panel der Anwendung

Nachdem die Textur-Bilder einer BTF rekonstruiert wurden, wird ein Bild im Fenster „SVD Image“ in Bezug auf Kamerarichtung und Lichtrichtung gezeigt. Dazu taucht das Error-Bild im Fenster „Distance-SVD“ auch auf, das die Distanz zwischen dem originalen und dem rekonstruierten Bild bezeichnet. Damit kann man besser beurteilen, welche Leistung die SVD-Kompression in Abhängigkeit von der Anzahl der Hauptkomponenten einbringt. Je dunkeler das Error-Bild aussieht, desto besser funktioniert die Kompression. Wenn alle Pixel des Error-Bildes schwarz sind, kann man festhalten, dass die Textur-Bilder ohne Informationsverlust wiederaufgebaut wurden.





Abbildung 10.4: Die Fenster des Textur-Bildes und des Error-Bildes

#### 4 Panel zum Median-Cut-Algorithmus

Ebenso wie beim Panel der SVD-Kompression läßt sich das Panel zum Median-Cut-Algorithmus in zwei Teile repräsentieren (Siehe Abbildung 10.5).

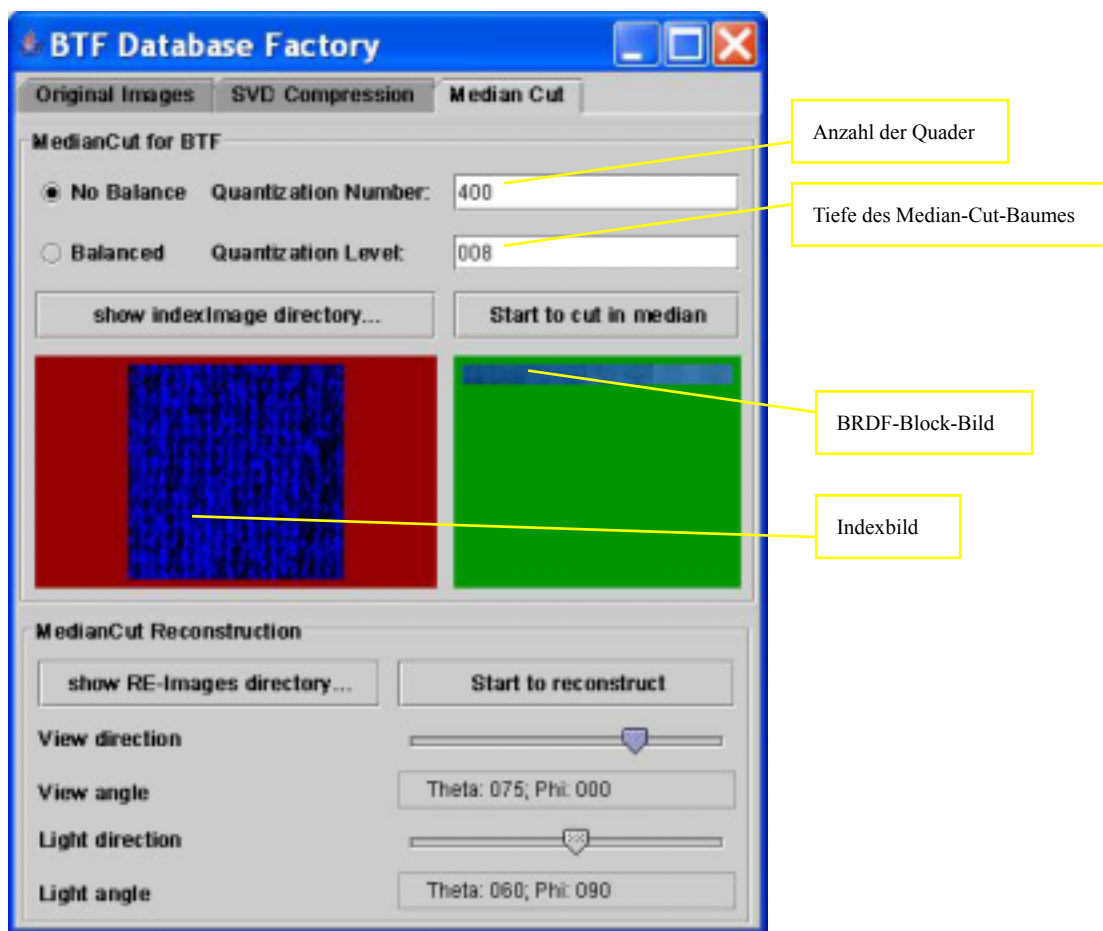


Abbildung 10.5: Das Panel für Median-Cut-Algorithmus

Der obere Teil ist für die Erzeugung des Indexbildes und des BRDF-Block-Bildes verantwortlich. Ein Benutzer hat noch die Möglichkeit, sich zu entscheiden, auf welcher Art und Weise die Quantisation durchgeführt werden sollte. Wenn man die Option „no balance“ wählt, wird der Ablauf der Quantisation viel länger dauern als „balanced“, bei der die Anzahl der Quader in Höhe von  $2^{level}$  ist. Aber die Qualität der Quantisation von „no balance“ ist selbstverständlich besser, das haben wir schon im letzten Kapitel durch die gerenderten Bilder bewiesen.

