Hybrides Verfahren zur realistischen, echtzeitfähigen Beleuchtung virtueller Szenen unter Berücksichtigung dynamischer Objekte auf der Basis von Spherical Harmonics

Diplomarbeit

Vorgelegt von

Matthias Biedermann

UNIVERSITÄT KOBLENZ · LANDAU

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer: Dipl.-Inform. Thorsten Grosch
Prüfer: Prof. Dr.-Ing. Stefan Müller

Juni 2004

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass die vorliegende Arbeit selbständig verfasst wurde und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet wurden.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

..... , den ....

(Ort, Datum) (Unterschrift)

Danksagung

Eine solche Arbeit ist ohne die fachliche und inhaltliche wie auch die persönliche Unterstützung vieler Menschen kaum denkbar. Daher möchte ich diese Gelegenheit nutzen und mich an dieser Stelle persönlich dafür bedanken.

In erster Linie ist dabei Prof. Dr.-Ing. Stefan Müller zu nennen, der mich schon im Rahmen der Studienarbeit auf das spannende, wenn auch recht anspruchsvolle Thema „Spherical Harmonics“ aufmerksam gemacht hat. Natürlich möchte ich dabei auch nicht meinen Betreuer, Dipl.-Inform. Thorsten Grosch sowie Dipl.-Inform. Markus Geimer unerwähnt lassen, ohne deren Unterstützung in Form von anregenden Diskussionen, konstruktiver Kritik und einem jederzeit offenen Ohr diese Arbeit sicherlich nicht zu dem geworden wäre, was sie ist.
Darüber hinaus möchte ich der Volkswagen AG für die gute und unkomplizierte Zusammenarbeit bei der Bereitstellung von Testdaten danken, die insbesondere auf den guten Kontakt und das Engagement von Herrn Steffen Haupt sowie Herrn Frank Purschke zurückzuführen ist.
Vor allem die im Zusammenhang mit der Implementation aufgetauchten Fragen und Probleme konnten mit Hilfe verschiedener Mailinglisten, Foren und persönlichen Mails geklärt werden. Auch die Umsetzung in OpenSG wäre sicherlich nicht ohne diese Unterstützung möglich gewesen, für die ich mich insbesondere bei Dr. Dirk Reiners bedanken möchte.
Nicht unerwähnt möchte ich aber auch die zahlreichen Gespräche mit vielen Studienkolleginnen und -kollegen lassen, die meine Arbeit oft auch gleichzeitig als Freunde sowohl mit guten Ideen als auch anderen Blickwinkeln in vielerlei Hinsicht bereichert und verbessert haben.

Zu guter Letzt möchte ich mich natürlich auch bei meiner Familie für die Ermöglichung und Unterstützung dieses Studiums bedanken. Dieser Dank gilt in besonderer Weise meiner Frau Alexandra, die trotz der mitunter wenigen gemeinsamen und bei einer solchen Arbeit auch nicht immer stressfreien Zeit zu mir hielt und mich unterstützte.

Matthias Biedermann

Inhaltsverzeichnis

Danksagung **iii**

Einleitung **1**

I Theoretische Grundlagen **3**

1 Spherical Harmonics **5**

1.1 Grundlegende Konzepte 5

1.1.1 Von Fourier zu den Spherical Harmonics 5

1.1.2 Erweiterung auf die Kugel 7

1.1.3 Die Basisfunktionen der Spherical Harmonics 8

1.2 Besondere Eigenschaften 11

1.2.1 Linearität der Approximation 12

1.2.2 Tiefpassfilter-Analogie 12

1.2.3 Die Rotationsinvarianz 14

1.2.4 Vom Doppelintegral zum inneren Produkt 15

1.3 Die Verarbeitung der Spherical Harmonics 16

1.3.1 Die Vorverarbeitungsschritte 16

1.3.2 Die Schritte beim Rendern 18

2 Verschiedene Technologien **21**

2.1 Programmierbare Grafikhardware 21

2.1.1 Allgemeine Grundlagen 21

2.1.2 Die Grafikprogrammiersprache Cg 24

2.2 Verarbeitung von HDR-Bildern 25

2.2.1 HDR-Bilder 26

2.2.2 Tonemapping-Verfahren 29

2.3 Schattenverfahren 30

2.3.1 Direkte Erzeugung von Schatten 31

2.3.2 Ambient Occlusion 34

II Spherical Harmonics in dynamischen Umgebungen 39

3 Dynamische Beleuchtung 41
3.1 Umgebungsbeleuchtung 41
3.2 Lokale Beleuchtung 42
3.2.1 Radiance Sampling 42
3.2.2 Vertex-basierte Beleuchtungsgewichtung 43

4 Dynamik der Szene 47
4.1 Lokale Dynamik 47
4.1.1 Objekte als Ganzes 47
4.1.2 Festkörper-Transformationen 49
4.1.3 Dynamik höherer Transferfunktionen 49
4.2 Dynamik mit globalem Einfluss 50
4.2.1 Keyframing 51
4.2.2 Vorberechnete Dynamiksimulationen 52
4.2.3 Nachbarschaftstransfer 54

III Die praktische Umsetzung 59

5 Die verwendeten Komponenten 61
5.1 Die Plattform 61
5.1.1 Programmiersprache 61
5.1.2 Betriebssystem und Hardware 62
5.2 Das Renderingsystem 63
5.2.1 Integration eines `half`-Datentyps 63
5.2.2 Erweiterung der vorhandenen Cg-Schnittstelle 64
5.2.3 Spezialisieren der `OSGQGLManagedWidget`-Klasse . 64

6 Beschreibung des Systems 66
6.1 Allgemeiner Überblick 66
6.1.1 Das Grundkonzept 66
6.1.2 Designentscheidungen 68
6.2 Die Bedienung 71
6.2.1 Die Programmoberfläche 71

6.2.2 Laden von Elementen 72
6.2.3 Editieren der Szene 73
6.2.4 Vorberechnungen 73
6.3 Technische Details 76
6.3.1 Cube Mapping 76
6.3.2 Tonemapping per Fragmentshader 77
6.3.3 Spherical Harmonics auf der GPU 80
6.3.4 Stereographische Projektion 81

7 Ergebnisse und Bewertung 85
7.1 Verarbeitung herkömmlicher Daten 85
7.1.1 Darstellung von HDR-Bildern 85
7.1.2 Einfache Modelle 86
7.2 Der Umgang mit komplexeren Daten 88
7.3 Resümee 88

Zusammenfassung und Ausblick 91

Anhang 95

Literaturverzeichnis 100

Abbildungsverzeichnis

1 Die Legendre-Polynome 10
2 Die Spherical Harmonics 11
3 SH-Koeffizienten einer Testszene 13
4 Die Struktur von SH-Rotationsmatrizen 19
5 Programmierbare Komponenten der OpenGL-Pipeline . . . 23
6 Vergleich von Ambient Occlusion per Shadowmapping . . . 37
7 Gegenüberstellung lokaler und entfernter Beleuchtung . . . 42
8 Skizze für lokale SH-Beleuchtung mit Cubemaps 44
9 Shading mit und ohne Ambient Occlusion 50
10 SH-Beleuchtung deformierbarer Objekte 54
11 Verschiedene Möglichkeiten des Nachbarschaftstransfers . . 55
12 Der Szenengraph im Überblick 67
13 Programmoberfläche von „sashimi“ 72
14 Fragmentprogramme für HDR-Darstellung 79
15 Vertexprogramm für SH-Beleuchtung 82
16 Vertexprogramm für stereographische Projektion 83
17 Wirkung der stereographischen Projektion 83
18 Beispielmodell „harley“ mit Ambient Occlusion 87
19 Modell „harley“ mit AO und SH 87
20 Polo-Modell mit Ambient Occlusion und Spherical Harmonics 89
21 Polo-Modell mit Fresnel-Material und Spherical Harmonics 89
22 Polo-Modell mit dynamischer EnvironmentMap 90

Einleitung

In den letzten Jahren hat die Entwicklung der Grafikhardware den schnellen Fortschritt der allgemeinen Computertechnik um ein Vielfaches überholt. Neben der ständig gesteigerten Verarbeitungsgeschwindigkeit moderner Grafikchips ist auch immer mehr ein Trend zu flexibleren Programmiermöglichkeiten zu erkennen. Während früher die Funktionalität der Grafikkarte nahezu vollständig durch die Hardware vorgegeben war und Programmierer nur in geringem Umfang Einfluß auf die Verarbeitung nehmen konnten, bieten Grafikkarten der aktuellen (vierten) Generation ein hohes Maß an Programmierbarkeit. Dies zeigt sich neben den Spezifikationen der Hardware selbst auch an der mittlerweile sehr großen Zahl von Implementationen und Verwendungszwecken.

Obwohl diese Entwicklung neue Qualitätsstufen in der Echtzeitgrafik ermöglicht hat, können immer noch nicht alle für eine (photo-)realistische Darstellung der Szene benötigten Berechnungen in akzeptabler Zeit durchgeführt werden. Vor allem die Erzeugung globaler Beleuchtungseffekte wie realistische Schatten oder indirekte Beleuchtung in einer Szene sind ohne starke und damit sichtbare Vereinfachungen nicht realisierbar. Zwar haben die aus der Quantenphysik stammenden Spherical Harmonics eine effiziente Möglichkeit geschaffen, solche globalen Effekte darzustellen; gewisse Grundannahmen dieser Technologie schränken ihren universellen Einsatz jedoch ein.
Eine denkbare Lösung, die in dieser Arbeit theoretisch und exemplarisch auch in der Beispielimplementation verfolgt werden soll, ist die Kombination der Vorteile sowohl herkömmlicher als auch auf Spherical Harmonics basierender Echtzeitgrafik. Bisherige Verfahren setzen als grundlegende vereinfachende Annahme Lichtquellen in Punktgröße voraus; natürliche Beleuchtungsverhältnisse, die meist nur durch flächenförmige Lichtquellen reproduzierbar sind, lassen sich mit dieser Annahme kaum vereinbaren. Die Repräsentation mit Hilfe der Sphärisch-Harmonischen Funktionen hingegen wird umso effizienter, je *größer* die Lichtbereiche sind.
In diesem Zusammenhang ist es für die physikalische Plausibilität unabdingbar, die Beleuchtungsinformation mittels HDR-Daten (HDR = High Dynamic Range) zu repräsentieren. Auch hier bieten die neuesten Grafikkarten die Möglichkeit, Echzeitdarstellungen auf Basis dieser HDR-Daten zu erzeugen und dadurch deutlich realistischer aussehen zu lassen. Und einmal mehr sind es insbesondere aktuelle Entwicklungen der Spieleindustrie, die sich diese Technik zunutze machen, um den Realitätsgrad immer wieder in neue Dimensionen zu steigern.

Ausgehend von diesen sehr schnellen und spannenden Entwicklungen verfolgt die vorliegende Arbeit hauptsächlich zwei Ziele.

Zum einen soll ein hybrides System entworfen und umgesetzt werden, das die Vorteile der herkömmlichen Echzeitgrafik mit den Möglichkeiten der Beleuchtung auf Basis der Spherical Harmonics verbindet. Eine grundlegende Untersuchung vorhandener Techniken für die echtzeitfähige Beleuchtung virtueller Objekte in Kombination mit Spherical Harmonics steht dabei im Vordergrund und wird durch eine Erläuterung besonderer programmiertechnischer Details der Beispielimplementation abgerundet.
Der zweite Aspekt der Arbeit setzt sich theoretisch mit Möglichkeiten auseinander, neben statischen auch dynamische Szenen darstellen zu können. Wie oben schon angedeutet, können diese wichtige Art von Visualisierungen nicht ohne weiteres mit Techniken wie den Spherical Harmonics umgesetzt werden, da diese aufgrund ihrer Vorberechnungen nur für statische Geometrien geeignet sind. Dabei werden verschiedene, bereits existierende Lösungsansätze besprochen und in dem durch das Thema beschriebenen Kontext bewertet.

Teil I
Theoretische Grundlagen

In dem ersten Teil der Arbeit wird auf die für die vorliegende Arbeit benötigten Spherical Harmonics eingegangen. Neben einer Erläuterung der wichtigsten theoretischen Grundlagen und Zusammenhänge dieses mathematischen Systems soll auch auf die prinzipielle Verwendung in der Praxis eingegangen werden.
Darüber hinaus werden weitere technologische Grundlagen erarbeitet, die für den weiteren Verlauf der Arbeit nötig sind. Dies schließt zum einen die rasanten Fortschritte im Bereich der Grafikhardware ein, deren Potential mit Hilfe von Shaderprogrammen flexibel ausgenutzt werden kann. Zudem bieten neueste Entwicklungen auch die Möglichkeit, die Spherical Harmonics basierte Beleuchtung unabdingbaren HDR-Bilder direkt auf der Grafikhardware zu verarbeiten. Sozusagen als Gegenteil von Beleuchtung bildet die Erläuterung verschiedener Schattenverfahren den Abschluss dieses ersten Teiles, der zudem als Grundlage für die weiteren Kapitel der vorliegenden Arbeit dienen soll.

1 Spherical Harmonics

Obwohl die Spherical Harmonics (sphärisch-harmonische Funktionen, Kugelfunktionen) als mathematisches Werkzeug in vielen Naturwissenschaften schon seit Jahrzehnten im Einsatz sind, spielen sie erst seit relativ kurzer Zeit auch in der Computergrafik eine Rolle. Während frühe Arbeiten wie [CMS87] oder [WAT92] eher eine platzsparendere und effizientere Beschreibung von Material- und Reflexionseigenschaften anstrebten, haben im Wesentlichen die Arbeiten von Ramamoorthi et al. [RH01a], [RH01b] und Sloan et al. [SKS02] diese Technik für eine echtzeitfähige Beschreibung natürlicher Beleuchtungssituationen etabliert.
Ausgehend von der Idee dieser Arbeiten sollen diese Funktionen auch eine wesentliche Komponente des zu entwerfenden Systems bilden. Eine Repräsentation der Beleuchtung auf dieser Basis hat unter anderem den Vorteil, für großflächige Lichtquellen sehr gut geeignet zu sein. Um mit dieser relativ komplexen Technik und den damit verbundenen Begriffen umgehen zu können, bedarf es jedoch zunächst einer grundlegenden Einführung und Motivation für die Spherical Harmonics, wie sie in umfangreicherer Form in [Bie03] und den entsprechenden Referenzen zu finden ist.

1.1 Grundlegende Konzepte

1.1.1 Von Fourier zu den Spherical Harmonics

Wenn man sich Literatur zu Spherical Harmonics anschaut, wird oft nur am Rande erwähnt, dass diese Funktionen auf den Fourierreihen basieren und als eine Erweiterung dieser aufgefasst werden können. In der Signalverarbeitung im Allgemeinen und in der Bildverarbeitung im Speziellen gehört die Fouriertransformation jedoch schon seit sehr langer Zeit zu einem Standardwerkzeug, weshalb auch für eine weitergehende Betrachtung auf [Pau02] und [Jäh97] verwiesen wird. Die Aufgaben der Fouriertransformation im Bereich der digitalen Bildverarbeitung sind dabei sehr weitläufig:

- Hervorhebung und/oder Isolierung von bestimmten Bildmerkmalen (z.B. periodische Bildstörungen)
- effiziente Transformationen von Bildern (Skalierung, Rotation)
- Ausnutzung bestimmter Eigenschaften zur Vereinfachung und Verbesserung von Prozessen (Positionierung, Filterung, statistische Methoden etc.)
- Kompression der (sichtbaren) Bildinformation auf einen Bruchteil der ursprünglichen Datenmenge

Vor allem die letzten beiden Eigenschaften, also die effizientere Verarbeitung und Reduktion der für die Darstellung benötigten Daten ist ein wichtiger Grund, warum die photorealistische Computergrafik auf diese Technologie nicht verzichten sollte.

Prinzipiell ist die Fouriertransformation eine reversible Abbildung vom Ortsraum in den Frequenzraum, d.h. die Bildinformation wird nicht durch das **räumliche** Gefüge der Signalwerte (Helligkeit der Pixel), sondern durch die Kombination verschiedener **Frequenzen** von Basisfunktionen repräsentiert. Für diesen Prozess wird das zweidimensionale Bild üblicherweise als kontinuierliches, periodisches Signal angenommen, indem man sich das Bild sozusagen „unendlich gekachelt" oder zu einem Torus „gefaltet" vorstellt.
Um das Ganze mathematisch etwas zu konkretisieren, seien im Folgenden die wichtigsten Formeln beim Umgang mit der Fouriertransformation erläutert, die für das Verständnis der Mathematik der Spherical Harmonics unabdingbar sind. Für eine detailliertere Betrachtung sei jedoch auf einschlägige Literatur verwiesen.
Zunächst ist die allgemein übliche Formulierung der Fourierreihe in der (komplexwertigen) Euler'schen Schreibweise wie folgt definiert:

$$f(t) \;=\; \frac{1}{2\pi} \sum_{k=-\infty}^{+\infty} c_k e^{ikt} \quad \text{, mit } k \in \mathbb{Z},\; c \in \mathbb{C} \tag{1.1}$$

wobei die Koeffizienten c_k dabei beschrieben sind durch

$$c_k = \int_{-\pi}^{\pi} f(t) e^{-ikt} dt \tag{1.2}$$

Nimmt man nun an, dass das Eingangssignal nicht nur 2π-periodisch ist, sondern das Periodizitätsintervall gegen unendlich geht, erweitert sich die Koeffizientenformel zur sog. **Gewichtungsfunktion**, die im Allgemeinen auch als die **Fouriertransformierte** der Funktion $f(t)$ bezeichnet wird.

$$F(\xi) = \int_{-\infty}^{+\infty} f(t) e^{-i\xi t} dt \quad \text{, mit } \xi \in \mathbb{R} \tag{1.3}$$

Damit lässt sich schließlich die **inverse** Transformation notieren als:

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} F(\xi) e^{i\xi t} d\xi \quad . \tag{1.4}$$

Mit dieser Beschreibung kann nun jede beliebige periodische, **ein**dimensionale Funktion mit Hilfe der Fouriertransformation repräsentiert und in beide Richtungen umgewandelt werden. Somit ist es dann auch ohne weiteres möglich, Operationen im Ortsraum durch ihr Äquivalent im Frequenzraum auf das „Bild“ anzuwenden, um nach einer anschließenden Rücktransformation das gemäß der Operation veränderte Ausgangsbild zu erhalten. Ein sehr gutes Beispiel dafür ist die schon anfangs erwähnte Faltung. Dieser, im Ortsraum unter Umständen recht aufwändigen Operation, entspricht nach einer Transformation in den Frequenzraum nichts Einfacheres als eine Multiplikation.
Darüber hinaus gibt es noch weitere Eigenschaften, die in der transformierten Repräsentation direkt ausgenutzt werden können. Als Stichwort sei hier beispielsweise die Verschiebungsinvarianz genannt, deren Pendant auch bei den Spherical Harmonics existiert (siehe Abschnitt 1.2.3).

1.1.2 Erweiterung auf die Kugel

Für den Übergang von der Fouriertransformation zu den Sphärisch-Harmonischen Funktionen ist es nötig, die (Bild-)Eingabedaten umzuwandeln. Abgesehen von der universellen, eindimensionalen Parametrisierung der Funktionen wie beispielsweise zur Fouriertransformation durch t, wurde aufgrund der üblichen Verwendung in der Computergrafik bisher implizit von kartesischen Koordinaten (x, y) bzw. (x, y, z) ausgegangen.
Im Falle der Beleuchtung einer virtuellen Szene durch eine reale (Beleuchtungs-)Umgebung greift man jedoch üblicherweise auf die sogenannten **sphärischen** Koordinaten (θ, φ) zurück, da die Verteilung von Beleuchtungsstärken eine Richtungsfunktion ist. Diese sphärischen Koordinaten lassen sich aus (x, y, z) wie folgt berechnen:

$$\theta = \arccos\left(\frac{y}{r}\right) \quad \text{, mit } r = \sqrt{x^2 + y^2 + z^2} \tag{1.5}$$

$$\varphi = \begin{cases} \arccos\left(\dfrac{x}{r\sin\theta}\right) & \text{für } z \geq 0 \\ -\arccos\left(\dfrac{x}{r\sin\theta}\right) & \text{für } z < 0 \end{cases} \tag{1.6}$$ [1]

θ beschreibt dabei den Winkel zwischen der vertikalen Achse (üblicherweise y) und der vertikalen Auslenkung; man bezeichnet diesen Wert daher auch als „Latitudinale“ (*engl.:* latitude = Breitengrad) oder „elevation“ (*engl.:* Erhebung). Der horizontale Winkel φ ist als Winkel zwischen der x-Achse und der „Longitudinalen“ (*engl.:* longitude = Längengrad) des Punktes definiert; üblich ist hier auch der Begriff „azimuth“ (*engl.:* Drehwinkel).

[1] Alternativ kann man auch die C-Funktion `atan2(z,x)` verwenden, die die Fallunterscheidung intern vornimmt.