Auf dem unteren Teil des Panels kann man die BTF-Bilder durch Einstellung des Verzeichnisses, wo die Indexbilder und die BRDF-Block-Bilder abgespeichert sind, wiederaufbauen. Wir gehen immer davon aus, dass ein Benutzer zuerst die originalen BTF-Bilder komprimiert und dann dekomprimiert. Aus diesem Grund wird nichts im Fenster „MedianCut image“ (Siehe Abbildung 10.6) angezeigt, wenn keine Indexbilder und BRDF-Block-Bilder in dem gegebenen Verzeichnis vorhanden sind.

Nachdem die Textur-Bilder einer BTF rekonstruiert wurden, werden zwei Bilder im Fenster „MedianCut image“ und Fenster „Distance-MedianCut“ bezeichnet. Ähnlich wie beim Panel der SVD-Kompression zeigt das Error-Bild im Fenster „Distance-MedianCut“ den Unterschied zwischen dem originalen und dem rekonstruierten Textur-Bild.



Abbildung 10.6: Die Fenster des Textur-Bildes und des Error-Bildes

## 5 Möglichkeiten zum Erweitern

Obwohl wir bereits viele Zeit und Energie bei dieser Arbeit investiert haben, sind wir immer davon überzeugt, dass es bestimmt noch Möglichkeiten gibt, sie weiter zu verbessern und zu erweitern. Z. B. kann man es auf dem Panel ermöglichen, durch die Auswahl von einigen BRDFs, nicht allen BRDFs, die zu einer BTF gehören, eine Menge von neuen BTFs zu erzeugen. Oder das Panel kann so weiter entwickelt werden, dass ein Benutzer mit mehreren Varianten der Strategien vom Median-Cut-Algorithmus die BRDFs besser quantisieren kann.

## Literaturverzeichnis

[**Addy04**] Addy Ngan, Fredo Durand, Wojciech Matusik. Experimental Validation of Analytical BRDF Models. *SIGGRAPH 2004*.

[**Andrews77**] H. Andrews and Hunt. Digital Image Restoration. *Prentice-Hall, 1977*.

[**Anthony99**] Anthony A. Apodaca, Larry Gritz, Ronen Barzel, Sharon Calahan, Clint Hanson, Scott Johnston. Advanced RenderMan: Creating CGI for Motion Pictures. *Morgan Kaufman, 1999*.

[**Ashikhmin00**] Ashikhmin, Michael, Simon Premoze, and Peter Shirley. A micro facet-based BRDF generator; *Computer Graphics (SIGGRAPH 2000 Proceedings)*, pp. 67-74, July 2000.

[**Blinn77**] James F. Blinn. Models of Light Reflection for Computer Synthesized Pictures. *Computer Graphics, Vol. 11, No.2, 1977*.

[**BonnTex06**] Bonn BTF Database, <http://btf.cs.uni-bonn.de/>. *AG Computergraphik, FB Informatik, University Bonn, zuletzt besucht: 21.01.2006*.

[**Burt83**] P. J. Burt and E. H. Adelson. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics, 2(4):217–236, Oct. 1983*.

[**Calkins**] Katherine Calkins. Thesis presentation: An Empirical Model of Thin Film Interference Effects in Computer Graphics. Slides von [http://www-viz.tamu.edu/showcase/thswking/k\\_calkins/presentation/4\\_prev%ios/pr evious01.html](http://www-viz.tamu.edu/showcase/thswking/k_calkins/presentation/4_prev%ios/pr evious01.html).

[**Cohen93**] Michael F. Cohen und John R. Wallace: Radiosity and realistic image synthesis. *Academic Press Professional, Bosten, MA, 1993*.

[**Cornell06**] Cornell BRDF Database. *NSF Graphics and Visualization Center, Cornell University*. <http://www.graphics.cornell.edu/online/measurements/reflectance>, *zuletzt besucht: 21.01.2006*.

[**Cook81**] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. *Technical report, Computer Graphics, vol. 15, no. 3, ACM., 1981*.