Liegen die Daten bereits in sphärischen Koordinaten vor und sollen beispielsweise für andere Transformationen in kartesische umgewandelt werden, kann man folgenden Zusammenhang verwenden:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r \sin\theta \cos\varphi \\ r \sin\theta \sin\varphi \\ r \cos\theta \end{pmatrix} \quad \text{, mit } r = \sqrt{x^2 + y^2 + z^2} \tag{1.7}$$

Abschließend seien hier nochmal die üblichen Intervalle für die sphärischen Parameter zusammengefaßt *(alle Winkel im Bogenmaß)* :

$$\theta \in [\, 0, \ldots, \pi \,] \qquad \text{bzw.} \qquad \varphi \in [\, 0, \ldots, 2\pi \,[\;^{2} \tag{1.8}$$

1.1.3 Die Basisfunktionen der Spherical Harmonics

Ein weiterer wichtiger Begriff zum Verständnis der Sphärisch-Harmonischen Funktionen ist der einer **Basisfunktion**. Basisfunktionen sind eine Menge einfach beschreibbarer und kompakter Funktionen $B_i(x)$, die durch zugehörige Koeffizienten c_i skaliert werden und in ihrer Summe eine beliebige andere Funktion approximieren. Um die entsprechenden Koeffizienten bestimmen zu können, muss für jede einzelne Basisfunktion $B_i(x)$ die „Ähnlichkeit" zu der anzunähernden Funktion $f(x)$ berechnet werden. Dies wird durch das innere Produkt von Funktion und Basisfunktionen über dem Definitionsbereich der Ursprungsfunktion erreicht, d.h.:

$$c_i = \int f(x) B_i(x) dx \quad \text{, mit } \; i \in \mathbb{N} \tag{1.9}$$

Als Basisfunktionen kommen prinzipiell sehr viele Funktionen in Frage, wobei man sich im Wesentlichen auf orthonormale Funktionen beschränkt, da sich dadurch weitere Berechnungsschritte erheblich vereinfachen:

$$\int F_m(x) F_n(x) dx = \begin{cases} 1 & \text{für } m = n \\ 0 & \text{sonst} \end{cases} \tag{1.10}$$

Beispiele für solche Basisfunktionen lassen sich viele finden: von den einfachen trigonometrischen Funktionen bei der Signalverarbeitung über die in der Computergrafik verwendeten Bernsteinpolynome bis hin zu den Basisfunktionen der universell einsetzbaren Wavelets.

Insbesondere für die Spherical Harmonics kommen jedoch andere Basisfunktionen zum Einsatz. Diese sollen im Folgenden etwas detaillierter

[2] Je nach Implementation kann `atan2(z,x)` auch Werte aus $[\, -\pi, \ldots, \pi \,[$ liefern, die jedoch durch einfaches Addieren von 2π auf die negativen Werte „korrigierbar" sind.

betrachtet werden, da die entsprechenden Definitionen für die spätere Implementation benötigt werden. So kann man analog zu den Ausführungen in Abschnitt 1.1.1 jede quadratisch-integrierbare, auf der Kugel definierte Funktion schreiben als:

$$f(\theta,\varphi) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} C_{\ell,m} Y_{\ell,m}(\theta,\varphi) \quad \text{, mit } \ell \in \mathbb{N},\ m \in \mathbb{Z} \tag{1.11}$$

wobei die Koeffizienten $C_{\ell,m}$ definiert sind durch:

$$C_{\ell,m} = \int_0^{2\pi} \int_0^{\pi} f(\theta,\varphi)\, Y_{\ell,m}(\theta,\varphi)\, d\theta\, d\varphi \tag{1.12}$$

Obwohl die Spherical Harmonics ebenso wie die Fourierreihe von (1.1) im Komplexen beschrieben sind, ist aufgrund der Verwendung von realen Werten in der Computergrafik folgende Definition ausreichend:

$$Y_{\ell,m}(\theta,\varphi) = \begin{cases} \sqrt{2}\, K_{\ell,m}\, \cos(m\varphi)\, P_{\ell,m}\,(\cos\theta) & \text{für } m > 0 \\ \sqrt{2}\, K_{\ell,m}\, \sin(-m\varphi)\, P_{\ell,-m}\,(\cos\theta) & \text{für } m < 0 \\ K_{\ell,0}\, P_{\ell,0}\,(\cos\theta) & \text{für } m = 0 \end{cases} \tag{1.13}$$

wobei $K_{\ell,m}$ einen Normalisierungsfaktor für die Funktionen darstellt:

$$K_{\ell,m} = \sqrt{\frac{(2\ell+1)(\ell-|m|)!}{4\pi(\ell+|m|)!}} \tag{1.14}$$

Es fehlen an dieser Stelle noch die Basisfunktionen selbst, die bisher durch den Term $P_{\ell,m}$ bezeichnet wurden. Diese Polynome basieren auf Theorien aus der Quantenmechanik und finden dort als sogenannte **Legendre-Polynome** wegen ihrer Orthonormalität (siehe Formel (1.10)) und anderen Eigenschaften vielfach Verwendung. Da diese jedoch wiederum komplexe Werte liefern, verwendet man hier die **zugeordneten Legendre-Polynome** (*engl.:* associated Legendre polynomials).
Diese Polynome werden dabei durch die beiden schon häufig verwendeten Parameter ℓ und m indiziert und liefern als Funktionswerte entsprechend von ℓ und m abhängige reelle Zahlen. Man bezeichnet ℓ dabei auch als **Band(-index)**, den Parameter m als **magnetische Quantenzahl**, was unmittelbar auf die Herkunft dieser Polynome hindeutet. In der folgenden Abbildung 1 sind zur Veranschaulichung einige der ersten zugeordneten Legendre-Polynome als Graphen gezeichnet.

Entgegen der ursprünglichen quantenphysikalischen Verwendung wird für Berechnungen im Zusammenhang mit den Spherical Harmonics die magnetische Quantenzahl m jedoch etwas anders definiert. Während beide

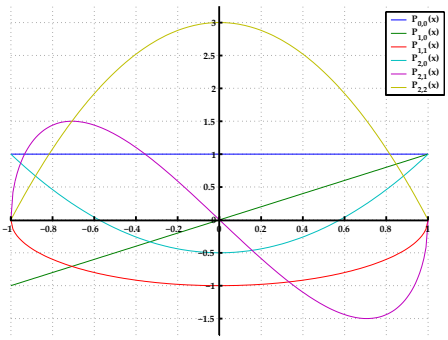

Abbildung 1: Die ersten sechs zugeordneten Legendre-Polynome.

Parameter auch in Abbildung 1 in der bisherigen Form ($0 \leq m \leq \ell$) verwendet wurden, ergibt sich für m als neuer Definitionsbereich:

$$0 \leq \ell < +\infty \qquad \text{sowie} \qquad -\ell \leq m \leq \ell \qquad \forall\, \ell, m \tag{1.15}$$

Die Herkunft der Polynome zeigt sich auch an der für reellwertige Funktionen ungewöhnlichen Eigenschaft, dass die Parameter ganzzahlige Werte sind, wobei ℓ das Band spezifiziert, in dem es $2\,\ell + 1$ verschiedene Varianten gibt[3]; dieser Zusammenhang ist auch aus der Reihenschreibweise der Spherical Harmonics in (1.11) ersichtlich.

Da es im Rahmen der vorliegenden Arbeit nicht möglich ist, sich mit dem mathematisch recht anspruchsvollen Prozess der direkten Berechnung der Legendre-Polynome zu beschäftigen, sollen hier nur die **rekursiven Definitionen** der Basisfunktionen genannt werden, die auch in der Implementation Verwendung finden. Die Bestimmung der Polynomwerte per Rekursion ist numerisch von Vorteil, da Fließkommawerte nur eine begrenzte Genauigkeit haben und der Berechnungsaufwand proportional zur Rekursionstiefe nach oben abschätzbar bleibt (siehe dazu [PTVF02]). Somit ergeben sich folgende Definitionen für $P_{\ell,m}$:

$$P_{\ell,m}(x) = \frac{x(2\ell-1)P_{\ell-1,m} - (\ell+m-1)P_{\ell-2,m}}{\ell-m} \tag{1.16}$$

$$P_{m,m}(x) = (-1)^m\,(2m-1)!!\,\sqrt{(1-x^2)^m} \tag{1.17}$$

$$P_{m+1,m}(x) = x(2m+1)P_{m,m} \quad , \qquad P_{0,0}(x) = 1 \tag{1.18}$$

$$\text{(mit } k!!{=}n!\,, n \in N,\ N := \{n \in \mathbb{N} | \exists i \in \mathbb{N}_0 : n = (2i+1)\}\text{)}$$

[3] In [Bie03] wird der Parameter ℓ entgegen der allgemeinen Notation als $0 < \ell < \infty$ verwendet. Da dieser Unterschied das Verständnis und die Vergleichbarkeit mit anderer Literatur erschwert, wird hier die übliche Definition $0 \leq \ell < \infty$ verwendet.

1.2 Besondere Eigenschaften

Ausgehend von den umfangreichen Betrachtungen der vorigen Abschnitte ist es durchaus sinnvoll, die wichtigsten Eigenschaften der Spherical Harmonics zusammenzufassen bzw. neu vorzustellen. Es ist nämlich kein Zufall, dass in den letzten Jahren vor allem diese Funktionen für die Approximation von Beleuchtungsverhältnissen herangezogen wurden, obwohl neueste Entwicklungen auch schon Alternativen untersuchen. Als besondere Eigenschaften sollen in diesem Abschnitt erläutert werden:

- **Approximationslinearität** — durch Hinzufügen weiterer Koeffizienten wird das Ergebnis exakter
- **Analogie zum Tiefpassfilter** — eine diffuse Oberfläche wirkt wie ein Tiefpassfilter auf die Leuchtdichteverteilung der Umgebung
- **Rotationsinvarianz** — eine Rotation der SH-Koeffizienten ist äquivalent zu einer Rotation der Umgebung
- **Inneres Produkt statt doppelter Integration** — die Integration über die Hemisphäre kann durch ein Skalarprodukt ersetzt werden

Um sich aber vor der detaillierten Betrachtung der Eigenschaften zunächst einmal eine bessere Vorstellung von den Spherical Harmonics machen zu können, sind in Abbildung 2 die ersten fünf Bänder grafisch dargestellt. Die Funktionswerte werden in den größeren Modellen durch den Abstand zum Ursprung, auf den Einheitskugeln anhand ihrer Richtung repräsentiert; grüne Bereiche entsprechend dabei positiven und rote Bereiche negativen Werten.

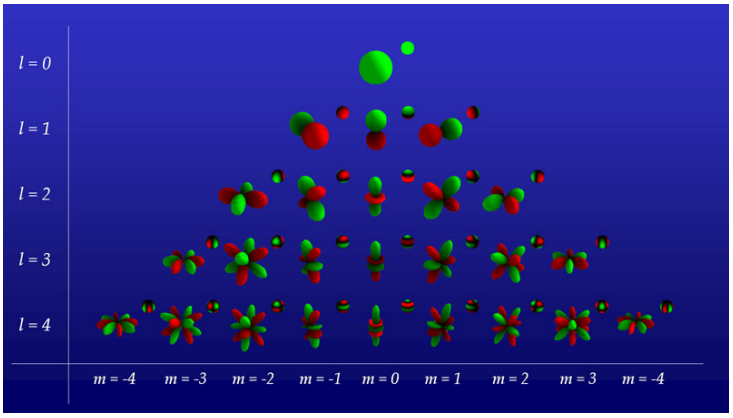

Abbildung 2: Die ersten 25 Spherical Harmonics $Y_{l,m}$. (Abb. überarbeitet aus [Gre03])

1.2.1 Linearität der Approximation

Zunächst sollte man sich bewußt machen, dass durch die Diskretisierung der analogen Funktionen wie Helligkeitsverteilungen, Raumwinkel, Positionen usw. keine beliebig exakte Darstellung erreicht werden kann. Zudem beschränken die Datentypen bei der digitalen Repräsentation der eigentlich kontinuierlichen Werte im Computer weiter die Genauigkeit. Eine „unendlich“ genaue Berechnung ist durch Zeit- und/oder Speichergrenzen letzlich ohnehin nicht möglich.
Diesem, zugegeben etwas ernüchternden Bild steht jedoch zum einen die Begrenztheit der menschlichen Wahrnehmung gegenüber: von der räumlichen Auflösung des Sehens bis hin zur Farb- und Helligkeitswahrnehmung sind auch der Natur „Grenzen gesetzt“. Zudem liegt die Leuchtdichtefunktion schon in Form von diskretisierten Werten (Pixel im HDR-Bild) vor, so dass durchaus mit einer endlichen Repräsentation eine ausreichend genaue Darstellung erreicht werden kann.
Eine solche Approximation ist vor allem dann gut geeignet, Werte angenähert darzustellen, wenn sie **kontinuierlich** ist. Die Stetigkeit von Funktionen wäre ein passender Vergleich, denn die Kontinuität garantiert, dass es keine undefinierten Stellen oder Sprünge in den Werten gibt, die den Eindruck vor allem im visuellen Bereich empfindlich stören könnten.
Diese Eigenschaft wird noch durch die Tatsache unterstrichen, dass das Hinzufügen weiterer Koeffizienten einer kontinuierlichen Erhöhung der Genauigkeit gleichkommt. Man spricht in diesem Fall auch von einer **linearen** Approximation, da die weiteren Terme entsprechend ihrer Reihenfolge fortlaufend ergänzt werden. Dabei ist jedoch zu beachten, dass nur die Koeffizienten in ihrer Gesamtheit zu einem korrekten Ergebnis führen, da durch die bessere Approximation durch mehr SH-Bänder die Koeffizienten der niedrigeren Frequenzen angepaßt werden. Andere Verfahren wie [NRH03], auf die hier nicht näher eingegangen werden kann, verwenden hingegen eine nicht-lineare Annäherung, indem sie beispielsweise nur die signifikantesten SH-Werte betrachten und weiterverarbeiten.

1.2.2 Tiefpassfilter-Analogie

Eine weitere Eigenschaft, die einen der Kernpunkte der vorliegenden Arbeit betrifft, ist wieder unmittelbar mit der Fourier-Transformation verwandt. Transformiert man ein Bild vom Orts- in den Frequenzraum, beschreiben eine (theoretisch unendliche) Menge von Koeffizienten das Bild. Diese können wie bei der besprochenen Fourierdarstellung komplexwertig sein, oder durch eine einfache Variation der Basisfunktionen (Sinus/Cosinustransformation) auch aus reellen Zahlen bestehen. Unabhängig von ihrer Definitionsmenge beschreiben die Koeffizienten in aufsteigender Reihenfolge immer höhere Frequenzen.

Diese Tatsache kann man sich aber auch zu Nutze machen, um die hohen Frequenzen im Bild zu unterdrücken: man berechnet die Koeffizienten nur bis zu einer Grenzfrequenz. Als Resultat erhält man ein **tiefpassgefiltertes** Bild, das nach der Rücktransformation in den Ortsraum auch entsprechende Merkmale aufweist (weichgezeichnete Kanten, Unschärfe etc.).
Für den hier betrachteten Fall der Beleuchtung virtueller Objekte mit diffusen Oberflächen haben vor allem Ramamoorthi et al. [RH01b], [RH01a] und [RH02]) festgestellt, dass nahezu die gesamte „Information" in Form von Strahlungsverteilung in den ersten Koeffizienten enthalten ist; der mittlere Fehler liegt bei der Verwendung von nur 9 Koeffizienten (entspricht einem Bandindex $\ell \leq 2$) im unteren einstelligen Prozentbereich.

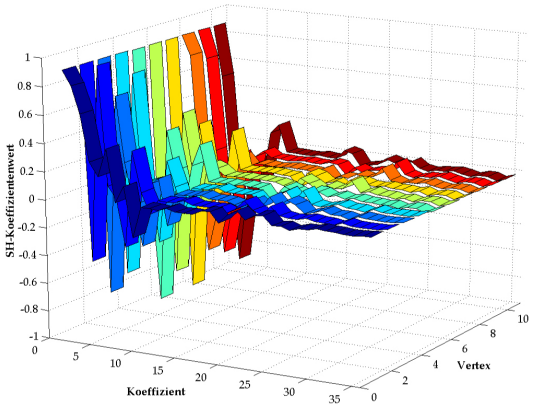

Abbildung 3: Die ersten 36 Koeffizientenwerte einer typischen Beispielszene *(10 Samples (Vertices), SH-Beleuchtung ohne Verschattung)*

Dies belegt auch die Grafik in Abb. 3, in der die ersten 36 Koeffizienten (Bandindex $\ell = 5$) einer Beispielszene aufgetragen sind. Hier ist deutlich erkennbar, wie stark die Beträge der Werte nach den ersten drei SH-Bändern abfallen — ein Indiz dafür, dass nahezu die gesamte einfallende Strahlungsintensität in den ersten Spherical Harmonics Koeffizienten enthalten ist.
Als Ergebnis lässt sich also festhalten, dass es für die Beleuchtung von diffusen Objekten unter den gleichen Bedingungen wie beim Environment Mapping (Abstand von Licht zu Objekt sehr groß im Verhältnis zu den Objektmaßen) vollkommen ausreichend ist, die Helligkeitsverteilung der Umgebung durch eine Approximation der niedrigen Frequenzen (d.h. der ersten Bänder) zu realisieren. Hierdurch wird die Datenmenge ähnlich deutlich wie bei der Quantisierung der JPEG-Kompression reduziert, womit ein wesentlicher Schritt für die äußerst effiziente Repräsentation von Leuchtdichteverteilungen gemacht ist.

1.2.3 Die Rotationsinvarianz

Um den Vergleich zur Quantisierung der DCT-Koeffizienten[4] beim JPEG-Verfahren noch einmal aufzugreifen, betrachte man die Artefakte einer zu starken Kompression: Es bilden sich u.a. Störungen vor allem in kontrastreichen Bildregionen sowie „blocky structures“, also regelmäßige blockartige Strukturen. Würde man diese Cosinus-Transformation durch die Fouriertransformation ersetzen, könnte man von der oben schon erwähnten Verschiebungsinvarianz der Fouriertransformation profitieren und die Artefakte vermeiden. (Da die Fouriertransformation jedoch auf komplexen Werten beruht und daher die doppelte Datenmenge anfällt, würde das Ziel eines reduzierten Speicherbedarfs bei der Bildkompression per DCT wieder zunichte gemacht; daher findet die Fouriertransformation in diesem Bereich keine praktische Verwendung.) „Invarianz“ bedeutet in diesem Zusammenhang, dass eine Verschiebung des Bildes sich bei Beachtung der Periodizitätsforderung nicht auf die visuelle Repräsentation der Fouriertransformierten auswirkt — ein Grund, warum in der Bilderkennung verschobene Bilder nicht neu positioniert werden müssen.
Eine ganz ähnliche Eigenschaft besitzen auch die Spherical Harmonics: hier liegt eine Invarianz gegenüber Rotationen vor. Das bedeutet, dass es egal ist, ob man die darzustellende Funktion erst rotiert und dann in SH-Koeffizienten transformiert, oder die Rotation erst *nach* der Umwandlung durchführt. In Formelschreibweise heißt das also für eine Funktion f_{Rot}, die die durch eine Rotation Rot transformierte Funktion f darstellt:[5]

$$\widehat{f}_{Rot}(s) = \widehat{f}(Rot(s)) \tag{1.19}$$

Diese auf den ersten Blick recht unscheinbare Eigenschaft ist aber ein weiterer Schlüssel zur Beschleunigung der grafischen Darstellung: dadurch ist es möglich, einmal in die Spherical Harmonics umgewandelte Helligkeitsverteilungen ohne weitere rechenaufwändige Hin- und Rücktransformationen zu rotieren.
Auf diesen etwas komplexeren Themenbereich soll im nächsten Abschnitt noch einmal näher eingangen werden, da es sich vor dem Hintergrund der programmiertechnischen Umsetzung anbietet, den Schwerpunkt mehr auf die praktische Verwendung als auf die zugrunde liegenden mathematischen Theorien zu legen.

[4] DCT = Diskrete Cosinus-Transformation; wird beim JPEG-Kompressionsverfahren verwendet und ist mit der (diskreten) Fouriertransformation verwandt.

[5] Mit $\widehat{f}(s))$ sei im Folgenden die durch die Spherical Harmonics-Koeffizienten approximierte Funktion $f(s)$ bezeichnet.

1.2.4 Vom Doppelintegral zum inneren Produkt

Die wichtigste Eigenschaft der Spherical Harmonics für die Verwendung in der photorealistischen Computergrafik ist die Tatsache, dass man die für die korrekte Beleuchtung eines Objektes benötigte Doppelintegration über die gesamte Hemisphäre auf ein einfaches inneres Produkt reduzieren kann. Möglich wird dies durch eine Transformation sowohl der **Beleuchtungsfunktion $L(s)$** [6], als auch der sogenannten **Transferfunktion $T(s)$**; letztere beschreibt, wie einfallendes Licht an der Oberfläche des Objektes reflektiert wird. Man kann an dieser Stelle auch eine komplette BRDF verwenden, so wie es Cabral et al. [CMS87] bereits umgesetzt haben.
Liegen nun beide Komponenten — also die Helligkeitsverteilungsfunktion und die Transferfunktion — in der Spherical Harmonics Repräsentation als Koeffizienten vor, folgt durch Ausnutzung der Orthogonalität der Spherical Harmonics, dass ein Aufsummieren der multiplizierten Komponenten ausreicht. Diese Eigenschaft steht auch wieder in direkter Analogie zur Fouriertransformation.
Der weiter oben schon erwähnten Faltung eines (Bild-)signals im Ortsraum beispielsweise mit Filtern entspricht die Multiplikation beider Komponenten im Frequenzraum, weshalb die Anwendung großer Filter auf das Bild mit Hilfe der Fouriertransformation wesentlich beschleunigt wird; schließlich reduziert sich die Integration über den Filterkern für jeden Bildpunkt auf ein inneres Produkt. Für den hier betrachteten Fall, also der Erweiterung der über $\mathcal{S}^1$ (Kreis) definierten Fouriertransformation auf die über $\mathcal{S}^2$ (Kugel) definierten Spherical Harmonics, gilt dieser Zusammenhang gleichermaßen. Zudem ist die für die Beleuchtung nötige Integration über den vorderen Halbraum (der oberen Hemisphäre) geradezu prädestiniert für die Transformation in den Frequenzraum, da die Integration in Form des auf den positiven Wertebereich beschränkten Cosinus-Operator aus dem Lambert'schen Beleuchtungsmodell im Sinne der Bildverarbeitung einem sehr großen Filter entspricht.
Abschließend sei der Zusammenhang zwischen dem (Doppel-)Integral und dem inneren Produkt in mathematischer Notation wie folgt beschrieben:

$$\int L(s)T(s)ds \quad \Longrightarrow \quad \int \widehat{L}(s)\widehat{T}(s)ds \approx \sum_{i=0}^{n^2} L_i T_i \tag{1.20}$$

Dadurch ist es möglich, die Darstellung beleuchteter Objekte von den Beleuchtungsverhältnissen zu trennen und die Komponenten vorab zu berechnen, so dass sie dann sogar „on the fly", also während des Renderns

[6] Der Parameter s steht hier vereinfachend für einen Punkt auf der Kugel S. Bei der konkreten Umsetzung würden hier sphärische Koordinaten verwendet werden, die auch die zweifache Integration begründen.

zusammengefügt werden kann. Wie im weiteren Verlauf der Arbeit noch deutlich wird, schränkt eine solche Vorberechnung die Anwendung des Verfahrens insbesondere bei dynamischen Szenen ein; eine komplette Integration pro Frame, die die herkömmliche Technik erfordern würde, wäre jedoch mit heutiger Hardware immer noch weit von interaktiven Geschwindigkeiten entfernt...

1.3 Die Verarbeitung der Spherical Harmonics

Diese Zusammenhänge bilden die Grundlage für den Einsatz in der Beispielimplementation, in der — neben anderen Techniken — auch die Spherical Harmonics verwendet werden. Zuvor bedarf es aber noch einer Konkretisierung der Theorie, damit die eingesetzten Algorithmen und Verfahrensweisen nachvollziehbar werden. Dazu ist zunächst ein Überblick über die verschiedenen Schritte zur Verarbeitung der Spherical Harmonics hilfreich, der im Gegensatz zu den vorigen Abschnitten mehr auf die praktische Umsetzung abzielt.

Die Verarbeitung von Spherical Harmonics basierter Beleuchtung unterteilt sich im Allgemeinen in einen Vorverarbeitungsprozess (*engl.:* offline (pre-)process), bei dem die zumeist aufwändigen oder einmaligen Berechnungen durchgeführt werden, und in einen Verarbeitungsteil, der zur Laufzeit (*engl.:* online process) berechnet wird. Zunächst sollen die grundlegenden Konzepte des Vorprozesses erläutert werden.

1.3.1 Die Vorverarbeitungsschritte

Während in der folgenden Aufzählung die Bestimmung der Basiskoeffizienten *vor* den anderen Arbeitsschritten und nur einmal durchgeführt werden muss, können die weiteren Berechnungen in beliebiger Reihenfolge und Häufigkeit geschehen; insbesondere die Transferfunktion $T(s)$ und die Beleuchtungsfunktion $L(s)$ sind unabhängig voneinander:

1. **Erstellung der Basiskoeffizienten**
   Zunächst müssen die Koeffizienten der SH-Basis berechnet werden, über der die zu approximierenden Funktionen ($L(s)$ und $T(s)$) integriert werden.

   (a) **Samplen der Umgebung**
   Hierbei wird mit Hilfe von Zufallszahlen eine ausreichende Anzahl (quasi-)gleichverteilter Positionen auf der Einheitskugel bestimmt und deren sphärische Koordinaten gespeichert.

(b) **Berechnen der Basiskoeffizienten für k SH-Bänder**
Anhand der gespeicherten Position jedes einzelnen Samples werden mit Hilfe der Formel 1.13 (Seite 9) die ersten k^2 Koeffizienten berechnet.

(c) **Speichern der Koeffizienten**
Schließlich werden alle k^2 Koeffizienten in jedem Sample für die spätere Verwendung abgespeichert.

2. **Approximation der Beleuchtungsfunktion**
Bei diesem Schritt wird anhand der Basissamples eine Annäherung der Lichtfunktion bestimmt, die die einfallende Beleuchtung in kompakter Form repräsentiert.

   (a) **Farbwerte aus Umgebungsbeschreibung bestimmen**
   Hierbei werden die sphärischen Positionen der Samples verwendet, um in die Abbildung der Umgebung (Environment Map) zu indizieren und den dort enthaltenen RGB-Farbwert auszulesen.

   (b) **Integration der Lichtfunktion**
   Die gesammelten RGB-Daten werden über die vorher bestimmten SH-Basiskoeffizienten integriert (per innerem Produkt, Formel 1.11) und als *ein* Vektor von RGB-Koeffizienten gespeichert.

   (c) **Gewichtung der Koeffizienten**
   Da die Positionsbestimmung der N Samples auf einem stochastischem Prozess beruht und bezüglich der (Einheits-)Kugeloberfläche praktisch gleichverteilt ist, muss jeder Koeffizientenwert noch mit dem Gewicht $\frac{4\pi}{N}$ multipliziert werden.

3. **Bestimmung der SH-Koeffizienten für Transferfunktion**
Dieser Schritt, für den prinzipiell jede funktionale Beschreibung des Transfers von einfallender zu ausgehender Beleuchtung geeignet ist[7], ist aufgrund seiner Komplexität der aufwändigste. Je nach verwendetem Verfahren ist die Rechenzeit zusätzlich zur Zahl der Basissamples von der Anzahl der Objektsamples abhängig, im vorliegenden Fall also von der Anzahl der Vertices der Geometrie. Auf diesen Schritt wird bei der Beschreibung der Beispielimplementation im dritten Teil der Arbeit detaillierter eingegangen.

Wie im zweiten Teil der Arbeit noch näher untersucht wird, können manche Arbeitsschritte in vereinfachter oder modularisierter Form auch zu einem späteren Zeitpunkt, d.h. während des Renderns durchgeführt

[7] Beispielsweise können hier auf Radiosity basierende Transferfunktion wie in [SKS02] oder [Gre03] verwendet werden. Auch „höhere“ Transfers wie Subsurface Scattering (Grundlagen siehe z.B. [AMH02], praktische Umsetung v.a. in [Slo03]) oder komplette BRDF-/BSSRDF-Beschreibungen sind möglich.

werden. Insbesondere der letzte Vorgang, also die Bestimmung der (idealerweise hochwertigen) Transferfunktion bereitet aber aufgrund des hohen Berechnungsaufwandes große Probleme bei einer Umsetzung dynamischer Konzepte.

1.3.2 Die Schritte beim Rendern

Sowohl bei der Berücksichtigung dynamischer Kontexte als auch bei der herkömmlichen, statischen Verwendung der Spherical Harmonics werden die im vorigen Abschnitt berechneten Komponenten beim Rendern in jedem Frame zusammengefügt. Obwohl im Rahmen dieser Arbeit die SH-Beleuchtung nur für diffuse Oberflächen verwendet wird (für andere Möglichkeiten siehe beispielsweise [SKS02] oder [KSS02]), für die die reflektierte Beleuchtung gemäß dem Lambert'schen Gesetz blickwinkelunabhängig ist, wirkt sich eine Änderung der Objekt- oder Umgebungstransformation auf die Beleuchtung aus; lediglich für den trivialen Fall des ersten SH-Bandes ($\ell = 0$) bleibt auch dann die Beleuchtung konstant.
Verändert man in der Szene daher nicht nur die Betrachtungsperspektive (d.h. die Kameraposition), sondern auch die Rotation der Umgebung oder des Objektes, geht der ursprüngliche Zusammenhang, in dem die SH-Koeffizienten berechnet wurden, verloren. Eine naive Lösung wäre nun, die Koeffizienten eben für diese veränderte Objekt-Umgebungskonstellation neu zu berechnen; der hohe Berechnungsaufwand, der insbesondere für die Transferfunktion nötig ist, soll aber gerade mit den Spherical Harmonics vermieden werden...

Wie in Abschnitt 1.2.3 auf Seite 14 erläutert wurde, sind die Spherical Harmonics rotationsinvariant, d.h. die berechneten Koeffizienten*werte* ändern sich bei einer Rotation der Beleuchtung und/oder des zu beleuchtenden Objektes nicht. Doch wie „rotiert" man SH-Koeffizienten? Neben der damit verbundenen Frage, in welcher Form das System Rotationen verarbeitet (siehe hierzu [Gre03]), muss man letztlich auch die Koeffizientenwerte selbst „drehen". Hierzu gab es bis vor kurzem sehr eingeschränkte Verfahren, die nur für die ersten SH-Bänder ($\ell < 3$) sinnvoll einsetzbar waren und zudem noch größere trigonometrische und damit sehr rechenaufwändige Ausdrücke beinhalteten.
Erst vor wenigen Jahren haben die Forschungen von Ivanic et al. [IR96] sowie Choi et al. [CIGR99] — ursprünglich im Bereich der physikalischen Chemie — für Computerberechnungen wesentlich besser geeignete, *rekursive* Verfahren hervorgebracht. Diese verwenden die 3×3-Rotationsmatrix aus dem laufenden Programm und vermeiden dadurch viele Nachteile bisheriger Verfahren, wie z.B. die Umsetzung der Wigner-D-Funktionen mit der nicht unproblematischen ZYZ-Eulerrepräsentation von Drehwinkeln.

Dazu wird abhängig von der verwendeten Anzahl n von Koeffizienten eine $n \times n$-Matrix berechnet, die mit dem Koeffizientenvektor multipliziert wird und einen n-Vektor mit rotierten Werten liefert. Diese in der Computergrafik sehr übliche Operation ist aus Gründen der Linearität der Spherical Harmonics auch hier direkt möglich. Diese Rotationsmatrix R_{SH} lässt sich der Kategorie der dünn besetzten (*engl.:* sparse), genauer der blockweise-diagonal dünn besetzten Matrizen (*engl.:* block-diagonal sparse) zuordnen. Wie aus Abbildung 4 ersichtlich ist, befinden sich nur in quadratischen Blöcken entlang der Hauptdiagonalen Werte ungleich Null (hier: $\boldsymbol{x}$); es ist daher also mehr als wünschenswert, statt einer naiven Matrix-Vektor-Multiplikation mit allen n^2 Einträgen nur die relevanten Werte in die Berechnung mit einzubeziehen.

$$R_{SH} = \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \dots \\ 0 & 0 & 0 & 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \dots \\ 0 & 0 & 0 & 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \dots \\ 0 & 0 & 0 & 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \dots \\ 0 & 0 & 0 & 0 & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \boldsymbol{x} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Abbildung 4: Die allgemeine Struktur der SH-Rotationsmatrizen

Zudem ergibt sich aus der Orthogonalität der Spherical Harmonics, dass sich die Koeffizienten der verschiedenen Bänder (hier: Blöcke) gegenseitig nicht beeinflussen. Durch diese Eigenschaft ist es möglich, sowohl die Rotationsmatrix als auch den Koeffizientenvektor in die entsprechenden Bänder zu unterteilen, einzeln zu berechnen und bis zur gewünschten SH-Ordnung ℓ zusammenzufügen. Diese Rotation des Objektes lässt sich auf die inverse Rotation der Umgebung zurückführen und ist daher für jeden zu beleuchtenden Vertex gleich, was zu einem erheblich geringerem und zudem von der Anzahl der Vertices unabhängigen Aufwand führt; lediglich die Beleuchtungsberechnungen selbst steigen proportional zur Vertexzahl. Dieser rotierte Vektor mit den RGB-Koeffizienten der Lichtfunktion kann den evlt. verwendeten Shaderprogrammen als Liste von Konstanten zur Verfügung gestellt werden.[8]

[8] Eine genauere Beschreibung der Möglichkeiten von GPU-Berechnungen in Form von Shadern folgt in Abschnitt 2.1, ab Seite 21

Nach dieser ausführlichen Beschreibung der Rotation der Spherical Harmonics fasst folgende Aufzählung die beiden wesentlichen Schritte, deren Reihenfolge hier offensichtlich beibehalten werden muss, noch einmal zusammen:

1. **Rotation der SH-Koeffizienten**
   Eine mögliche Änderung der Objekt- und/oder Umgebungsrotation muss in jedem Frame ausgewertet und in die folgendenen Berechnungen mit einbezogen werden.

   (a) **Bestimmung der permutierten Rotationsmatrix**
   Wie oben schon beschrieben, muss aus den Rotationsmatrizen des Grafiksystems (hier: OpenSG/OpenGL) für das Objekt und die Umgebung eine kombinierte und umsortierte Matrix erzeugt werden.

   (b) **Erstellung der SH-Rotationsmatrix**
   Die permutierte 3×3-Matrix dient als Eingabe in das Berechnungssystem (siehe Anhang 1 von [Gre03] bzw. [IR96]) und resultiert in der oben erläuterten $n \times n$-Rotationsmatrix.

   (c) **Rotation des Lichtvektors**
   Obwohl eine Rotation der SH-Koeffizienten des Objektes wie in [Bie03] möglich ist, empfiehlt sich aufgrund des wesentlich geringeren und von der Objektgröße unabhängigen Datenaufkommens (ein n-Vektor je Farbkanal) eine Rotation des *Lichtvektors*; dazu wird für das Objekt die Inverse der Rotationsmatrix verwendet. Die Multiplikation des n-Vektors mit der $n \times n$-Rotationsmatrix ergibt schließlich den rotierten Vektor mit n RGB-Koeffizienten.

2. **Kombination von Licht- und Transferfunktion**
   Als letzter Schritt wird das innere Produkt der beiden, durch SH-Koeffizienten beschriebenen Funktionen $L(s)$ und $T(s)$ gebildet, d.h. für jeden Sample auf dem Objekt (hier: Vertex) werden die Produkte der (ggf. rotierten) Koeffizienten aufsummiert und ergeben den Farbwert dieses Samples. Diese Berechnung kann sowohl auf der CPU (siehe z.B. [Bie03]), als auch aus Gründen der höheren Effizienz auf der GPU wie in der vorliegenden Implementation durchgeführt werden.

Im dritten Teil der Arbeit wird auf einige weitere programmiertechnische Details eingegangen. Insbesondere die gemeinsame Verarbeitung der Spherical Harmonics auf der Grafikhardware und auf der CPU kann mit den technischen Grundlagen aus dem nächsten Kapitel im praktischen Teil angemessen erläutert werden.

2 Verschiedene Technologien

In diesem Abschnitt soll zunächst ein Grundverständnis für die eingesetzten Techniken geschaffen werden, das viele Details und Entscheidungen, die im Zusammenhang mit der Beispielimplementation getroffen wurden, transparenter macht. Dies umfasst neben einer kurzen Erläuterung wesentlicher Komponenten moderner Grafikhardware auch einen Überblick über Konzepte und Strukturen der verwendeten Grafikprogrammiersprache Cg. Darüber hinaus spielt für die vorliegende Arbeit der Einsatz von High Dynamic Range Bildern eine wichtige Rolle, so dass eine Untersuchung vorhandener Formate und Beschreibung der besonderen Verarbeitung dieser Bilddaten sinnvoll ist.

2.1 Programmierbare Grafikhardware

Um in echtzeitfähigen 3D-Anwendungen die benötigte Performance zu erreichen und dabei nach Möglichkeit nicht auf die visuelle Qualität ihrer nicht-interaktiven Pendants verzichten zu müssen, ist der Einsatz programmierbarer Grafikhardware heute nicht mehr wegzudenken. Neben der zunehmenden Programmierbarkeit moderner GPUs (*engl.:* Graphical Processing Units) ermöglicht auch die rasant steigende Komplexität ständig neue Geschwindigkeitsrekorde, wie aktuelle Computerspiele und Demos belegen.
Da sich die vorliegende Arbeit vor allem mit der realistischen Abbildung natürlicher Beleuchtung beschäftigt, ist es ebenso sinnvoll wie notwendig, das Potential der Grafikhardware auszunutzen. Natürlich kann aufgrund des Themenumfangs nicht jede noch so kleine Geschwindigkeitssteigerung ausgenutzt werden; eine ausreichende Beschleunigung wurde dennoch angestrebt. Zudem kann hier aus Platzgründen nur kurz auf die notwendigen Grundlagen und Konzepte eingegangen werden, weshalb auch hier auf einschlägige Literatur wie beispielsweise allgemein in [BB03], [AMH02] oder auch speziell für Cg in [FK03] verwiesen wird.

2.1.1 Allgemeine Grundlagen

Die Architektur von Grafikhardware unterscheidet sich in ihrer Struktur, ihrem Einsatzgebiet und dem zugrunde liegenden Paradigma deutlich von der aktueller CPUs. Zudem haben Grafikprozessoren mittlerweile eine wesentlich höhere Komplexität und Rechenleistung als „normale“ Computerchips erreicht — selbst das Moore'sche Gesetz wird noch durch die rund halbjährlich verdoppelte Grafikleistung weit übertroffen.
Während herkömmliche Prozessoren dafür konzipiert sind, allen möglichen Einsatzgebieten (*engl.:* general purpose) zu genügen, hat sich in der Com-

putergrafik schon seit langem das Streaming-Konzept in Form einer Rendering-Pipeline etabliert. Dies beschreibt zunächst, dass die Daten sequenziell durch mehrere Stationen laufen und dabei, vergleichbar einem Fließband, mehrere Daten zur gleichen Zeit, jedoch in verschiedenen „Fertigungsstadien“ in der Pipeline sind. Außerdem lassen sich in der Verarbeitung grafischer Informationen viele Arbeitsschritte parallelisieren, da zwischen den Daten (Vertices, Pixel etc.) keinerlei Abhängigkeiten bestehen. Diese Parallelisierung kann sowohl bezüglich einer Dateneinheit (z.B. vier Werte für eine RGBA-Farbe) als auch durch eine einfache Vervielfachung der Pipelines erreicht werden. Im ersten Fall spricht man vom sogenannten SIMD-Konzept (*engl.:* Single Instruction Multiple Data), bei dem ein Befehl auf alle Komponenten gleichzeitig angewendet werden kann[9]; bei der Parallelisierung „im Großen“ sind heutzutage bis zu 16 Pixel-Pipelines üblich.[10]

Die Platzierung in der Pipeline

Wie in der Einleitung schon erwähnt, haben sich die verschiedenen Generationen von Grafikkarten von fest vorgegebenen Funktionen (*engl.:* fixed-function pipeline) zu immer flexibler programmierbaren Prozessoren entwickelt. Da diese Flexibilität jedoch keineswegs mit der von CPUs vergleichbar ist, gibt es immer noch Beschränkungen und Regeln, die eine allgemeine Verwendung erschweren, aufgrund der nötigen Performance jedoch unabdingbar sind.
Diese Einschränkungen kann man auch sehr gut daran erkennen, dass die im Allgemeinen als *Shader* bezeichneten Programme nur Teile der Grafikpipeline übernehmen, jedoch nicht deren gesamte Funktionalität ersetzen. Zudem sind diese programmierbaren Bereiche der Grafikverarbeitung optional, d.h. der Programmierer kann bei Bedarf auch auf die herkömmliche (fixed-function-)Pipeline zurückgreifen. Weiterhin gilt für Shaderprogramme, die pro Vertex bzw. Fragment ausgeführt werden, dass sie jeweils nur ein Datum verarbeiten können, das heißt weder Nachbarvertices/-fragments zur Verfügung stehen, noch entsprechende Elemente erzeugt oder gelöscht werden können.

Wie in Abbildung 5 erkennbar ist, übernehmen die Shader zwei Abschnitte in der Pipeline und bieten daher auch unterschiedliche Möglichkeiten. Auf die Einzelheiten soll in den folgenden Abschnitten kurz eingegangen werden, da diese Konzepte für das Verständnis der im dritten Teil der Arbeit realisierten Techniken nötig sind.

[9] Dieses Prinzip findet auch schon seit einiger Zeit in verschiedenen CPU-Architekturen (SSE, AltiVec etc.) Verwendung.
[10] siehe beispielsweise [NVI04]

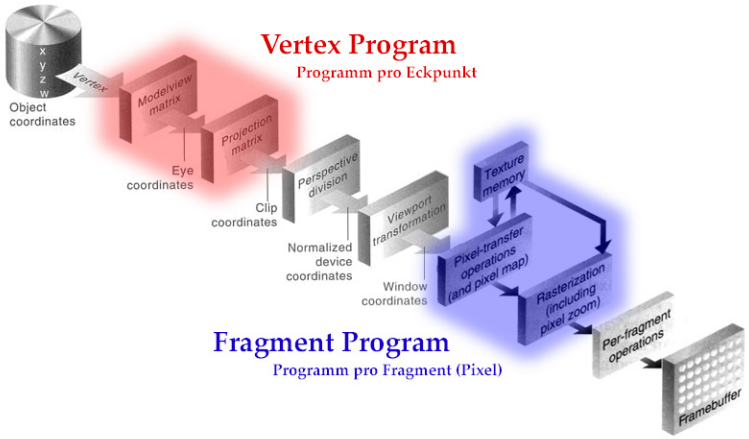

Abbildung 5: Schematische Darstellung der (optional) programmierbaren Komponenten in der Rendering-Pipeline von OpenGL. (Abb. überarbeitet aus [Ope00a])

Vertexprogramme

Im vorderen Abschnitt der Grafikpipeline werden die Eckpunktdaten der Geometrie in Form von Positionen, Normalen usw. transformiert und für die Beleuchtung ausgewertet. Man findet daher auch oft noch die Bezeichnung „Transform&Lighting“ (T&L), die eben diese Funktionalität widerspiegelt. Bei diesen Transformationen werden die Daten aus ihrem Objektkoordinatensystem zunächst in Weltkoordinaten abgebildet (*engl.:* modeling transform). Anschließend überführt die sogenannte „view transform“ diese in Kamerakoordinaten, da die Szene aus der Sicht einer Kamera, die wiederum selbst in Weltkoordinaten positioniert war, betrachtet werden soll. Da das Ziel die Beleuchtung der Szene anhand der Vertexdaten (v.a. Normalen) ist und dafür das Weltkoordinatensystem aus Effizienzgründen nicht sonderlich geeignet ist, werden diese beiden Transformationen meist zu einer „modelview transform“ zusammengefasst und danach die Berechnungen in Kamerakoordinaten ausgeführt.
Wie aus Abbildung 5 ebenso hervorgeht, umfassen Vertexprogramme jedoch auch die als „projection transform“ bezeichnete Umwandlung in die Clippingkoordinaten. Das Clipping selbst sowie die perspektivische Division sind dabei jedoch *nicht* mehr Bestandteil des Vertexshaders, sondern werden von der sich anschließenden fixed-function-Pipeline durchgeführt. Ebenso führt die Beschränkung der Farben auf den Wertebereich von $[0 \ldots 1]$ (*engl.:* color clamping) dazu, dass Werte, die dieser meist verlustbehafteten Umwandlung nicht unterzogen werden sollen (z.B. Texturkoordinaten, HDR-Farbwerte), getrennt übergeben werden müssen.