- [Curet06] Columbia Utrecht Texture Database. *Columbia University and Utrecht University*, <http://www1.cs.columbia.edu/CAVE/curet/>, zuletzt besucht: 21.01.2006.
- [Dana98] K.J. Dana and S.K. Nayar. Histogram model for 3D textures. *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 618-624, June 1998.
- [Dana99] K. J. Dana, B. van Ginneken, S. K. Nayar, and J. J. Koenderink. Reflectance and Texture of RealWorld Surfaces. *ACM Transactions on Graphics*, 1(18):1-34, 1999.
- [Dana99b] Dana, K., Nayar, S.. 3D Textured Surface Modeling. *WIAGMOR Workshop, IEEE Conference on Computer Vision and Pattern Recognition*, 1999.
- [Efros99] Efros, A., Leung, T.. Texture Synthesis by Non-parametric Sampling. *IEEE International Conference on Computer Vision, Corfu, Greece*, pp. 1033-1038, Sept. 1999.
- [Efros01] Efros, A., Freeman, W.. Image Quilting for Texture Synthesis and Transfer. *SIGGRAPH 2001*, pp. 341-346, 2001.
- [Foley90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice. Addison-Wesley, second edition, 1990.*
- [Gebhardt00] Nikolaus Gebhardt. *Einige BRDF Modelle. 2000*
- [Giraud04] Luc Giraud, Julien Langou, Miroslav. Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numerische Mathematik 2004*
- [He91] Xiao D. He, Kenneth E. Torrance, Francois X. Sillion, and Donald P. Greenberg. A Comprehensive Physical Model for Light Reflection. *Technical report, Cornell University, 1991.*
- [Hill03] F. S. Hill, JR. *Computer graphics using OpenGL. second edition 2003*
- [Jolliffe92] Jolliffe I. *Principal Component Analysis. Springer-Verlag, 1992.*
- [Kautz99] Jan Kautz, Michael D. McCool. Interactive Rendering with arbitrary BRDFs using separable Approximations. *In Eurographics Workshop on Rendering, 1999*
- [Kautz00] Kautz J, Seidel H-P. Towards Interactive Bump Mapping with Anisotrope Shift-Variant BRDFs. *Proceedings of Graphics Hardware 2000*, pp. 51-58, 2000.

[**Kautz04**] Kautz J., Sattler M., Sarlette R., Klein R., Seidel H.-P. Decoupling BRDFs from surface mesostructures. *To appear in proceedings of Graphics Interface 2004, 2004.*

[**Koudelka03**] Koudelka M. L., Magda S., Belhumeur P. N., Kriegman D. J.: Acquisition, compression and synthesis of bidirectional texture functions. In *3rd International Workshop on Texture Analysis and Synthesis, 2003.*

[**Lafortune97**] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-Linear Approximation of Reflectance Functions. *Technical report, Cornell University, 1997.*

[**Larsen00**] Bent Dalgaard Larsen. Shading a Surface using BRDFs. *2000*

[**Latta02**] Lutz Latta and Andreas Kolb . Homomorphic Factorization of BRDF-based Lighting Computation. *ACM SIGGRAPH 2002, July 2002*

[**Lee99**] D.D. Lee and H.S. Seung. Learning the parts of objects by non-negative matrix factorization. *1999.*

[**Leung01**] Leung, T., Malik J.. Representation and Recognizing the Visual Appearance of Materials using Three-dimensional Textons. *International Journal of Computer Vision, 43(1):29-44, 2001.*

[**Liang01**] Liang, L., Lieu, C., Xu, Y., Guo, B., Shum, H. Real-Time Texture Synthesis by Patch Based Sampling. *ACM Transactions on Graphics, Vol. 20, No. 3, July 2001, pp. 127-150.*

[**Liu01**] Liu, X., Yu, Y., Shum, H.. Synthesizing Bidirectional Texture Functions for Real-World Surfaces. *SIGGRAPH 2001, pp. 97-106, 2001.*

[**Marschner99**] Stephen R. Marschner, Stephen H. Westin Eric P. F. Lafortune Kenneth E. Torrance Donald P. Greenberg. Image-Based BRDF Measurement Including Human Skin. *Program of Computer Graphics Cornell University. 1999*

[**McAllister02**] D. McAllister, A. Lastra, and W. Heidrich. Efficient rendering of spatial bidirectional reflectance distribution functions. *Graphics Hardware 2002, Eurographics / SIGGRAPH Workshop Proceedings, 2002.*

[**McCool01**] McCool, Michael D., Jason Ang., and Anis Ahmad. Homomorphic Factorization of BRDFs for High-performance Rendering. *Computer Graphics(SIGGRAPH 2001 Proceedings), 2001.*