Fragmentprogramme

Während Vertexshader einen relativ großen Teil der Pipeline ersetzen, bilden Fragmentprogramme den unmittelbaren Übergang zum Bildspeicher (*engl.:* framebuffer), der das Bild schließlich darstellt. Lediglich einige optionale Tests, die über die Verwendung des Fragments entscheiden (Stenciltest, Tiefentest etc.), werden noch von der vorgegebenen Pipeline durchgeführt. Somit bleiben als Aufgabe von Fragmentshadern die Verarbeitung der (interpolierten) Farb- und Texturdaten aus den vorigen Stufen, sowie die sehr mächtige Fähigkeit, auf Texturen zugreifen zu können[11].
Darüber hinaus sind Fragmentprozessoren auch um ein Vielfaches leistungsfähiger ausgelegt als Vertexshader, da das hier zu verarbeitende Datenaufkommen wesentlich höher ist: ein bildschirmfüllendes Rechteck mit nur vier Eckpunkten kann Millionen von Fragments entsprechen. Diese Ressourcenverteilung kann man auch zum einen an den teilweise noch eingeschränkteren Möglichkeiten von Fragment- gegenüber Vertexprogrammen (z.B. Anzahl der Instruktionen) erkennen, zum anderen wird die oben angesprochende Parallelisierung in Form von mehreren Pipelines hauptsächlich für Fragmenteinheiten eingesetzt.

2.1.2 Die Grafikprogrammiersprache Cg

Hintergründe

Obwohl die Programmierung der Grafikhardware schon seit längerer Zeit möglich ist — in [FK03] wird dies der zweiten Generation (Ende 1999) zugeordnet, geht der Trend analog zur traditionellen CPU-Programmierung zu Hochsprachen. Zwar können Shaderprogramme immer noch in Assembler geschrieben werden und in die Grafikprozessoren geladen werden; bei den ständig wachsenden Möglichkeiten ist dies aber zunehmend schwieriger und fehleranfälliger.
Es existieren viele verschiedene Ansätze von solchen Hochsprachen für Echtzeitgrafik, die letztlich von existierenden Shadersprachen der Offline-Grafik wie Renderman oder PixelFlow abgeleitet wurden. Zu den wichtigsten Vertretern zählen dabei NVIDIAs Cg („C for graphics“), das syntaktisch mit der von Microsoft entwickelten HLSL („High Level Shading Language“) korrespondiert. Ein weiterer Standard, weniger von einzelnen Firmen abhängt, ist OpenGL 2.0, der auch die zughörige Shadersprache „GLslang“ spezifiziert; die Entwicklung und Umsetzung in Form von benutzbaren Bibliotheken dauert zur Zeit jedoch noch ebenso an wie die breite Verfügbarkeit entsprechender Hardware inklusive Treibern.

[11] Mit dem neuen Vertex-Shader 3.0 Standard, der jedoch noch nicht von jeder aktuellen Hardware und API unterstützt wird, sind auch Texturzugriffe in Vertexprogrammen möglich.

Wie bei der Erläuterung der Beispielimplementation detaillierter begründet wird, fiel die Entscheidung auf Cg, weshalb im Folgenden kurz auf dessen Verarbeitungskonzept eingegangen wird. Eine Erläuterung der Zusammenhänge anhand konkreter Programme und ihrer Eigenschaften bietet sich zudem auch erst in Teil III an, in dem auch näher auf die Schnittstelle zur Applikation eingegangen werden kann.

Das Konzept von Cg

Wie der Name schon vermuten lässt, ist Cg syntaktisch zu der Programmiersprache C verwandt, wobei neben einigen Anpassungen auch zahlreiche Anleihen aus C++ gemacht wurden. Außerdem sind aufgrund des speziellen Anwendungsgebietes viele Sprachelemente auf das oben erläuterte SIMD-Konzept bzw. auf die in der Computergrafik übliche Mathematik zugeschnitten. Neben der reinen Syntax wird für den praktischen Einsatz auch die konzeptionelle Verwendung des Programmes benötigt; hier bietet Cg zwei unterschiedliche Möglichkeiten.
Zum einen können Cg-Programme wie herkömmliche Programme kompiliert und in dieser Form in die Grafikhardware geladen werden. Flexibler und einfacher, wenn auch in einigen wenigen Fällen weniger performant, ist die zweite Möglichkeit: die Cg-Laufzeitumgebung (*engl.:* Cg-Runtime). Hierbei wird das Cg-Programm in der ursprünglichen Form mit Hilfe von API-Befehlen durch die Applikation direkt eingelesen, erst bei Bedarf kompiliert und der Hardware zur Verfügung gestellt. Der Vorteil ist dabei, dass Shaderprogramme *zur Laufzeit* der Applikation (auch von der Anwendung selbst) geändert werden und neu geladen werden können und somit Probleme durch eventuell dynamische Ressourcen (z.B. Registeradressen) vermieden werden.

2.2 Verarbeitung von HDR-Bildern

Wie in der Einleitung schon angemerkt wurde, ist die Verwendung von HDR-Bildern in der photorealistischen Computergrafik nicht mehr wegzudenken. Sogar aktuelle Grafikpakete, 3D-Applikationen und Renderingsysteme verarbeiten diese neuen Bildformate, um die visuelle Qualität deutlich zu verbessern. Während herkömmliche Bildinformation aus Gründen der (früheren) Verarbeitungsgeschwindigkeit in dem sehr eingeschränkten 8 Bit RGB-Format abgespeichert wird[12], gehen natürliche Beleuchtungsverhältnisse weit über diesen Tonwertumfang hinaus; das Ergebnis sind schwache, undetaillierte Hell-/Dunkelbereiche und ein insgesamt kontrastarmes und blasses Erscheinungsbild.

[12] Von höheren Bittiefen für spezielle Anwendungen wie in der Medizin oder der professionellen Fotografie sei hier der Einfachheit halber abgesehen.

Eine vollständige Einführung in die Erstellung und Verarbeitung dieser HDR-Bilder sowie ihren Besonderheiten gegenüber einfachen Bilddaten würde hier zu weit führen. Es soll sich daher auf die Beschreibung des verwendeten Formates beschränkt und dessen Verwendung in der Beispielanwendung vorbereitet werden.
Dem interessierten Leser seien daher für grundlegende Informationen zu High Dynamic Range Bildern das „Ur-Paper“ von Debevec [DM97], allgemeine Informationen [AMH02] oder Teile von [Bie03] empfohlen. Darüber hinaus finden sich mittlerweile auf vielen Internetseiten und in der Dokumentation der oben erwähnten 3D-Programme umfangreiche und detaillierte Informationen zu HDR-Bildern.

2.2.1 HDR-Bilder

Parametrisierungen

Wie auch bei herkömmlichen LDR-Bildern (*engl.:* Low Dynamic Range), gibt es — zunächst unabhängig von Dateiformaten — unterschiedliche Formen von HDR-Bildern. Diese Parametrisierungen zielen auf verschiedene Einsatzgebiete ab und haben ihre verschiedenen Vor- und Nachteile. Obwohl zweidimensionale, rechteckige Bilddaten die üblichste Form von Bildern darstellen, spielen sie hier nur eine untergeordnete Rolle. Da es bei der vorliegenden Anwendung um die bildbasierte Beleuchtung virtueller Objekte geht, repräsentieren die Bilder, die mit Panorama- oder Rundumbildern in der traditionellen Fotografie verglichen werden können, in erster Linie die *Umgebung*, die natürlich komplett in der Bildinformation enthalten sein sollte.
In der Computergrafik haben sich mittlerweile mehrere solcher Parametrisierungen für Umgebungstexturen (*engl.:* Environment Maps) etabliert:

- **Spherical Maps/Lightprobes** als eine Abbildung der Umgebung auf eine als spiegelnd angenommene Kugel (*engl.:* sphere)
- **Latlong Maps** als ein „abgewickeltes“´ zweidimensionales Bild dieser Kugel, das durch die Breiten- und Längengrade (*engl.:* latitude, longitude) beschrieben werden kann
- **Cubemaps** als eine Zusammenstellung sechs quadratischer Bilder in alle Raumrichtungen, die einen gedachten Würfel (*engl.:* cube) repräsentieren

Während die erste Form recht gut die gängige Art der Erstellung solcher HDR-Umgebungsbilder (nach Debevec et al.) widerspiegelt, sind aufgrund der einfacheres Verarbeitung die beiden letztgenannten Formate üblich. Insbesondere die Cubemaps haben große Vorteile bezüglich der Varianz

der Samplingrate und der Beschleunigung auf heutiger Grafikhardware; zudem bietet sich diese Parametrisierung auch für eine künstliche Erstellung dieser Umgebungstexturen innerhalb des Programms selbst an.
Aus diesen Gründen basiert die Verarbeitung der HDR-Bilder in der vorliegenden Implementation auf dieser Parametrisierung. Vielfach erhältliche und auch zahlreich eingesetzte HDR-Bilddaten liegen im Latlong-Format vor, weshalb diese beim Laden direkt umgewandelt und zur weiteren Verwendung abgespeichert werden.

Dateiformate

Da HDR-Bilder mittlerweile eine recht große Verbreitung haben, gibt es auch verschiedene Dateiformate, die die Bildinformation mit unterschiedlichen Zielsetzungen und Techniken abspeichern. In der folgenden Übersicht seien die üblichsten Formate zusammengefasst und kurz charakterisiert:

- **PFM** Portable Float Map
  - an PNM angelehnte Struktur (einfacher Header + Sequenz von `float`-RGB-Werten), sehr einfach zu verarbeiten
  - verlustfreie Repräsentation natürlicher Helligkeitswerte
  - sehr speicherintensiv, kaum komprimierbar
- **HDR** Radiance-Format (nach Greg Ward)
  - in einem 32 Bit-`float` sind die RGB-Werte (je 8 Bit) mit einem gemeinsamen Exponenten (8 Bit) abgespeichert
  - sehr kompaktes Format
  - beschränkter Tonwertumfang, v.a. bei starker Dominanz einer Farbe
- **TIFF** Tagged Image File Format
  - Erweiterung des herkömmlichen LDR-TIFFs in verschiedenen Varianten (32 Bit, 24 Bit, 16 Bit je RGB(A)-Kanal)
  - Verwendung unterschiedlicher Farbräume und Bitaufteilungen
- **EXR** OpenEXR (Industrial Light&Magic)
  - in Anlehnung an IEEE 754 `float` definiertes 16 Bit `half`-Format
  - kompatibel zu `half`-Datentyp von NVIDIA-Grafikhardware
  - sehr guter Kompromiss aus maximalem Tonwertumfang und Speicherbedarf (zusätzliche Kompression möglich)
  - sehr umfangreiche Bibliothek (open source, seit Herbst 2002)

Obwohl das EXR-Format noch keine vergleichbare Verbreitung gefunden hat wie insbesondere das HDR-Format, gilt es aufgrund seiner Flexibilität und seinen Vorteilen bei der Repräsentation von realen Helligkeitsdaten als Favorit für zukünftige HDR-Bildformate. Diese Vorteile und die umfangreiche, frei verfügbare Programmierbibliothek waren letztlich auch ausschlaggebend für die Entscheidung, dieses Format in der vorliegenden Implementation zu verwenden. Bis sich dieses Format weiter verbreitet hat und auch vielfach eingesetzte Werkzeuge wie „HDR Shop“ EXR-Dateien direkt unterstützen, kann man sich mit Konvertierungstools behelfen, wie dies für bereits vorhandene HDR-Bilder auch im Rahmen der Beispielanwendung nötig war.

Das OpenEXR Format

Da dieses Format, das von Industrial Light&Magic (ILM) im Herbst 2002 als Open-Source Bibliothek frei verfügbar gemacht wurde und ständig weiter entwickelt wird, noch recht neu ist, sollen hier ein paar Details zu den Eigenschaften und der Verwendung erläutert werden.
Wie weiter oben schon angemerkt wurde, ist das zentrale Datenformat für die interne Repräsentation der neu definierte `half`-Datentyp; neben diesem stehen zusätzlich `float` (32 Bit, nach IEEE 754) sowie `uint` als vorzeichenloser Integertyp (32 Bit, unsigned int) zur Verfügung. Diese können verwendet werden, wenn die Genauigkeit von `half` nicht ausreicht (z.B. bei Tiefenwerten) oder wenn einfache, ganzzahlige Angaben wie Objektkennungen abgespeichert werden sollen.
Dies wird von Seiten der API (*engl.:* Application Programming Interface) auf zwei Arten unterstützt:

- ein bereits vorgefertigtes Dateiformat, das aus den vier üblichen Kanälen (RGBA) im `half`-Format besteht und für die meisten Zwecke ausreichen sollte
- ein universelles Format mit der Möglichkeit, in den Dateien eigene Kanäle zu definieren, z.B. Luminanz und Transparenz (je `half`), Tiefenwerte (`float`) und Objekt-IDs (`uint`)

Zudem bietet OpenEXR mehrere Arten von Kompressionsalgorithmen an, die für unterschiedliche Anwendungsgebiete ausgelegt sind und übliche Bilder je nach Verfahren auf bis zu 35 Prozent der Originalgröße komprimieren können. Zur Verfügung stehen:

- **PIZ** verlustfrei, basiert auf der Wavelet-Transformation
- **ZIP** einfaches, verlustfreies Verfahren (zlib)
- **RLE** lauflängencodiertes, verlustfreies Verfahren
- **PXR24** verlustbehaftetes, ZIP-ähnliches Verfahren (auf 24 Bit)

Im Zusammenhang damit steht auch die interne Organisation der Dateien. Nach einem Header, der neben beliebigen vielen, eigenen Attributen eine Reihe vorgegebener Informationen enthalten muss (u.a. Auflösung, Datenfenster, Kanäle, Reihenfolge der Bildzeilen), folgen die eigentlichen Pixel. Diese enthalten je nach verwendeten Kanälen die eigentliche Bildinformation und sind entweder zeilenbasiert (*engl.:* scanline-based) oder kachelartig (*engl.:* tile-based) strukturiert und indizierbar. Außerdem kann das Bild mehrere Auflösungsstufen enthalten, die nach dem Mipmap- oder Ripmap-Verfahren angelegt werden können (siehe dazu [AMH02].

Von der Qualität der HDR-Eigenschaften ist OpenEXR ein optimaler Kompromiss zwischen dem repräsentierbaren Tonwertumfang und dem dafür benötigten Speicherplatz. Zwar können Formate wie PFM oder 32-Bit-TIFF die Farbwerte (theoretisch) noch exakter darstellen, benötigen aber für Bilder realistischer Größe (als Cubemap z.B. 2048x1024 Pixel) unnötig viel Platz. Zum Vergleich: ein RGB-PFM dieser Auflösung benötigt inklusive Header rund 24 MB, ein gleichdimensioniertes, unkomprimiertes EXR-Bild nur ungefähr die Hälfte.
Relevanter für bildbasierte Techniken, wie sie auch in dieser Arbeit eingesetzt werden, sind jedoch Kriterien wie Tonwertumfang und Farbauflösung. Auch hier bietet OpenEXR als ein neu entwickeltes Format sehr gut auf diese Bedürfnisse abgestimmte Eigenschaften. Neben dem praktisch unbegrenzten Tonwertumfang von $10^{12}:1$, der selbst für extreme natürliche Szenarien (direktes Sonnenlicht und Schatten) mehr als ausreicht, setzt auch die Farbauflösung mit 2^{10} Abstufungen pro Blendenwert keinerlei Grenzen. Selbst für Bilder mit normalem Werteumfang (z.B. acht Blendenstufen) bietet das `half`-Format mit 8192 Stufen pro Kanal ein Vielfaches gegenüber herkömmlichen 8-Bit-Bildern (hier: ca. 235 Stufen).

Weiterführende Informationen und Details zu Eigenschaften und Verwendung dieses Bildformates, Möglichkeiten zum Download sowie zahlreiche GPU-orientierte Anwendungsbeispiele mit Cg-Code finden sich neben der offiziellen Website [HK+02] auch in [KBH04].

2.2.2 Tonemapping-Verfahren

Wie weiter oben schon angemerkt, basieren viele Bildinformationen aus (mittlerweile überholten) Performancegründen auf der 8 Bit-Repräsentation. Zudem können die Ausgabegeräte für Bilddaten — im Allgemeinen also Monitore oder ähnliche Geräte wie Projektoren — nur einen geringen Tonwertumfang darstellen, der aus technischen Gründen eben genau den 8 Bit pro RGB-Kanal entspricht; HDR-Displays befinden sich jedoch bereits in der Entwicklung.

Bis dahin ist es nötig, die Bilddaten großen Tonwertumfangs, wie sie beispielsweise als Bild eingelesen oder im System in eine float-Textur gerendert werden, auf den Wertebereich des Ausgabegerätes (also 8 Bit) abzubilden. Dass eine einfache lineare Skalierung der HDR-Daten nur in den seltensten Fällen zu brauchbaren Ergebnissen führt, liegt auf der Hand. Für diese als *Tonemapping* bezeichnete Technik gibt es zahlreiche Verfahren, die sich hauptsächlich in ihrer Motivation, dem Berechnungsaufwand und den Resultaten unterscheiden; einen guten Überblick dazu findet man beispielsweise in [Dev02] sowie auch [Mül02].

Ein wichtiges Unterscheidungskriterium bei diesen Verfahren ist die *Domäne* des Algorithmus. Man unterscheidet dabei in Anlehnung an die Bildverarbeitung *räumlich* zwischen globalen und lokalen Operatoren und bezüglich ihrer Zeitabhängigkeit. Während globale Verfahren die gleiche Abbildungsfunktion (z.B. logarithmisch, inverse Kamerafunktion) auf das gesamte Bild anwenden und dadurch inhärent schneller sind, können lokale Operatoren auf Unterschiede in verschiedenen Bildteilen eingehen. Die *temporale* Domäne ermöglicht eine dynamische Anpassung der Parameter des Tonemappings bei sich ändernden Bildinhalten, wie sie beispielsweise in interaktiven Anwendungen vorkommen. Obwohl dies offensichtlich vom Berechnungsaufwand her viel aufwändiger ist, sind die Ergebnisse sehr überzeugend, da sie der menschlichen Wahrnehmung sehr viel näher kommen (können) als zeitunabhängige Verfahren. Gerade diese dynamischen Verfahren, die meist pro Frame angewendet werden müssen, lassen sich mit Hilfe neuester Grafikhardware beschleunigen und sogar in echtzeitfähigen Geschwindigkeiten realisieren. Da eine ausführliche Beschreibung an dieser Stelle zu weit führen würde, sei stellvertretend auf das Verfahren von Goodnight et al. [GWWH03] verwiesen, das unter Verwendung von Grafikhardware eine zeitabhängige Umsetzung hochwertiger Tonemapping-Algorithmen realisiert.

2.3 Schattenverfahren

Ein wesentlicher Bestandteil realistischer Computergrafik sind Schatten. Nur durch die variierende Helligkeit von Bildbereichen — die natürlich mit der Szene korrespondieren muss — ist es möglich, räumliche Konstellationen wie Abstände, Entfernungen sowie Beleuchtungsrichtungen und Materialeigenschaften überzeugend darzustellen. Aus diesem Grund befasst sich (nicht nur) die Computergrafik schon seit Jahrzehnten mit verschiedensten Verfahren zur Erzeugung realistischer Schatten, insbesondere mit echtzeitfähigen Ansätzen. Eine angemessene Erläuterung an dieser Stelle ist daher praktisch unmöglich, weshalb für detailliertere Informatio-

nen und Referenzen zu den vorgestellten und anderen Methoden unter anderem auf [AMH02] und [BB03] sowie [F+04] verwiesen wird.
Im Folgenden sollen kurz die zwei üblichsten Echtzeitverfahren zur hardwarebeschleunigten Berechnung von Schatten besprochen werden, die bereits seit verhältnismäßig langer Zeit in vielen Anwendungen zum Einsatz kommen. Da diese jedoch durch die Beschränkung auf punktförmige Lichtquellen bis auf wenige Ausnahmen relativ unrealistisch wirken, soll ein weiteres Verfahren, das auch in der Beispielimplementation verwendet wird, die Betrachtung abrunden.

2.3.1 Direkte Erzeugung von Schatten

Shadow Mapping

Dieses Verfahren zur Berechnung von Schatten geht bereits auf das Jahr 1978 zurück, indem Williams vorschlug, die Tiefeninformation aus Sicht der Lichtquelle zu verwenden, um die Objekte der Szene korrekt zu verschatten. Motiviert wurde das Verfahren auch aus der grundlegenden Eigenschaft von Schatten, *betrachterunabhängig* zu sein, da direkte Verschattungen von Lichtquellen nur von diesen bezüglich Größe und Position abhängig sind.
Shadow Mapping beruht auf einem zweistufigen Prozess, der in Form von zwei Renderdurchgängen realisiert wird:

1. **Rendern der Szene aus Sicht der Lichtquelle**
   Hierbei wird der zu beleuchtende Teil der Szene lediglich mit aktiviertem Tiefenpuffer gerendert, der die geringsten Entfernungen zwischen den jeweiligen Elemente und der Lichtquelle enthält. Diese Werte werden für den nächsten Schritt in einer Textur (*engl.:* shadow map) abgelegt.

2. **Zweites Rendern aus Kameraperspektive mit Vergleich der Tiefenwerte**
   In diesem Schritt wird die Szene aus der eigentlichen Kameraperspektive gerendert. Dabei wird vor dem Setzen der jeweiligen Pixel ein Vergleich mit dem zugehörigen, meist ins Kamerakoordinatensystem transformierten Tiefenwert an dieser Stelle in der Shadow Map durchgeführt. Ist der gespeicherte Wert kleiner als der Tiefenwert aus Sicht der Kamera, liegt dieser Punkt im Schatten.

Da sich diese Technik selbst mit früheren Grafikkartengenerationen leicht beschleunigen lässt, ist Shadow Mapping schon seit langem ein beliebtes Verfahren für die Erzeugung von Schatten in Echtzeit. Darüber hinaus ist ein weiterer Vorteil, dass das Rendern komplexer Geometrie von deutlich

geringerem Aufwand ist, als die Berechnungen für die im nächsten Abschnitt vorgestellten Shadow Volumes und somit weniger abhängig von der Zahl der Eckpunkte ist.

Leider hat diese Technik aber auch bisweilen erhebliche Nachteile, die über die prinzipielle Beschränkung auf infinitesimal kleine Lichtquellen hinaus gehen. Zum einen führt die Erzeugung der Shadow Maps zu einer, aus der Perspektive des Lichtes gleichförmigen Diskretisierung (*engl.:* aliasing) der abgebildeten Szene in Form von Pixeln (bzw. Texeln). Die Position bzw. Sicht der Kamera ist mit hoher Wahrscheinlichkeit aber eine andere als die der Lichtquelle, weshalb die Diskretisierung nicht mehr uniform ist — die (quadratischen) Texel werden verzerrt dargestellt. Problematisch ist dabei vor allem die starke Vergrößerung der Texturpixel, wenn die Kamera zunehmend in die entgegengesetzte Richtung der Lichtquelle blickt.
Zwar ist generell eine Abhilfe durch ausreichend hohe Texturauflösungen möglich, spätestens bei Verwendung mehrerer Lichtquellen aber aufgrund des Datenaufkommens nicht mehr praktikabel. Auch Perspective Shadow Maps, die die Aliasingeffekte der Shadow Maps mit Hilfe des Koordinatensystems *nach* der Projektion wesentlich vermindern können, bereiten bei gewissen Konstellationen (z.B. Lichtquelle hinter Kamera) Probleme. Alternativ dazu versuchen bildbasierte Methoden die Aliasingstrukturen durch Filter zu verbessern und können dabei durch den intensiven Gebrauch des Fragmentprozessors gute Ergebnisse bei realistischer Performance erreichen.
Ganz abgesehen von schwierigen Situationen bezüglich Kamera- und Lichtposition stellt sich auch die generelle Frage, wie man eine durch ein omnidirektionales Licht beleuchtete Szene mit dieser Technik verschatten soll — denn hierfür ist kein sinnvoller Öffnungswinkel zum Generieren der Schattentextur mehr definierbar. Zwar hätte ein sechsfaches Rendern in „Shadow Cubemaps“ einen deutlichen Einfluss auf die Gesamtperformance, was aber mit Hilfe entsprechender Optimierungen und dem Einsatz modernster Hardware auch kein fundamentales Problem mehr darstellt.
Mit Hilfe der neuesten Grafikhardware lässt sich zudem ein weiteres Problem der Shadow Maps ohne großen Aufwand beseitigen: die begrenzte numerische Auflösung der Tiefenwerte. Diese, als streifen- oder fleckartigen Artefakte erkennbaren Ungenauigkeiten sind auf die auf Integer- bzw. Festkommawerte begrenzten Tiefenpuffer/-texturen zurückzuführen. Hier können moderne Fließkommaformate mit 32 Bit Genauigkeit sehr effektiv Abhilfe schaffen und finden auch bereits in einigen der oben genannten weiterführenden Schattenverfahren Verwendung.

Shadow Volumes

Ein grundsätzlich anderes Konzept zur Erzeugung von direkten Schatten auf Basis von punktförmigen Lichtquellen sind die Shadow Volumes, die auf Crow (1977) zurückgehen.[13] Es basiert auf der Idee, die Sichtstrahlen des Betrachters einer Szene durch Schatten*volumen* zu verfolgen und die dabei passierten Verschattungsbereiche auszuwerten. Im Wesentlichen sind dazu die (volumetrischen) Schatten in Form von Geometrie nötig, damit die Auswertung des Strahls auf einfache Weise möglich ist.
Obwohl eine genaue Besprechung und Implementierung des Verfahrens den Umfang dieser Arbeit bei weitem übersteigen würde, bietet es dennoch Vorteile gegenüber dem Shadow Mapping. Die heute gängige Variante des ursprünglichen Verfahrens verwendet für die Verwaltung der passierten Bereiche den Stencilbuffer der Grafikpipeline, der mit einer Maske für den Bildspeicher verglichen werden kann. Zunächst soll aber auch hier der Algorithmus kurz skizziert werden:

1. **Löschen des Stencilbuffers**

2. **Rendern der Szene nur mit ambientem und emissivem Term**
   Dadurch wird sichergestellt, dass sowohl der Framebuffer als auch der Depthbuffer mit diesen unschattierten[14] Beleuchtungsinformationen gefüllt ist.

3. **Zeichnen der Polygon*vorder*seiten zum Inkrementieren der Stencilwerte**
   Dabei ist sicherzustellen, dass auf den Tiefenpuffer nur lesend zugeriffen wird (bei aktivierter Vergleichsfunktion) und der Framebuffer deaktiviert ist. Die Funktion erhöht nun den Stencilbuffer um 1 an den Stellen, wo ein Polygon gezeichnet wird.

4. **Zeichnen der Polygon*rück*seiten zum Dekrementieren der Stencilwerte**
   Analog zu Schritt 3, jedoch wird hier der aktuelle Wert im Stencilbuffer um 1 reduziert.

5. **Rendern der Szene mit diffusem und spekularem Anteil in Bereiche, an denen der Stencilwert Null ist**
   Diesem letzten Schritt liegt die Tatsache zugrunde, dass ein Stencilwert von 0 einem Sichtstrahl entspricht, der Schattenvolumen ebenso oft betreten wie verlassen hat — und somit unverschattet ist.

[13] Der Leser sei für die detaillierten Referenzen zu den vorgestellten Schattenverfahren hier nochmals auf [AMH02] sowie [F+04] verwiesen.

[14] Hiermit ist nicht „verschattet“ (*engl.:* shadowed) gemeint, sondern die Variation der Objektschattierung durch Beleuchtungseinflüsse (*engl.:* shaded).

Auch hier soll eine kurze Zusammenfassung der wichtigsten Vor- und Nachteile dieses Schattenverfahrens die Betrachtung abrunden.
Obwohl die heute übliche Form des Shadow Volume-Algorithmus nicht sehr einfach ist, können dennoch ältere Grafiksysteme, die zumindest einen 8 Bit-Stencilbuffer bieten, ein solches Verfahren beschleunigen. Ein weiterer, wesentlicher Vorteil gegenüber dem oben vorgestelltem Shadow Mapping ist, dass Shadow Volumes nicht bildbasiert sind und daher auch nicht unter den Problemen durch Diskretisierung zu leiden haben. Die erzeugte Schattengeometrie ist unabhängig von dem Zusammenspiel Betrachter- und Lichtposition immer exakt scharf und zeigt keinerlei Aliasingstrukturen. Dies kann aber wiederum auch zum Nachteil werden, wenn zu kleine Geometrie Schatten wirft und entsprechend dünne Shadow Volumes erzeugt werden; diese können ihrerseits durch Rundungsfehler Artefakte erzeugen.
Viel problematischer ist aber die Erzeugung und Darstellung dieser Volumengeometrie. Um diese Polygonflächen, die den Schattenbereich definieren, zu generieren, muss von dem Schatten werfenden Objekt (*engl.:* occluder) zunächst die Silhoutte aus Sicht der Lichtquelle bestimmt werden. Dazu sind Nachbarschaftsinformationen zwischen den Primitiven (Dreiecke) nötig, um die entsprechenden Kanten effizient zu finden. Dies führt zum einen zu wesentlich höheren Anforderungen an die Applikation bezüglich der Datenstrukturen und -haltung, zum anderen ist die Bestimmung der Kanten immer noch sehr zeitaufwändig und zudem unmittelbar von der Anzahl der Dreiecke des Objektes abhängig.
Ein weiterer deutlicher Nachteil des Verfahrens findet sich sozusagen „am anderen Ende der Pipeline“: die Rasterisierung. Da Schattenvolumen realistischer Geometrien, wie z.B. Charaktere oder hochaufgelöster Modelle meist sehr komplexe Überlagerungsstrukturen bilden, müssen viele Pixel beim Rendern der Szene mehrmals gezeichnet und verbrauchen dadurch eine hohes Maß an zur Verfügung stehender Pixelfüllrate — selbst bei moderner Grafikhardware mit dreistelligen Millionenwerten pro Sekunde ein nicht zu verachtendes Problem...

2.3.2 Ambient Occlusion

Während die vorangegangenen Schattenverfahren komplett zur Laufzeit angewendet werden und dadurch inhärent auch für dynamische Umgebungen, wie sie in Teil II dieser Arbeit betrachtet werden, geeignet sind, geht diese Technik einen anderen Weg. Dabei unterteilt sich das erstmals von Landis [Lan02] für die Beleuchtung durch Environment Maps eingesetzte Verfahren in einen vorberechneten Hauptanteil und eine — je nach Implementation — Verarbeitung zur Laufzeit. Da dies sehr analog zu der

Verarbeitung der Spherical Harmonics nach Sloan et al. [SKS02] ist, soll hier auch von einem „offline"- bzw. „online"-Prozess gesprochen werden. Natürlich geht aufgrund des Vorverabeitungsschrittes viel Flexibilität für dynamische Szenarien verloren, was aber im praktischen Einsatz weniger Probleme bereitet, als man vermuten könnte (siehe dazu [PG04]).
Wie der Name dieses Verfahrens schon erahnen lässt, ist die Vorberechnung *unabhängig* von der Beleuchtung, so wie der ambiente Term im Phong'schen Beleuchtungsmodell auch keinerlei Lichtquellen berücksichtigt. Man kann sogar bei der Ambient Occlusion von einer „intelligenteren" Hintergrundbeleuchtung sprechen, da sie nicht unrealistisch konstant ist, sondern geometrische Eigenschaften für eine Variation dieser Konstanten mit einbezieht. Dazu wird von jedem Objektsample (hier: Vertex) die Selbstverdeckung (*engl.:* occlusion) im vorderen Halbraum bezüglich der Samplenormale durch das Objekt selbst bestimmt und der resultierende Skalar abgespeichert; dabei wird die Umgebung des Objektes, d.h. andere Objekte, nicht berücksichtigt. Für die konkrete Bestimmung dieser geometrischen Verdeckung können mehrere Verfahren eingesetzt werden, die im Folgenden kurz aufgeführt werden:

1. **Raytracing-basiert**
   Hierbei wird eine genügend große Anzahl von Strahlen (Schattenfühler) von allen Samples aus in den vorderen Halbraum verfolgt und gegen das Objekt selbst auf Schnittpunkte getestet. Es genügt dabei zu wissen, *ob* es einen Schnittpunkt gab, da dessen Position ist hier nicht relevant ist.

2. **Rendering-basiert**
   Für diese Methode, die auf dem Rendering durch die Grafikhardware beruht, gibt es prizipiell drei unterschiedliche Konzepte:

   - **Verarbeiten der gerenderten Ansicht**
     Dazu wird das Objekt aus Sicht des Samples in Richtung seiner Normalen gerendert. Dies kann mit einem maximal sinnvollen Öffnungswinkel geschehen oder in mehrere Richtungen (vgl. Single Plane oder Hemicube z.B. in [SP94]). Die resultierende Textur wird auf das Verhältnis der sichtbaren zu gesamten Pixeln hin untersucht und dieser Anteil üblicherweise als Prozentwert abgespeichert. Die Untersuchung kann:
     - **CPU-basiert** sein, d.h. durch Zurücklesen des Framebuffers (oder der Textur bei Render-to-texture) und pixelweises Analysieren des Bildes (Histogramm etc.) in der Applikation.
     - **auf der GPU** durchgeführt werden, indem üblicherweise eine wiederholte Anwendung hardwarebeschleunigten Mipmappings bis hin zu einer 1×1 Textur zum gesuchten Mittelwert führt.

- **Per Occlusion Querying**
  Dabei wird ähnlich dem vorigen Verfahren das Objekt aus Sicht des Samples gerendert. Statt dem Analysieren der Textur wird dazu jedoch das mittlerweile hardwareunterstützte Zählen gezeichneter Pixel eingesetzt und ebenso als Prozentwert gespeichert.
- **Shadowmap-basiert**
  Hier werden im Gegensatz zu den vorigen Verfahren *von außen* aus Sicht zahlreicher, kugelförmig angeordneter Lichtquellen nach dem Shadow Mapping-Verfahren (siehe 2.3.1) erzeugte Tiefentexturen mit *allen* Objektsamples verglichen und diese jeweils verschattet. Die Prozentwerte für die Verdeckung ergeben sich durch Akkumulieren der jeweiligen Werte pro Lichtquelle/Shadowmap. Für dieses Verfahren ist jedoch im Allgemeinen ein texturbasierter Ansatz nötig, d.h. die resultierenden Werte werden *pro Texel* ausgewertet und ergeben somit eine Ambient Occlusion Textur für das Objekt

Das erste der genannten Verfahren ist dabei das flexibelste und — bei entsprechender Implementierung — durchaus performant, da insbesondere bei großen Datenmengen der logarithmische Aufwand von Raytracing gegenüber Scanline-Verfahren zum Tragen kommt. Im Rahmen der Beispielimplementation „sushi“ in [Bie03] wurde diese Methode erfolgreich eingesetzt. Die Anforderungen, die dabei an das System für einen effizienten Einsatz gestellt werden, sind sehr hoch und wurden nur teilweise realisiert. Andererseits ermöglicht es auf einfache Weise die Bestimmung der durchschnittlichen unverschatteten Richtung (*engl.:* average unoccluded direction), die nach Landis auch als „bent normal“ bezeichnet werden. Dazu wird üblicherweise der Mittelwert der Strahlen ermittelt, die *keinen* Schnittpunkt hatten. Obwohl dabei ein Vektor erzeugt werden kann, der nicht mehr in eine unverschattete Richtung zeigt, ist der visuelle Fehler dabei marginal. Prinzipiell wäre zwar hier der Median als mittlerer der sortierten (unverschatteten) Vektoren korrekt, da nur aus bereits vorhandenen Elementen ausgewählt wird; die fehlende Ordnungsrelation von 3D-Vektoren bereitet bei der Sortierung jedoch grundsätzlich Probleme. Mit Hilfe dieser zusätzlichen Information lassen sich Beleuchtungsberechnungen und/oder Texturzugriffe variieren und das spätere Erscheinungsbild verbessern.
Die Berechnung der Bent Normals ist auch mit dem CPU-basierten Renderingverfahren relativ leicht umzusetzen, da auf Basis der Texelpositionen auch die Richtung des unverdeckten Bereiches bestimmbar ist. Problematisch ist dabei jedoch der hohe Kommunikationsaufwand zwischen CPU und GPU, der mit heutigen, meist asnychron arbeitenden Bussystemen ein Geschwindigkeitsproblem darstellt. Zwar sind für das Rendern kaum Texturen größer als 256×256 (entspricht 64k Strahlen) nötig, die sogar auf der

Grafikhardware in eine größere zusammengefasst werden können; dennoch ist — abgesehen von der Geschwindigkeit der CPU-Berechnungen — dieses Verfahren relativ aufwändig und ineffizient. Die GPU-basierte Variante dagegen kann zwar die Hardwarebeschleunigung für die Mipmap-Erzeugung ausnutzen, verliert dadurch aber jede Möglichkeit zur Bestimmung der Bent Normals.

Das nächste Verfahren, dass eine in nahezu jeder aktuellen Grafikkarte zur Verfügung stehende Erweiterung verwendet, ist in der vorliegenden Beispielimplementation umgesetzt und soll daher in Teil III im Detail erläutert werden. Da dieses Verfahren aber ebenso keine Möglichkeit bietet, die Richtung der Verdeckung/Sichtbarkeit direkt zu bestimmen, wurde auf eine Integration der Bent Normals für den weiteren Verlauf der Anwendung verzichtet. Das Konzept dieser zusätzlichen Normalen ist auch nicht ganz unproblematisch, da es ja nach Objektbeschaffenheit potenziell mehrere unverschattete Richtungen geben kann. Mehrere solcher Bent Normals zu verwenden wäre von den Beleuchtungsberechnungen her sicherlich die beste, aber auch aufwändigste Lösung.

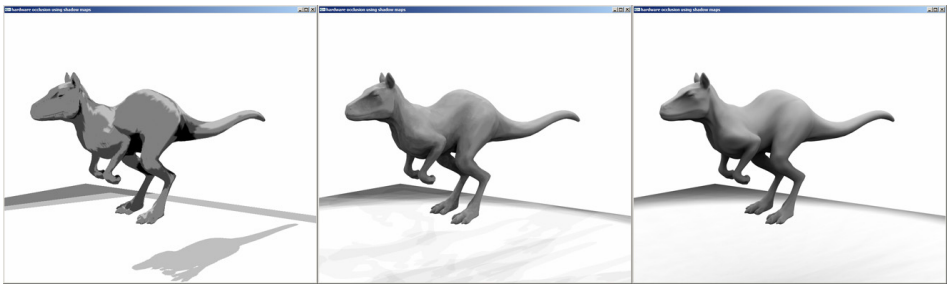

Abbildung 6: Gegenüberstellung verschiedener Auflösungsstufen für die Shadowmap-basierte Berechnung der Verdeckung mit 4 (links), 32 (Mitte) und 512 Lichtquellen (rechts). (Abb. aus [PG04])

Die auf dem Shadow Mapping basierende Variante unterscheidet sich aufgrund ihrer Vorgehensweise deutlich von den bisherigen Verfahren. Dazu wird zu Beginn der Vorverarbeitung eine ausreichende Zahl von Lichtquellen auf einer imaginären Kugel mit gewissem Abtand um das Objekt herum angeordnet. Wie in Abbildung 6 gut erkennbar ist und in [PG04] vollständig zu finden ist, sind erst ab ca. 128 Lichtpositionen kaum Artefakte erkennbar; optimale Ergebnisse erzielen die von Pharr verwendeten 512 Samples, die auf der Testhardware rund vier Sekunden Berechnungszeit benötigten. Man ist damit zwar immer noch recht weit von interaktiven Frameraten entfernt, kann aber diesen offline-Prozess selbst für komplexe

Szenen in praktikable Dimensionen bringen. Zum Vergleich: die oben gezeigte Szene (ca. 150.000 Dreiecke) hätte in der erwähnten Implementation „sushi“ mindestens eine Stunde Vorberechnungszeit benötigt...

Damit ist die einführende Betrachtung der wichtigsten theoretischen und technischen Grundlagen abgeschlossen. Insbesondere im dritten Teil der Arbeit, werden einige der angesprochenen Themen noch einmal aufgegriffen und im Rahmen der praktischen Umsetzung angewendet.

Teil II
Der Einsatz von Spherical Harmonics in dynamischen Umgebungen

Nachdem im ersten Teil sowohl auf die Theorie und die Verarbeitung der Spherical Harmonics als auch auf wichtige weiterführende Techniken eingegangen wurde, soll nun die Verallgemeinerung der Spherical Harmonics auf dynamische Umgebungen in Betracht gezogen werden. Um sich diesem komplexen Thema strukturiert nähern zu können und sowohl vorhandene Ansätze als auch potentielle Möglichkeiten zu erörtern, bedarf es zunächst einer Unterscheidung, was mit „dynamisch“ gemeint ist.

Obwohl es sicherlich mehrere Möglichkeiten gibt, eine sinnvolle Einteilung in verschiedene Kategorien zu definieren, soll hier eine zweistufige Trennung angewendet werden.
In der ersten Stufe wird zunächst die Dynamik von Lichtquellen von der der Objekte getrennt betrachtet. Das zweite Kriterium bildet in den beiden Gruppen jeweils die Unterscheidung in lokale und globale Dynamik bzw. deren Einflüsse im objektweiten und szenenweiten Kontext.

3 Dynamische Beleuchtung

Bisher wurde bei dem Begriff der Beleuchtung immer davon ausgegangen, dass diese gemäß der grundlegenden Annahme des Environment Mappings (EM) nach [BN76] unendlich weit entfernt ist. Auf dieser Vereinfachung basieren auch viele Arbeiten im Zusammenhang mit Spherical Harmonics wie beispielsweise [RH01a], [Gre03] oder [Bie03].
Obwohl sich damit bereits beeindruckende Ergebnisse erzielen lassen, haben insbesondere Sloan et al. [SKS02] gezeigt, dass auch die Beleuchtung von Objekten aus unmittelbar nahen Lichtquellen mit Hilfe von Spherical Harmonics möglich und sinnvoll ist.