- [Meseth03a]** Meseth J., Müller G., Klein R.: Preserving realism in real-time rendering of bidirectional texture functions. In *OpenSG Symposium 2003* (April 2003), Eurographics Association, Switzerland, pp. 89–96, 2003.
- [Meseth03b]** Müller G., Meseth J., Klein R.: Compression and real-time Rendering of Measured BTFs using local PCA. In *Vision, Modeling and Visualisation 2003*, pp. 271–280, November 2003.
- [Meseth03c]** Meseth J., Müller G., Sattler M., Klein R.: BTF rendering for virtual environments. In *Virtual Concepts 2003*, pp. 356–363, November 2003.
- [Meseth04a]** Meseth J., Müller G., Klein R.: Reflectance field based real-time, high-quality rendering of bidirectional texture functions. *Computers and Graphics* 28, 103–112, February 2004.
- [Meseth04b]** Müller G., Meseth J., Klein R.: Fast Environmental Lighting for Local-PCA Encoded BTFs. In *Computer Graphics International 2004 (CGI2004)*, June 2004.
- [Michael00]** Michael Ashikhmin, Peter Shirley. An Anisotropic Phong BRDF Model., *Journal of Graphics Tools: JGT*, August 2000.
- [Möller99]** Thomas Möller, Eric Haines. Real-Time Rendering. *AK Peters*, 1999.
- [Moon04]** Jon Moon, Steve Marschner. The BRDF, light transport in a Vacuum. 2004
- [Müller04]** G. Müller, J. Meseth, M.Sattler, R.Sarlette, R. Klein. Acquisition, Synthesis and Rendering of Bidirectional Texture Funktionen. *EUROGRAPHICS 2004*, 2004.
- [Nayar89]** Shree K. Nayar, Eatsushi Ikeuchi, and Takeo Kanade. Surface Reflection: Physical and Geometrical Perspectives. *Technical report, The Robotics Institute, Carnegie Mellon University*, 1989.
- [Nene97]** S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19:989–1003, 1997.
- [Ngan04]** Addy Ngan, Frédo Durand, Wojciech Matusik. Experimental Validation of Analytical BRDF Models. *ACM SIGGRAPH 2004*, 2004.

- [**Nicodemus77**] Nicodemus, F. E., J. C. Richmond, J. J. Hsia, I. W. Ginsberg and T. Limperis. Geometric considerations and Nomenclature for reflectance. *National Bureau of Standards(US)*, October 1977.
- [**Pfister99**] Hanspeter Pfister. BRDFs and the Rendering Equation. *Anselmo Lastra, UNC, Leonard McMillan, MIT*, 1999.
- [**Praun00**] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. *Proceedings of SIGGRAPH 2000*, pages 465–470, July 2000.
- [**Rusinkiewicz01**] Szymon M. Rusinkiewicz. A New Change of Variables for Efficient BRDF Representation. 2001.
- [**Sattler03**] Sattler, M., Sarlette, R., And Klein. Efficient and realistic visualization of cloth. In *Proceedings of Eurographics Symposium on Rendering 2003*, pp. 167.177, 2003.
- [**Schlick94**] Christophe Schlick. An inexpensive BRDF model for physically based rendering. *Technical report*, 1994.
- [**Schneider04**] Schneider M.: Real-Time BTF Rendering. In *Proceedings of CESC 2004*, 2004.
- [**Suykens03**] Suykens, F., Vom Berge, K., Lagae, A., And Dutré. Interactive rendering with bidirectional texture functions. *Computer Graphics Forum* 22, 3, 463.472. (*Proceedings of Eurographics 2003*), 2003.
- [**Tong 02**] Tong, X., Zhang, J., Liu, L., Wang, X., Guo, B.I, Shum, H.. Synthesis of Bidirectional Texture Functions on Arbitrary Surfaces. *SIGGRAPH 2002*, 2002.
- [**Upstill92**] Steve Upstill. The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics. *Addison-Wesley*, second edition, 1992
- [**Ward92**] Gregory J. Ward. Measuring and modeling anisotropic reflection. *ACM SIGGRAPH 1992*, 1992.
- [**Watt00**] Alan Watt. 3D Computer Graphics. *Pearson Education Limited, Harlow, Essex CM20 2JE England*, 2000.
- [**Wei00**] Wei, L., Levoy, M.. Fast Texture Synthesis Using Tree-Structured Vector Quantization. *SIGGRAPH 2000*, pp. 479-488, 2000.

**[Wong97]** T.-T. Wong, P.-A. Heng, S.-H. Or, and W.-Y. Ng. Image-based rendering with controllable illumination. *In Rendering Techniques 97, pages 13–22, 1997.*

**[Wynn00]** Chris Wynn. An Introduction to BRDFBased Lighting. *Technical report, NVIDIA Corporation, 2000.*

**[Yakov03]** Yacov Hel-Or, Tom Malzbender, Dan Gelb. Synthesis of Reflectance Function Textures from Examples. *Hewlett-Packard Laboratories, 2003*

**[Yee02]** Hector Y. Yee, Philip Dutr'e, and Sumanta N. Pattanaik. Fundamentals of Lighting and Perception: The Rendering of Physically Accurate Images. *In Game Developer Conference 2002 Proceedings, 2002.*