3.1 Umgebungsbeleuchtung

Bei der bislang verwendeten und auch weit verbreiteten Repräsentation der Umgebung beschreibt eine aus einem Panorama-Bild erzeugte oder künstlich gerenderte Environment Map die einfallende Beleuchtung (*engl.:* incident lighting). Diese Technik hat sich in der Computergrafik schon seit langem etabliert und ist bereits seit einigen Jahren dank Hardwareunterstützung sehr performant.[15]
Während EM fast ausschließlich für Reflexionen auf spiegelnden Oberflächen verwendet wird, stellt es im Kontext der Spherical Harmonics die Grundlage für die Berechnung der Lichtfunktion dar. Mit Hilfe einer ausreichend großen Anzahl von gleichverteilten Samples auf der als Kugel angenommenen Umgebung[16] werden sowohl die Basiskoeffizienten $C_{\ell,m}$ als auch die der RGB-Beleuchtungsfunktion $L(s)$ bestimmt.
Wenn man nun die mögliche Dynamik einer solchen unendlich weit entfernten Umgebung untersucht, stellt man relativ schnell fest, dass die einzige sinnvolle Transformation eine Rotation um den Ursprung ist; alle anderen Veränderungen haben im Hinblick auf die Grundannahme des EMs keinerlei Auswirkungen auf die Beleuchtung der als unendlich klein angenommenen Szene.
Je nachdem, wie Rotationen im SH-System repräsentiert werden (vereinfacht nach [SP94] bzw. [Bie03] oder vollständig, wie unter anderem in [IR96] und auch im vorliegenden System), gehen in die Berechnung der rotierten Beleuchtungskoeffizienten nur die horizontale Auslenkung φ oder zusätzlich die Latitudinale θ mit ein — insgesamt also ein unkomplizierter und in der Verarbeitung der Spherical Harmonics integrierter Fall der Dynamik.

[15] Neueste Entwicklungen wie z.B. [NVI04] ermöglichen auch die direkt integrierte Verwendung von Fließkomma-HDR-Bildern.

[16] Die heutzutage übliche Repräsentation der Environment Map als Würfel (*engl.:* cube map) ist von dieser Annahme unberührt, da man durch Verwendung von Richtungsvektoren als Samples eine entsprechend verteilte Indizierung des Würfels erreicht.

3.2 Lokale Beleuchtung

Um einiges anspruchsvoller ist die Berücksichtigung von Lichtquellen[17], die sich in der unmittelbaren Umgebung des zu beleuchtenden Objektes befinden. Während bei sehr weit entfernten Umgebungen nur Rotationen sinnvoll sind, spielen hier alle wichtigen Transformationen (Rotation, Skalierung und Translation) eine Rolle. Zudem reicht es nicht mehr aus, das Objekt als unendlich klein im Verhältnis zur Umgebung anzusehen und nur bezüglich eines Punktes (meist Schwerpunkt/Pivotpunkt der Objektgeometrie) zu beleuchten. Denn die sich scheinbar daraus ergebende Möglichkeit, eine „lokale Environment Map", die die entsprechende Lichtquelle enthält, zu erzeugen und diese wie eine herkömmliche Umgebungsbeleuchtung zu behandeln, würde einen zu großen, deutlich sichtbaren Fehler verursachen.
In [SKS02] wird dies eindrucksvoll demonstriert: die fehlerhafte Beleuchtung ist viel zu dunkel und zeigt kaum Unterschiede zwischen nahen und entfernten Regionen der Beispielgeometrie in Relation zur Lichtposition, wie aus der folgenden Abbildung deutlich hervorgeht:

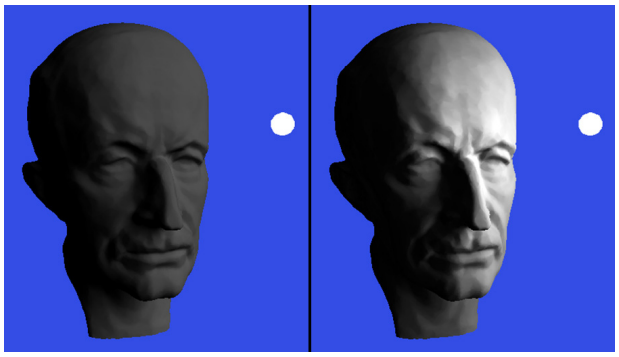

Abbildung 7: Lokal beleuchtete Geometrie mit falschem Ergebnis durch Verwendung einfachen EMs (links) und angenäherte Korrektur durch Interpolation mehrerer lokaler Samples (rechts). (Abb. aus [SKS02])

3.2.1 Radiance Sampling

Als Lösung schlagen Sloan et al. das sogenannte „Radiance Sampling" vor, bei dem statt nur von einem Punkt aus die Szene (insbesondere die lokale Lichtquelle) aus mehreren Perspektiven berücksichtigt wird. Dazu werden

[17] Da hier die Behandlung durch Spherical Harmonics untersucht wird, sind implizit sinnvoll repräsentierbare, d.h. flächenförmige Lichtquellen ausreichender Größe gemeint.

für die zu beleuchtende Geometrie mit Hilfe des aus der Vektorquantisierung stammenden „Iterated Closest Point"-Algorithmus i ausreichend gut verteilte Samples erzeugt. Da beim Einsatz von Spherical Harmonics bereits ein sehr großer Anteil von Rechenzeit im Laufe der Vorberechnungen für die Transferfunktionen verwendet wird, fällt dieser zeitlich auch recht anspruchsvolle Algorithmus nicht weiter ins Gewicht. Neben der Tatsache, dass sich nur niederfrequente Beleuchtung ausreichend mit Spherical Harmonics rekonstruieren lässt und daher wenige ($i < 10$) Samples schon genügen, wäre eine höhere Zahl von Punkten — gar eine Berechnung pro Objektsample (z.B. Vertex) — nicht mehr praktikabel.
Während des Rendervorganges werden schließlich die i leicht unterschiedlichen Lichtfunktionen je nach Abstand zwischen aktuell zu beleuchtendem Punkt auf dem Objekt und dem Samplepunkt gewichtet, der resultierende Helligkeitsbeitrag berechnet und schließlich über die Objektoberfläche interpoliert. Da dieses Verfahren gemäß [SKS02] ohne Probleme sehr hohe Frameraten erzielt, ist man geneigt, es für alle Arten von vorberechneten Transferfunktionen zu verwenden. Während diese Approximation für den unverschatteten und insbesondere für den verschatteten Transfer sehr gute Ergebnisse erzielt, kann sie für höhere Transfers wie Selbstillumination nicht mehr herangezogen werden. Durch die geringe Anzahl der Samplingpunkte und der damit verbundenen starken (linearen) Interpolation der Gewichte über große Teile der Objektgeometrie hinweg, wird implizit von einer lokalen Beleuchtung ausgegangen, während Indirektionen des Lichtweges, d.h. die gegenseitige Beleuchtung der Objektflächen globalen Charakter haben; die Folge wären fehlerhafte Darstellungen dieser Materialeigenschaften.

3.2.2 Vertex-basierte Beleuchtungsgewichtung

Ebenso wie die zuvor beschriebene Technik von soll mit Hilfe von „lokalen" Cubemaps die Beleuchtung von objektnahen Lichtquellen simuliert werden. Um jedoch den großen Aufwand, den das ICP-Verfahren sowohl bei der Vorberechnung (d.h. bei der Bestimmung der i Samples) als auch zur Laufzeit (es müssen i Maps gerendert und verarbeitet werden) zu vermeiden, verwendet der folgende Ansatz zunächst eine herkömmliche Herangehensweise.
Dazu wird aus dem Objektursprung heraus eine Cubemap geringer Auflösung[18] erzeugt, die die Umgebung des Objektes inklusive der Lichtquelle enthält. Um den Berechnungsaufwand zu dezimieren, werden statt zufälliger Samples direkt alle Pixel der Cubemap für die Umwandlung in Spherical Harmonics verwendet. Trotz der dabei fehlenden Gleichvertei-

[18] Sloan et al. [SKS02] schlagen 16×16 Pixel große Bilder pro Cubemap-Seite als ausreichend vor.

lung der „Samples“ ist kein visueller Unterschied zu erwarten, da die in Abschnitt 1.2.2 (Seite 12) erläuterte Tiefpassfilter-Analogie der Spherical Harmonics die potentiell entstehenden Aliasingstrukturen prinzipiell aufhebt. Zudem hat das konstante Sampling den Vorteil, die von der Position der Samples (hier: Pixel) abhängigen Basiskoeffizienten vorab berechnen zu können, da die sphärischen Koordinaten der einzelnen Pixel in der Cubemap bereits bekannt sind.

Bisher hat sich jedoch an dieser Variante gegenüber dem Verfahren von Sloan et al. nichts geändert, im Gegenteil: durch die Verwendung nur *eines* Samples (vom Objektursprung aus) ist dies eine übliche Environment Map. Wenn man nun aber eine zusätzliche metrische Information verwendet, um den Abstand zwischen aktuell zu beleuchtendem Objektvertex und der lokalen Lichtquelle mit einzubeziehen, kann man eine entfernungsabhängige Gewichtung der Beleuchtung erreichen. Die Grundlage für dieses Verfahren bilden die geometrischen Zusammenhänge beim EM im Verbindung mit den Eigenschaften der Spherical Harmonics.

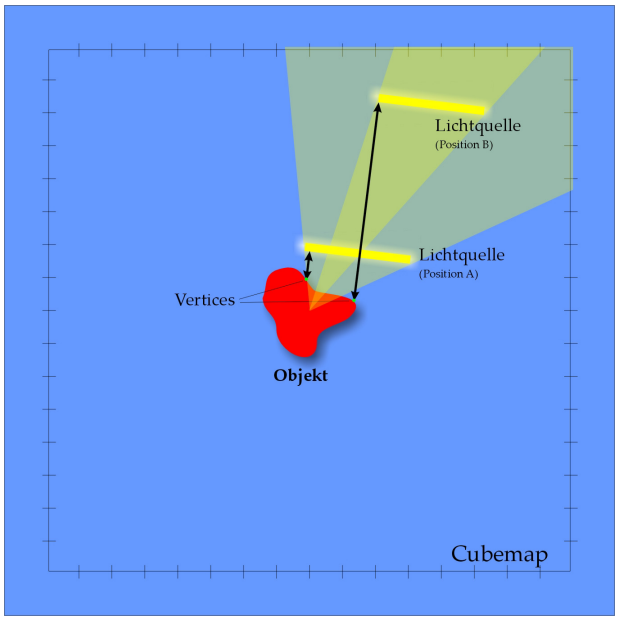

Abbildung 8: Skizzierung des Cubemapping-Verfahrens zur abstandsgewichteten Anpassung der SH-Beleuchtung mittels Vertexshader

Wie aus der in Abbildung 8 skizzierten Konstellation hervorgeht, wird die gleiche Lichtquelle in Abhängigkeit ihrer Entfernung zum Objekt auf unterschiedlich große Bereiche in der Cubemap abgebildet. Da diese Cubemap, die letzlich eine dreidimensionale Lichtverteilungsfunktion darstellt,

mit Hilfe des üblichen Verfahrens durch SH-Basiskoeffizienten angenähert wird, ist das „Signal“ bei der näheren Lichtquelle größer. Anders gesagt: der Raumwinkel, unter dem die Lichtquelle aus Sicht des Objektes gesehen wird, ist umgekehrt proportional zu deren quadratischem Abstand vom Objektmittelpunkt. Um nun eine lokale „Modulation“ der Beleuchtung des Objektes zu erreichen — schließlich unterliegt die vom Objektschwerpunkt aus gerenderte Cubemap immer noch den Eigenschaften herkömmlicher Environment Maps — wird für jeden Vertex im Shaderprogramm der Abstand Vertex–Lichtquelle berechnet und als Gewichtungsfaktor für die Beleuchtung dieses Vertex verwendet.
Darüber hinaus hat die Vergrößerung des Signals in der Lichtfunktion bei geringerem Abstand den positiven Nebeneffekt, dass die bei gegebener maximaler SH-Koeffizientenzahl n (hier: 16, $\ell = 3$) repräsentierbare Mindestgröße der Lichtquelle kleiner wird — sie erscheint ja umso größer, je näher sie dem Objekt ist.

Diese abstandsbasierte Anpassung der Beleuchtung hat gegenüber der oben beschriebenen Methode von [SKS02] zwei grundlegende Vorteile. Da nicht zwischen wenigen vorberechneten Samples interpoliert wird, sondern jeder Objektvertex bei der Bestimmung der Oberflächenfarbe selbst als (modulierendes) Sample eingesetzt werden kann, ist das Ergebnis bezüglich der Interpolation exakter; der Mehraufwand im Vertexprogramm ist dabei für Grafikhardware der vierten Generation vernachlässigbar. Zudem ermöglicht die Auswertung der Vertexdaten ohne weitere Vorberechnungen (wie beim ICP-Algorithmus) eine Veränderung der Geometrie *pro Frame* — also letzlich auch eine echtzeitfähige Deformation des Objektes.
Um dieses Verfahren in seinen verschiedenen Schritten zu verdeutlichen, seien die einzelnen Schritte im Folgenden noch einmal zusammenfassend beschrieben:

1. **Rendern der Cubemap vom Objektursprung aus**
   Hierbei wird idealerweise pro Frame eine Cubemap geringer, aber fester Auflösung als HDR-Textur gerendert und ausgelesen. Als Kameraposition dient hierbei der Ursprung des Objektkoordinatensystems, wobei das Objekt selbst nicht mit gerendert werden darf.

2. **Bestimmung der SH-Koeffizienten der Lichtfunktion auf der CPU**
   Mit Hilfe der konstanten Pixelpositionen wird die Lichtfunktion (d.h. die RGB-Werte der Pixel an ihren (sphärischen) Positionen) in die SH-Basis projiziert und als Reihe von RGB-Vektoren gespeichert.

3. **Laden der `uniform`-Parameter des Objekt-Vertexprogramms**
   Für das Vertexprogramm des Objektes, das bereits als Material vorliegt und zugewiesen ist, werden folgende konstante Parameter gesetzt:

- RGB-Koeffizienten der Lichtfunktion
- 3D-Position der lokalen Lichtquelle
- Parameter für Gewichtungsstärke (optional)

4. **Beleuchten des Objektvertex**
   Folgende Aufzählung fasst die wichtigsten Schritte des Vertexprogrammes zusammen, wobei hier insbesondere der zweite Punkt von Belang ist; die Reihenfolge ist teilweise variabel.

   (a) Inneres Produkt aus SH-Koeffizienten T(s) und L(s)
   (b) Gewichtung dieses Summenwertes mit Abstandsmaß
   (c) Verrechnung mit AO
   (d) Weiterleiten an Fragmentshader

Für die Bestimmung des beschriebenen Abstandsmaßes bieten sich in diesem Zusammenhang zwei Möglichkeiten an. Zum einen eine z-Buffer basierte Variante, die beim Rendern der lokalen Umgebung in die Cubemap zusätzlich die Tiefeninformation bereitstellt sowie eine „manuelle“ Bestimmung des Abstandes von der Kameraposition zur (als Fläche angenommenen) Lichtquelle.
Während die erste Methode den Vorteil hat, prinzipiell auch andere Lichtquellenformen zu unterstützen (z.B. Volumenlichter) und unabhängig von der Anzahl und Ausrichtung der Lichtquellen zu sein, ist sie schwieriger umzusetzen und aufwändiger auszuwerten. Neben der Tatsache, dass ein zusätzlicher Rendering-Pass für den Tiefenpuffer nötig ist[19], stellt die Technik neben dem generellen Aufwand auch hohe Ansprüche an die Genauigkeit der Tiefenwerte. Darüber hinaus kann es an den Rändern der sechs Seiten der Cubemap zu Problemen kommen, wenn sich durch Rundungsfehler unterschiedliche Werte an den Kanten ergeben.
Daher beschränkt sich das umgesetzte Verfahren auf die einfache Abstandsbestimmung anhand der 3D-Position der Lichtquelle. Wie oben schon aufgeführt wird als zusätzliche Konstante die Position der Lichtquelle per `uniform`-Parameter in das Vertexprogramm übergeben. Da die Bewegung der Beleuchtung bereits seitens der Anwendung (also auf der CPU) pro Frame ausgewertet werden muss, ist hier keine merkliche Reduzierung der Geschwindigkeit zu erwarten. Eine Anpassung der Gewichtung kann darüber hinaus durch einen weiteren Parameter leicht erreicht werden. Zudem ist eine gesonderte Berücksichtigung von Lichtquellen, die teilweise von dem Objekt abgewandt sind, nicht notwendig, da die Beleuchtungsinformation durch die Abbildung der Cubemap gegeben ist — und dabei inherent nur die sichtbare Fläche benutzt wird.

[19] Aufgrund des verwendeten Renderingsystems und der noch nicht abgeschlossenen Entwicklung in diesem Bereich kann nicht von mehreren, gleichzeitig nutzbaren Renderingzielen (MRT, Multiple Render Targets) ausgegangen werden.

4 Dynamik der Szene

4.1 Lokale Dynamik

Als erstes Unterscheidungskriterium soll hier die Lokalität der Bewegung verwendet werden. Analog zu dem in der Computergrafik üblicherweise eingesetzten lokalen Beleuchtungsmodell ist hiermit die Dynamik eines Objektes unabhängig von seiner Umgebung gemeint, d.h. die Bewegung wirkt sich nicht auf andere, in der Szene eventuell vorhandene Objekte aus. Im Gegensatz zu dem beispielsweise in OpenGL verwendeten Phong-Beleuchtungsmodell wird das Objekt jedoch als Ganzes betrachtet, damit auch höhere Transferfunktionen definiert sind. Wäre die Lokalität wie beim Phong-Modell auf den aktuellen Punkt (Vertex, Normale etc.) beschränkt, ließe sich nur der unverschattete Transfer beschreiben, der aber im Gegensatz zu höherwertigen Transferfunktionen keinen wesentlichen visuellen Vorteil gegenüber herkömmlichen (lokalen) Beleuchtungsmodellen bietet. Interessant ist dabei, dass sich die unverschatteten SH-Koeffizienten der ersten Bänder sehr gut für eine Interpolation selbst bei deformierbaren Objekten eignen, wie beispielsweise die Animation von Charakteren bei herkömmlicher Beleuchtung realisiert wird; auf diese Kategorie der Dynamik soll aber später noch einmal genauer eingegangen werden.

4.1.1 Objekte als Ganzes

Die separate Berücksichtigung der Beleuchtung für jedes einzelne Objekt hat natürlich gewisse Konsequenzen. Zum einen bedeutet dies in einer Szene mit mehreren, sich bewegenden Geometrien, dass jede für sich korrekt und hochwertig durch eine — vereinfacht gesagt — „Vervielfachung“ der bisherigen Verwendung der Spherical Harmonics beleuchtet werden kann. Obwohl dieser Ansatz sehr primitiv erscheint, kann damit ein gewisser Teil von dynamischen Szenen überzeugend dargestellt werden. Um dies zu belegen, bietet sich eine kurze Überlegung bezüglich der besonderen Eigenschaften und Vorteile der SH-Beleuchtung in allgemeinen Szenarien an.
Die eigentliche Motivation für den Einsatz der Spherical Harmonics zur Beleuchtung virtueller Objekte war die von Ramamoorthi et al. [RH01a] gemachte Feststellung, dass insbesondere diffuse Oberflächen die einfallende Beleuchtung wie ein sehr starker Tiefpassfilter reflektieren und daher die ersten SH-Bänder zur Repräsentation der Beleuchtung ausreichen. Wenn dies aber für unendlich weit entfernte und von allen Seiten einfallende Beleuchtung gilt, so lässt sich diese Annahme auch bis zu einem gewissen Abstand aufrechterhalten, wie dies in Abschnitt 3.2 (Seite 42) bereits erörtert wurde. Konkret heißt das also, dass sich relativ weit von-

einander entfernte Objekte bezüglich der Beleuchtung durch sehr große Lichtquellen (z.B. teilweise bedeckter Himmel, großflächige Fensterfronten in Gebäuden) *nicht sichtbar* beeinflussen. Diese Beobachtung läßt sich auch physikalisch begründen: nach dem photometrischen Gesetz ist die Beleuchtungsstärke E umgekehrt proportional zum Abstandsquadrat der Lichtquelle mit Lichtstärke I:[20]

$$E = \frac{I \cos \theta}{d^2} \tag{4.1}$$

Obwohl dieses Gesetz nur für infinitesimal kleine Lichtquellen (Punktlichter) korrekt ist, kann es dennoch in den meisten Fällen eine gute Approximation auch für realistische Lichtquellen sein. Da sich der in der Computergrafik bestens bekannte Naturwissenschaftler J.H.Lambert (1728-1777) umfassend mit der Theorie des Lichts und dessen Reflexionsverhalten beschäftigt hat, formulierte er unter anderem auch die Abschätzung, dass das oben genannte Abstandsgesetz als Annäherung genügt, wenn die Lichtquelle mindestens fünfmal so weit von dem Objekt entfernt ist, wie ihre geometrischen Ausmaße sind. Erst bei geringerem Abstand wird der Raumwinkel, unter dem die Lichtquelle „gesehen“ wird, zu groß und führt einen nicht mehr vernachlässigbaren Fehler bei Anwendung des Gesetzes aus Formel 4.1 ein.
Durch diese Betrachtungen sollten die mit diesem einfachen Ansatz der lokalen SH-Beleuchtung erreichbaren Effekte abschätzbar sein — und zugleich auch ihre Grenzen. Sobald sich beispielsweise selbst ein weit entferntes Objekt in den Bereich bewegt, aus dem ein anderes Objekt beleuchtet wird, werden die Ergebnisse kaum mehr überzeugen, da die Reduzierung der einfallenden Lichtmenge eigentlich berücksichtigt werden müßte. Bevor man aber dazu geneigt ist, den Ansatz generell zu verwerfen, sei noch einmal auf die grundlegende Motivation des gesamten Themas der Arbeit hingewiesen. Es soll ja gerade die Technik der SH-basierten Beleuchtung mit den Stärken der herkömmlichen Technologien der Echtzeitgrafik zu einem hybriden Ansatz kombiniert werden. Gerade die Spherical Harmonics (in der hier verwendeten Form) eignen sich nur für sehr großflächige Lichtquellen, während das oben skizzierte Szenario eher von einer kleinen Lichtquelle ausgeht; die Signifikanz der Beleuchtungsänderung durch das zweite Objekt wäre sonst nicht so groß gewesen. Dadurch ist eine angemessene Darstellung der Szene mit üblichen Schattenverfahren, wie sie bereits in Abschnitt 2.3.1 auf Seite 31 kurz erläutert wurden, durchaus zu erwarten.

[20] Dies gilt jedoch nur, wenn beeinflussende Faktoren wie atmosphärische Effekte (Nebel, Dunst etc.) vernachlässigt werden.

4.1.2 Festkörper-Transformationen

Mit dem oben geschilderten Verfahren lassen sich damit die Klasse der Festkörpertransformationen (*engl.:* rigid body transforms) unter Berücksichtigung der Einschränkungen objektbezogener, lokaler[21] Beleuchtung gut repräsentieren. Der Unterschied ist dabei jedoch, dass die Objekte nicht mehr unabhängig voneinander betrachtet werden, sondern vielmehr als Objekt*teile* anzusehen sind. Insbesondere ist die damit verbundene, unmittelbare Nähe der einzelnen Teile problematisch, denn die im vorigen Abschnitt hergeleiteten Konzepte beruhen ja gerade auf der Annahme eines „Mindestabstandes". Und obwohl sich mit diesen Transformationen bereits eine sehr große Gruppe von Bewegungen realer Objekte darstellen lassen, kann die direkte Erweiterung der (höherwertigen) SH-Beleuchtung von ganzen Objekten auf Objektteile nicht überzeugend realisiert werden. Als Beispiel sei hier eine sich schließende Autotür aufgeführt: sobald sich die Tür der Karosserie des Fahrzeugs nähert, kann mit einer getrennt lokalen Verarbeitung die gegenseitige Beeinflussung der Teile nicht mehr dargestellt werden. Zwar kann man bei Szenarien wie diesem davon ausgehen, dass die entstehenden Artefakte oder Unplausibilitäten durch die zunehmende Verdeckung gar nicht oder weniger sichtbar werden; dennoch ist eine korrekte Funktionsweise unabhängig von der Szene unabdingbar. Somit ist es sinnvoll, die Betrachtung dieser Bewegungskategorie in den Bereich der *globalen* Dynamik zu verlagern, da diese durch die Berücksichtigung gegenseitiger Einflüsse sich in der Szene bewegender Objekte besser geeignet ist.

4.1.3 Dynamik höherer Transferfunktionen

Wie in der Einleitung dieses Abschnittes auf Seite 47 schon angesprochen wurde, bildet die SH-basierte Beleuchtung der ersten Stufe (unverschatteter Transfer) eine Sonderstellung. Obwohl es bei einer Beschränkung auf diese Repräsentation sogar möglich ist, deformierbare Objekte als nächste und höchste Kategorie lokaler Dynamik korrekt darzustellen, bringt sie gegenüber herkömmlicher Echtzeitbeleuchtung kaum Vorteile. Lediglich die Verarbeitung beliebig großer Lichtquellen ist nur mit den Spherical Harmonics sinnvoll und auch sehr effizient möglich. Doch gerade für die menschliche Wahrnehmung und emotionale Anmutung sind Schatten von sehr großer Bedeutung; es erscheint einfach nicht plausibel, wenn teilweise verdeckte Stellen des Objektes ebenso hell dargestellt werden wie exponierte Bereiche, wie in folgender Gegenüberstellung gut erkennbar ist:

[21] Im folgenden ist damit die Verarbeitung des aktuellen Objektes als Ganzes im Sinne der beschriebenen SH-Beleuchtung gemeint.

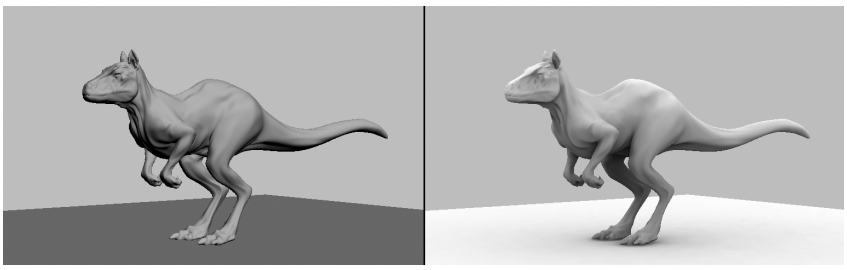

Abbildung 9: Gegenüberstellung herkömmlicher Beleuchtung (links) und durch Ambient Occlusion definierter Beleuchtung (rechts). (Abb. aus [PG04])

Zwar wäre für das unverschattete Objekt auf der linken Seite eine Animation ohne Beeinträchtigung der Qualität bis hin zur Deformation ohne großen Aufwand möglich; wesentlich überzeugender und realistischer wirkt jedoch das zusätzlich verschattete Modell auf der rechten Seite der Abbildung. Insbesondere Bereiche wie die Unterseite der Figur oder die Innenseiten der Beine spielen bei dem visuellen Eindruck eine wesentliche Rolle. Doch was macht es letztlich so schwierig, dieses überzeugend verschattete, jedoch statische Modell zum Beispiel per Keyframes im Kontext der Echtzeitgrafik zu animieren?
Das Hauptproblem ist generell die hohe Varianz der Visibilität bzw. der im Sinne der Spherical Harmonics hochfrequente Verlauf dieser Sichtbarkeitsfunktion. Während sich die Koeffizienten der ersten SH-Bänder inhärent für eine Interpolation (und damit auch für eine Keyframe-Animation) eignen, sind für eine ausreichende Darstellung der Verschattung sehr viele Einzelpositionen nötig, um den Interpolationsfehler gering zu halten und Artefakte zu vermeiden. Die Praxis hat gemäß [PG04] oder [Lan02] jedoch gezeigt, dass sich insbesondere die auch relativ niederfrequente Selbstverschattung, wie sie durch Ambient Occlusion berechnet und dargestellt werden kann, ohne größere Artefakte interpolieren lässt.
Eine ausführlichere Beschreibung des Konzeptes und die verschiedenen Herangehensweisen für dieses Verfahren findet sich in Abschnitt 2.3.2 (Seite 34) im ersten Teil der Arbeit. Darüber hinaus wird in Teil III nochmals kurz auf die Umsetzung dieses Schattenverfahrens im Rahmen der Beispielimplementation eingegangen.

4.2 Dynamik mit globalem Einfluss

Während sich das bisherige Verständnis von Dynamik auf die objektweite Beleuchtung bezogen hat, geht es nun um die mindestens ebenso wichtige

Einbeziehung der (näheren) Umgebung. Wie aus Abbildung 9 gut hervorgeht und nachzuvollziehen ist, hätte eine Animation der Figur unmittelbare Auswirkungen auf die Beleuchtung des Bodens gehabt: sowohl die sehr weiche allgemeine Verschattung unterhalb der Geometrie als auch die sogenannten „Berührungsschatten“ (*engl.:* contact shadows) hätten sich in vielen Bereichen grundlegend verändert. Auf die gleiche Problematik wurde weiter oben schon bei dem Beispiel der sich schließenden Autotür hingewiesen, bei der auch ohne fehlende Berücksichtigung der umgebenden Objekte zu große Fehler gemacht werden.

4.2.1 Keyframing

Die naheliegendste, zugleich aber auch primitivste Lösung ist natürlich die jederzeit durchführbare Vorberechnung der Beleuchtung möglichst vieler Bewegungszustände in Form von Keyframes. Zwar ist es nach entsprechender Vorarbeit während der Laufzeit der Applikation möglich, die Visualisierung von Bewegungen beliebiger Komplexität in hoher Qualität durchzuführen. Dennoch sprechen einige wesentliche Eigenschaften gegen dieses Vorgehen:

- **Der Zeitbedarf steigt linear mit der Anzahl der Keyframes.**
  Als durchaus realistisches Szenario sei das Beispiel der sich schließenden Autotür angenommen, bei dem die komplette Vorberechnung *einer* geometrischen Konstellation ca. 1,5 Stunden dauert. Soll nun diese Tür zur Laufzeit animiert und entsprechend realistisch beleuchtet werden, sind beispielsweise für einen üblichen Bewegungsspielraum einer Tür von rund 75° rund 15 Einzelpositionen[22] zu berechnen — insgesamt also eine Vorberechnungszeit von mehr als 20 Stunden...

- **Der Speicherbedarf und Verarbeitungsaufwand ist sehr hoch.**
  Da für die Darstellung der SH-basierten Beleuchtung eine Unterstützung zur Laufzeit durch Grafikhardware zu empfehlen bzw. erforderlich ist, stellt der quasi-linear wachsende Speicherbedarf ein zunehmendes Problem dar. Zum einen können heutige Grafikkarten nur eine begrenzte Anzahl von Daten in Form von Texturen im Speicher halten und genügend viele Resourcen müssen auch den restlichen Anwendungsdaten (Vertices, andere Texturen etc.) vorbehalten bleiben. Zum anderen stellen die zusätzliche Kommunikation zwi-

[22] Das entspricht einem Winkel von 5° pro Keyframe, bei dem keine wesentlichen Artefakte durch die lineare Interpolation zu erwarten sind.

schen CPU und GPU sowie die aufwändigeren Shaderprogramme sehr hohe Anforderungen an heutige Grafik(sub)systeme.

- **Problematische Interpolation zwischen Keyframes.**
  Verbunden mit dem Speicherproblem ist auch die effiziente Berechnung und Verarbeitung der Interpolation: es müssen mindestens zwei Datensätze im Speicher gehalten werden, damit eine solche Berechnung überhaupt möglich ist. Zudem ist die Interpolation höherwertiger Transferfunktionen schwieriger, da insbesondere die Verschattung eine hohe Varianz aufweist und somit leicht fehlerhafte Darstellungen entstehen können.

- **Vorberechnung schränkt Echtzeitanwendung stark ein.**
  Natürlich stellt die Herangehensweise, also die Vorberechnung mehrerer Zustände eine deutliche Einschränkung bezüglich der Darstellung dar. Dies mag zwar bei dem oben genannten Beispiel der Autotür noch akzeptabel sein — schließlich ist deren Bewegungsraum an sich schon sehr eingeschränkt; aber andere Szenarien, die einen wesentlich höheren Grad an Flexibilität erfordern, würden den Ansatz, alles vorzuberechnen ad absurdum führen.

Offensichtlich ist das Keyframe-Konzept für eine Vorberechnung aller Positionen in der späteren Anwendung mehr als fragwürdig. Dennoch muss man dem entgegenhalten, dass die oben geführte Argumentation von einer direkten und damit naiven Implementation ausgeht, da keinerlei Kohärenzen der Daten ausgenutzt werden.
Besonders im Bereich der Algorithmen für die herkömmliche (d.h. nicht SH-basierte) globale Beleuchtung, mit denen die hier geschilderten Fragestellungen eng verwandt sind, wird schon seit langem an Optimierungen dieser Art geforscht. Da an dieser Stelle nicht auf diese Ansätze eingegangen werden kann, sei auf den sehr umfassenden Überblick und Vergleich der verschiedenen Konzepte von Damez et al. [DDM03] verwiesen.

4.2.2 Vorberechnete Dynamiksimulationen

Einen etwas anderen, wenn auch ähnlichen Weg gehen die Arbeiten von Doug L. James [JF03] und Kayvon Fatahalian [Fat03]. Die darin beschriebenen Techniken bieten sich an dieser Stelle für eine nähere Erläuterung an, da sie für die Beleuchtung auch auf Spherical Harmonics basierende Verfahren nach Sloan et al. [SKS02] zurückgreifen. Obwohl der Hintergrund der Autoren eher in der physikalischen Echtzeitsimulation liegt, zeigen sie mit ihrem Ansatz, dass eine echtzeitfähige, SH-basierte Beleuchtung sogar für interaktiv deformierbare Objekte prinzipiell möglich ist.

Das von ihnen vorgestellte Verfahren gliedert sich im Wesentlichen in die folgenden Verarbeitungsschritte:

1. Erstellen des gewünschten dynamischen Systems in einer beliebigen externen 3D-Anwendung
2. Physikalische Simulation aller möglichen Zustände der durch externe Kraftimpulse beeinflussbaren Szene
3. Reduktion dieses Simulationsmodells durch koeffizientenbasierte Approximation der Impulsfunktionen
4. Berechnung der globalen Beleuchtung anhand dieses Modells
5. Kombination und Interpolation der komprimiert abgespeicherten Bewegungs- und Beleuchtungsdaten zur Laufzeit

Ohne hier auf die genauen Details des Verfahren eingehen zu wollen, seien insbesondere die letzten drei Schritte kurz skizziert.
Da eine vollständige Simulation aller Zustände eines dynamischen Systems praktisch nicht möglich ist, beschränken sich die Autoren zunächst auf Impulsfunktionen. Diese beschreiben, wie das in Schritt 1 erstellte Modell durch externe Impulse (Stöße, ruckartige Bewegungen etc.) aus seiner Ruheposition gebracht wird und dabei allgemeinen physikalischen Gesetzen folgt. Diese in sehr aufwändigen Vorberechnungen erstellten Datensätze werden mittels verschiedener Verfahren — im Wesentlichen der Hauptkomponentenanalyse (*engl.:* PCA, Principal Component Analysis) — auf ein Datenmodell geringerer Ordnung reduziert; dabei werden in ähnlicher Weise wie in Abschnitt 1.3.1, Seite 16 beschrieben die entsprechenden Funktionen (hier: Impuls-Antwort-Funktionen) diskretisiert und approximiert.
Auf diesem reduzierten Simulationsmodell wird die aus [SKS02] bekannte Technik zur Berechnung der SH-basierten Transferfunktionen angewendet und ergibt damit das zum Bewegungsmodell gehörende „Erscheinungsmodell“ (*engl.:* appearance model). Dieser Vorgang entspricht der üblichen Vorberechnung der globalen Beleuchtung, wie sie im ersten Teil der Arbeit beschrieben wurde und wird daher an dieser Stelle als bekannt vorausgesetzt.
Zur Verwendung des Programms im letzten Schritt ermöglicht ein hybrides System aus CPU und GPU die Kombination der Erscheinungs- und Simulationsdaten weitestgehend in Echtzeit. Insbesondere die Änderung der Umgebungsbeleuchtung zwingt die Anwendung in niedrige Frameraten, was laut Fatahalian hauptsächlich auf den hohen Bedarf an Texturzugriffen in den Shaderprogrammen zurückzuführen ist (siehe [Fat03]). Dennoch sind

die Ergebnisse, wie sie aus Abbildung 10 hervorgehen sehr beeindruckend, vor allem vor dem Hintergrund der weitestgehend uneingeschränkten Benutzerinteraktion.

Abbildung 10: Beispiele für in Echtzeit deformierbare Objekte in globaler SH-Beleuchtung: links ein Tuch an einer schwenkbaren Tür, rechts ein horizontal verschiebbarer Pflanzentopf. (Abb. aus [JF03])

Diese Qualität hat jedoch ihren Preis: zum einen ist das verwendete Verfahren ohne sehr tiefe Kenntnisse in Dynamik, physikalischer Simulation und der zugehörigen Mathematik kaum realisierbar. Zum anderen sind die Zeitangaben für die Vorberechnung auf einem handelsüblichen PC der gehobenen Klasse sowohl für die Dynamiksimulation als auch für die SH-Beleuchtung mehr als fragwürdig. So geben die Autoren beispielsweise für das in Abb. 10 gezeigte Tuch (ca. 25k Dreiecke) eine SH-Vorberechnungszeit von gut 71 Stunden an, während die physikalische Simulation knapp 17 Stunden benötigte; das Pflanzenmodell (ca. 10k Dreiecke) benötigte hingegen rund 12 Stunden bzw. eine Woche (!) für die Dynamikberechnungen...

4.2.3 Nachbarschaftstransfer

Ein möglicher Lösungsansatz stammt wieder von Sloan et al. [SKS02], die den sogenannten „Nachbarschaftstransfer“ (*engl.:* neighborhood transfer) entwickelt und demonstriert haben. Dazu wird nicht wie bei den bisherigen Transferfunktionen nur die Objektoberfläche, sondern auch die (leere) *Umgebung* des Objektes durch Samples abgetastet und vorberechnet. Aus Gründen der Einfachheit verzichten Sloan et al. dabei auf eine adaptive, szenenabhängige Unterteilung des Raumes und verwenden statt dessen ein einfaches, gleichförmiges Raumgitter (*engl.:* uniform grid). Für jeden dieser Raumpunkte werden an Stelle der bisherigen, verschatteten SH-Koeffizienten*vektoren* ganze SH-*Matrizen* berechnet und abgespeichert. Da die Oberflächenbeschaffenheit des zu verschattenden Objektes bei der

Vorberechnung nicht feststeht — schließlich sollen ja beliebige Objekte in die Verschattungsumgebung bewegt werden können — müssen die Koeffizienten für eine (begrenzte) Anzahl von Normalenrichtungen vorberechnet werden; dies begründet die Notwendigkeit von Matrizen anstatt Vektoren. Eine Berechnung nur der unverschatteten Koeffizienten macht hingegen bei dieser Form des Transfers offensichtlich kaum Sinn und kommt daher auch nicht weiter in Betracht.

Bewegt sich nun ein weiteres Objekt in die gesampelte Umgebung, so werden für die Beleuchtung zusätzlich zu dessen eigenen Daten die im Raum gespeicherten Koeffizienten ausgewertet und modulieren schließlich den Ergebniswert für die zu verschattende Position. Dadurch ist es möglich, den Einfluß eines Objektes auf die Beleuchtung umliegender Geometrie darzustellen. Dies eignet sich im Fall der von Sloan et al. gezeigten Umsetzung nicht nur für die Verschattung der Umgebung, sondern kann sogar spekulare Beleuchtungseffekte wie Kaustiken[23] simulieren. Die folgende Abbildung verdeutlicht zum einen die realistischen, dynamischen Schatten durch den Neighborhood-Transfer, zum anderen die damit realisierbaren Kaustikeffekte:

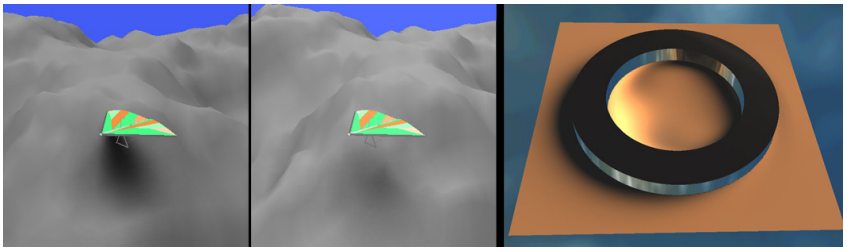

Abbildung 11: Darstellung des Neighborhood-Transfer für die dynamische Verschattung von beliebigen Objekten (Bilder links) und kaustikartige Beleuchtung einfacher Geometrien (Bild rechts). (Abb. aus [SKS02])

Wie man an den Details dieses an sich recht interessanten Ansatzes bereits erahnen kann, ist der Aufwand gegenüber der bisherigen Verwendung der Spherical Harmonics sehr viel höher. Zum einen resultieren aus der Tatsache, dass die Objekt*umgebung* abgetastet wird, andere Anforderungen an das Vorverarbeitungssystem im Hinblick auf Sampling, Daten-

[23] Da die Repräsentation der Kaustiken mit Spherical Harmonics nur eine Approximation durch wenige SH-Koeffizenten (hier: 25, $\ell = 4$) ist, sind diese Effekte nicht mit anderen Techniken wie Photon Mapping vergleichbar. Demgegenüber sind die hiermit erzeugten Indirektionen des Lichtes dynamisch, d.h. in Echtzeit änderbar — was mit herkömmlichen Methoden nicht möglich ist.

haltung und –aufkommen. Damit verbunden ist natürlich auch eine sehr viel geringere Performance, die nur durch Einschränkungen von Freiheitsgraden im interaktiven oder echtzeitfähigen Bereich liegt. So stellen Sloan et al. einen Einbruch der Rendergeschwindigkeit von rund 120 fps (statische Lichtposition) auf 4 fps bei keinerlei Einschränkungen fest. Da diese Performancewerte bereits für eine ziemlich grobe Diskretisierung der Objektumgebung (hier: $64 \times 64 \times 8$ Samples) verhältnismäßig gering sind, kann weder von einem effizienten Einsatz komplexerer Objekte noch von einer Integration in interaktive Anwendungen, die gleichzeitig noch Simulation, Ein-/Ausgabeverarbeitung etc. realisieren müssen, ausgegangen werden. Es ist auch für die nähere Zukunft eher unwahrscheinlich, dass neue Grafikkartengenerationen diesen Aufwand bewältigen können. Zudem widersprechen auch die teilweise sehr langen Vorberechnungszeiten (ca. 3 Stunden für das genannte Beispiel) einem flexiblen Einsatz — von dynamischen Kontexten bis hin zu verformbaren Objekten ganz zu schweigen.

In diesem Zusammenhang verspricht eine andere Herangehensweise, die dem im vorliegenden System realisierten Ansatz ähnlich ist, viel flexiblere Einsatzmöglichkeiten. Die demnächst erscheinende Arbeit von Kautz et al. [KLA04] führt auf einem Geometriemodell geringerer Komplexität eine Visibilitätsberechnung *pro Frame* aus, weshalb dieses Verfahren völlig ohne Vorberechnungen auskommt; die Projektion der Lichtfunktion in die Basis der Spherical Harmonics geschieht dabei ebenfalls „on-the-fly". Interessanterweise ist das Verfahren von Kautz et al. aufgrund der besonderen Bitoperationen rein CPU-basiert, obwohl die Autoren für die Zukunft eine bessere Lastverteilung zwischen CPU und GPU anstreben.
Natürlich schränkt die Verwendung von low-poly-Modellen die Flexibilität und den Einsatz für reale Anwendungen ein. Demnach wäre eine Reduktion komplexer Geometrie, wie sie im Rahmen der praktischen Umsetzung anhand von Beispieldaten der Volkswagen AG nicht unproblematisch, zumal die Autoren in [KLA04] einfache Modelle der Größenordnung <500 Dreiecke verwenden und auf Highend-PC-Systemen arbeiten. Andererseits wird hier von *deformierbaren* Objekten ausgegangen, was beispielsweise bei der oben genannten, schließenden Tür unnötig wäre. Darüber hinaus gehen solche exemplarischen Umsetzungen meist von einfachen Szenen (ein Objekt in unendlich weit entfernter Umgebung) aus, die bei der Erweiterung auf komplexere Szenarien (mehrere Objekte, lokale Beleuchtung etc.) konzeptuell umgestaltet und/oder stark vereinfacht werden müssen.

Diese Gründe führten in ihrer Gesamtheit zu der Entscheidung, diese Technik für eine Umsetzung in der vorliegenden Beispielimplementation nicht in Betracht zu ziehen.

Abschließend lässt sich daher festhalten, dass der Einsatz Spherical Harmonics basierter Beleuchtung in dynamischen Kontexten alles andere als trivial ist. Wie in den vorangegangenen Abschnitten gezeigt wurde, lassen zwar einige Situationen bzw. vereinfachende Annahmen eine Verwendung dieser (wesentlich auf Vorberechnungen beruhenden) Beleuchtungsrepräsentation zu. Von einer performanten Umsetzung für *mehrere, deformierbare Objekte realistischer Größe*, wie sie beispielsweise in interaktiven Simulationen oder Computerspielen erforderlich sind, ist man jedoch immer noch weit entfernt.
Obwohl gerade im Rahmen einer solchen Arbeit eine universelle Dynamik aller Komplexitätsstufen nicht realisiert werden kann, wurde dennoch versucht, zumindest die Beschränkung der Technologie auf nur jeweils *ein* Objekt mit *rein diffusem* Material (wie beispielsweise in [Bie03]) aufzuheben. Wie im folgenden Teil beschrieben wird, ist es damit möglich, *mehrere* Objekte innerhalb der Szene zu *bewegen* und dabei auch *Reflexions- und Beleuchtungseffekte* zu integrieren. Auch die Durchführung der zeitaufwendigsten Anteile — insbesondere zur Berechnung der Selbstverschattung — ist von Grund auf verändert worden, so dass ein effizienterer und flexiblerer Einsatz möglich wird.

Teil III
Die praktische Umsetzung

Um einige der in den vorangegangenen Teilen theoretisch erörterten Konzepte zu konkretisieren und zu verdeutlichen, ist als Bestandteil der vorliegenden Arbeit die Beispielimplementation „sashimi“ erstellt worden.
Zunächst soll dabei auf den Entwurf und die Zielsetzung des Systems sowie auf die verwendeten Komponenten des Systems eingegangen werden. Dabei sollen auch die Entscheidungen für oder gegen bestimmte Verfahren oder Technologien nicht unerwähnt bleiben. Nach einer kurzen Beschreibung der Anwendung wird auf wesentliche technische Details und in diesem Kontext auch besondere Techniken näher eingegangen. Dieser Teil wird durch eine kurze Bewertung der Ergebnisse, die beim praktischen Einsatz erzielt wurden, abgerundet und soll abschließend für einen kurzen Ausblick motivieren.

5 Die verwendeten Komponenten

In diesem Abschnitt wird ein kurzer Überblick zu den verschiedenen Komponenten gegeben, aus denen die Beispielimplementation „sashimi“ aufgebaut ist. Dabei geht es neben einer kurzen Charakterisierung der Module vor allem auch um die Auswahl vorhandener Systeme oder Bibliotheken, sowie um die Beschreibung von nötigen Anpassungen.

5.1 Die Plattform

5.1.1 Programmiersprache

Ebenso wie die Beispielimplementation „sushi“ aus [Bie03], ist auch das vorliegende Programm komplett in **C++** geschrieben. Dies begründet sich hauptsächlich aus der Tatsache, dass es (nicht nur) in der Computergrafik auf maximale Performanz ankommt und somit praktisch kein Weg an C-basierten Sprachen vorbeiführt. Auch die Anbindung der weiter unten beschriebenen Komponenten ist ausschließlich für C/C++ vorgesehen, so dass diese Entscheidung von Beginn an feststand. Für die Benutzeroberfläche wird wieder die plattformunabhängige und sehr flexible Bibliothek **Qt** [Qt94] eingesetzt, zu der auch eine sehr gute Schnittstelle von Seiten des verwendeten Renderingsystems existiert.

Im Bereich der Sprachen für die Shaderprogramme sieht es dagegen nicht mehr so eindeutig aus; in Abschnitt 2.1.2 auf Seite 24 wurden die verschiedenen Hochsprachen bereits vorgestellt. Die Low-Level-Sprache Assembler, in der ebenso Shaderprogramme erstellt werden können und in früheren Generationen programmierbarer Grafikhardware die einzige Option war, schied aufgrund der Komplexität der benötigten Funktionen aus. Zwar wäre eine Programmierung in Assembler von der technischen Seite möglich, aber wie später an Beispielen zu sehen ist, sind Shader in einer Hochsprache wesentlich einfacher zu erstellen und zu lesen.
Aufgrund der derzeit noch fehlenden Unterstützung von „GLslang“ konnte diese wünschenswerte, da unabhängige Programmiersprache nicht verwendet werden. Ebenso kam HLSL im Zusammenhang mit DirectX nicht in Frage, da das unten erläuterte Renderingsystem auf OpenGL aufsetzt und keine Schnittstelle zu dieser proprietären Grafik-API bietet.
Somit blieb letztlich **Cg** [FK03] als einzige geeignete Alternative übrig. Der Vorteil dieser Shadersprache war zum einen, dass die Syntax aufgrund ihrer Nähe zu C/C++ leicht verständlich und direkt einsetzbar war. Zum anderen bot das verwendete Renderingsystem eine (nahezu) ausreichend vollständige Schnittstelle zu Cg, die nach geringfügigen Erweiterungen ideal in den benötigten Kontext der Anwendung passte.

5.1.2 Betriebssystem und Hardware

Idealerweise sollte eine Anwendung plattformübergreifend sein, d.h. sie sollte auf verschiedenen Betriebssystemen lauffähig sein. Aus den Erfahrungen mit „sushi“ war jedoch klar, dass dies aus verschiedensten Gründen nur mit zum Teil beträchtlichem Aufwand möglich ist. Darüber hinaus ist die vorliegende Implementation noch von weiteren externen Komponenten abhängig, deren Einsatzfähigkeit unter anderen Systemen zusätzlich sichergestellt werden müsste.
Daher wurde aufgrund des unkomplizierteren Umgangs bei der Entwicklungsarbeit das von den meisten Komponenten primär unterstützte Betriebssystem **Linux** verwendet; die Portierung auf andere Systeme ist zwar möglich, war aber nicht Gegenstand der Arbeit.

Im Bereich der Hardware gibt es prinzipiell keine Einschränkungen, die die eine oder andere Grafikplattform zwingend erforderlich machen. Die einzige unabdingbare Anforderung ist eine ausreichend moderne Grafikkarte der vierten Generation, was aber in heutigen Rechnern als de-facto-Standard vorausgesetzt werden kann.
Der Einfachheit halber wurde jedoch die Entscheidung getroffen, sich auf einen Grafikkartentyp festzulegen, da sich die zwei in Frage kommenden Systeme (ATI und NVIDIA) in Details teilweise erheblich unterscheiden. Die Wahl fiel hierbei auf **GeforceFX**-Grafikkarten, da diese im Zusammenhang mit der vorliegenden Arbeit mehrere Vorteile bieten:

- optimale Unterstützung der Shadersprache Cg
- durchgehende 32 Bit-Fließkommagenauigkeit (ATI: 24 Bit)
- vollständige Verfügbarkeit des `half`-Formats[24]
- unkomplizierte und stabile Treiberunterstützung unter Linux

Für die Zukunft bleibt daher zu hoffen, dass die Möglichkeiten der verschiedenen Grafiksysteme durch unterschiedliche Architekturen nicht noch weiter auseinander gehen bzw. ein breiter Standard wie OpenGL den universellen Einsatz von Anwendungen garantiert — ohne den Aufwand von Spieleherstellern für maximale Kompatibilität betreiben zu müssen...

[24] Mit „vollständig“ ist hier gemeint, dass auch in der Spezifikation von Cg definierte spezielle `pack`-/`unpack`-Methoden zur Verfügung stehen, die einen durchgehenden 32 Bit-`float`-Datentyp erfordern.

5.2 Das Renderingsystem

Je umfangreicher die Anforderungen an eine Anwendung werden, umso aufwändiger werden auch die Funktionen, die im Hintergrund laufen: vom Laden von Dateien über einfache Bearbeitungsfunktionen bis hin zu Verwaltungsaufgaben sollte alles vorhanden sein. Und da die Aufgabenstellung die gleichzeitige Verwendung *mehrerer* Objekte impliziert, ist die Notwendigkeit eines Szenengraphensystems offensichtlich. Obwohl es eine Vielzahl solcher Systeme gibt, ist die Entscheidung im Wesentlichen aus zwei Gründen nicht sonderlich schwer gefallen.
Zum einen bietet das hier verwendete, frei verfügbare Szenengraphsystem **OpenSG** [Ope00b] eine konkurrenzlose Flexibilität und Performance — nicht nur unter den Open-Source Systemen. Zum anderen setzt es unmittelbar auf der Grafikhardware-API **OpenGL** auf und ist daher sowohl herstellerunabhängig als auch plattformübergreifend einsetzbar. Zwar war die Dokumentation des Paketes zu Beginn der Arbeit nicht so ausgereift, wie man es oftmals benötigt hätte oder von anderen Bibliotheken kennt. An der Verbesserung wird jedoch ebenso erfolgreich gearbeitet wie an der ständigen Weiterentwicklung und Erweiterung des Funktionsumfangs.

Für die speziellen Bedürfnisse der hier vorliegenden Arbeit und insbesondere der Beispielimplementation kann OpenSG natürlich nicht jede benötigte Funktionalität mitbringen. Dennoch sind insbesondere die Anbindung von Cg in Form eines integrierten, jedoch optionalen Moduls (Contribution „CGChunk“) und die Handhabung von OpenGL-Extensions recht komfortabel und nahezu vollständig ausreichend. Es wurden lediglich einige Ergänzungen nötig, die in den nun folgenden Abschnitten erläutert werden sollen.

5.2.1 Integration eines `half`-Datentyps

Diese fundamentale Erweiterung des Systems steht in direktem Zusammenhang mit der bereits in Abschnitt 2.2.1 (Seite 28) vorgestellten Bibliothek **OpenEXR** [HK+02]. Da diese HDR-Bilder intern aus 16 Bit-Fließkommawerten aufgebaut sind und in direkter Zusammenarbeit mit NVIDIA entwickelt wurden, um eine direkte Hardwareunterstützung zu ermöglichen, lag es nahe, dieses `half`-Format auch an den relevanten Stellen der Applikation direkt zu verwenden. Zwar hätten sich diese 16 Bit-Bilder speziell für diesen Zweck mit bereits in OpenSG verfügbaren `Int16`-Daten, d.h. unmittelbar auf Bitebene repräsentieren lassen; eleganter und flexibler war jedoch die durchgehende Ergänzung des Systems um diesen neuen

`Real16`-Typ.[25] Eine vollständig integrierte Unterstützung in der offiziellen OpenSG-Version ist aber laut den Entwicklern aufgrund der noch sehr jungen und daher wenig standardisierten Technologie bisher nicht vorgesehen.

5.2.2 Erweiterung der vorhandenen Cg-Schnittstelle

Während die Cg-Shaderschnittstelle in Form der Contribution „CGChunk" alle wichtigen Funktionen der Cg-Runtime bereitstellt, erschwerte eine Beschränkung die Verwendung von komplexeren Shadern: es konnten als `uniform`-Parameter von der Applikation zu den Vertex- bzw. Fragmentprogrammen nur `float4`-Werte übergeben werden. Zwar ließen sich meist mehrere Einzeldaten zu einem solchen SIMD-Typ zusammenfassen, so dass auch gleichzeitig Ressourcen gespart werden konnten. Für die häufig benötigte Übergabe von `float4x4`-Matrizen wäre jedoch ein hoher Aufwand auf beiden Seiten der Schnittstelle (Applikation *und* Shader) zum Zerlegen bzw. Zusammenfügen erforderlich gewesen. Da bei Shaderprogrammen generell jede unnötige Instruktion aus Effizienzgründen vermieden werden sollte, wäre hier eine deutliche Leistungseinbuße zu erwarten; insbesondere die Verarbeitung der Spherical Harmonics auf der GPU (siehe 6.3.3) ist hier ein sehr gutes Beispiel für Notwendigkeit dieser Erweiterung.
Daher fügt die Erweiterung der vorhandenen `OSGCGProgramChunk`-Klasse noch zusätzliche Felder für eine direkte Unterstützung von 4×4-Matrizen (also `Matrix<Real32>`) ein, die völlig parallel zu den Strukturen für die `Vec4f`-Daten angelegt wurden. Intern werden die Daten entsprechend der Cg-Runtime-Spezifikation als `float4x4` bzw. `float4` übergeben; im Falle der Matrizen ist die spaltenweise Reihenfolge der Einträge (*engl.:* column major) zugunsten der Effizienz und Einfachheit konform zu OpenSG.

5.2.3 Spezialisieren der `OSGQGLManagedWidget`-Klasse

Da für die grafische Oberfläche der Anwendung Qt zum Einsatz kommt, wurden die von OpenSG bereitgestellten Widgets verwendet. Obwohl es von diesen Komponenten zum Darstellen des OpenSG-Inhalts mehrere Varianten gibt[26], war eine Spezialisierung aus zwei Gründen nötig:

- Objekte in der Szene sollten bequem transformiert werden können
- einige Cg-Shader benötigen *pro Frame* aktualisierte Daten in Form von `uniform`-Parametern

[25] Eine bedingte Kompilierung stellt dabei sicher, dass dieses `Real16` auch auf Systemen ohne entsprechenden Erweiterungen funktionsfähig ist. Bei Vorhandensein solcher Extensions wird `Real16` auf `half` abgebildet, andernfalls auf übliches `float`.

[26] mit/ohne `SimpleSceneManager` und eine spezielle Variante für den Qt-Designer

Für den ersten Punkt war es wünschenswert, die Manipulation selektierter Objekte und der Skybox, d.h. im Wesentlichen Rotationen und Translationen, so einfach und intuitiv wie möglich zu gestalten. Da OpenSG diese Art der Interaktion bereits für die standardmäßige „Betrachterkamera“ in Form des im `SimpleSceneManager` enthaltenen `Navigator`[27] zur Verfügung stellt, lag es nahe, diese Funktionalität auch für die Steuerung der Objekte zu verwenden. Da diese `Navigator`-Klasse aber primär für den Einsatz von *Kamera*steuerungen vorgesehen ist, musste aufgrund ungeeigneten Verhaltens nach anderen Möglichkeiten gesucht werden. Aus diesem Grund fiel schließlich die Wahl auf die `Trackball`-Klasse, die, obwohl sie nicht mehr zur Verwendung vorgesehen ist (deprecated), genau den Bedürfnissen entsprach und daher verwendet wurde.
Damit war zum einen sicher gestellt, dass die Szenenkamera innerhalb des `SimpleSceneManager`s nicht von Objekt-/Skyboxtransformationen beeinflusst wird. Zum anderen sollte es aber auch möglich sein, *verschiedene* Objekte *unabhängig voneinander* zu manipulieren — was mit nur einem `Trackball` innerhalb des Widgets nicht per se gegeben ist. Eine flexible Lösung wurde dadurch erreicht, dass die Transformation des aktuellen Objektes, die ohnehin ständig den Transformationsknoten oberhalb des Objektteilgraphen (siehe Abbildung 12, Seite 67) für die Darstellung aktualisiert, anfangs auf den `Trackball` übertragen wird. Dadurch kann die Manipulation eines Objektes bei der Auswahl an der Stelle fortgesetzt werden, an der sie vorher beendet wurde.

Außerdem benötigen die meisten der eingesetzten Vertexprogramme `uniform`-Parameter, die zwar für das gesamte Objekt (d.h. für alle Vertices dieser Geometrie) konstant bleiben, sich jedoch *pro Frame* ändern können; Beispiele hierfür wären Licht- und Kamerapositionen in Weltkoordinaten oder Belichtungseinstellungen für Tonemapping. Daher wurde die dafür zuständige Methode `paintGL` der Elternklasse (`OSGQGLManagedWidget`) überladen und erzeugt nun ein zusätzliches Qt-Signal, das die Applikation zum Aktualisieren aller relevanten Shaderprogramme veranlasst.

[27] Die Standardeinstellung ist hierbei eine Trackball-Metapher, die durch Aktvieren des internen `TrackballNavigator` realisiert wird.

6 Beschreibung des Systems

Zur Aufgabenstellung dieser Arbeit gehört auch die exemplarische Umsetzung verschiedener Aspekte, die in den vorigen Kapiteln erläutert wurden. Aufgrund des Umfangs und der Vielfalt der Möglichkeiten, wie sie vor allem im zweiten Teil vorgestellt wurden, kann und soll „sashimi“ nicht jedes Verfahren umsetzen. Darüber hinaus ist die Beispielanwendung auch nicht als vollständige Applikation zu verstehen, sondern soll einige der für das Thema dieser Arbeit relevanten Techniken demonstrieren.
In den folgenden Abschnitten wird daher zunächst auf grundlegende Entscheidungen eingegangen und das grobe Konzept des Programmes beschrieben. Einige besondere Details und umgesetzte Verfahren werden im Anschluss daran aus praktischer Sicht und aufbauend auf den Grundlagen des ersten Teils erörtert.

6.1 Allgemeiner Überblick

6.1.1 Das Grundkonzept

Wie bereits in Abschnitt 5.2 beschrieben wurde, basiert die Implementation auf OpenSG — nicht zuletzt aufgrund des sehr umfangreichen Funktionsumfangs dieses Szenengraph-Systems. Da es sich bei einem Szenengraphen aber in erster Linie um eine höhere Abstraktionsebene zu Grafikbibliotheken wie OpenGL handelt und daher für die *Darstellung* konzipiert ist, eignen sich dessen Strukturen nur bedingt für die vorliegenden Bedürfnisse. So sind beispielsweise die für die Spherical Harmonics benötigten Berechnungen zwar direkt auf dem Szenengraphen möglich, effizienter ist es aber in nahezu allen Fällen[28], die Teilhierarchie des entsprechenden Objektes zu reduzieren.
Aus diesem Grund werden neben dem Szenengraphen zur visuellen Repräsentation der gesamten Szene noch eine Reihe weitere, interne Daten angelegt und verarbeitet. Diese unterteilen sich — analog zur Struktur des Graphen für die Darstellung — zunächst in **Objekt-** und **Umgebungsdaten**. Diese Teile können jeweils unabhängig voneinander und entsprechend ihren Einsatzmöglichkeiten (siehe z.B. Abschnitt 3.1) transformiert werden, wobei Objekte zudem als Ganzes oder einzeln steuerbar sind.

Zur Verdeutlichung zeigt Abbildung 12 die wesentlichen Komponenten anhand eines OpenSG-Graphen. Auf der linken Seite ist die gesamte Szene, bestehend aus den geladenen Objekten (hier: Object1 bis ObjectN) und

[28] Wenn die gesamte Hierarchie des Objektes nur aus *einem* Knoten besteht, ist natürlich kein Gerschwindigkeitsvorteil zu erwarten. Da dies jedoch selten der Fall ist (in [Bie03] jedoch vorausgesetzt werden musste), kann im Allgemeinen von einem Performancegewinn ausgegangen werden.

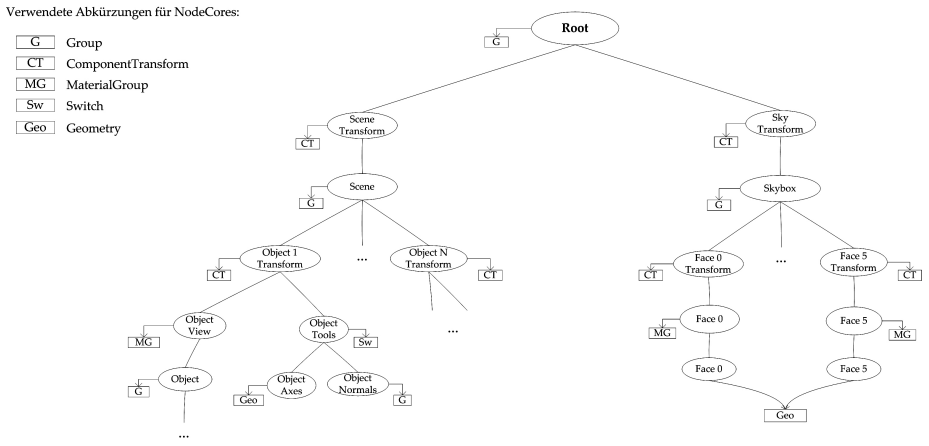

Abbildung 12: Der verwendete Szenengraph der Beispielimplementation im vereinfachten Überblick.

zahlreichen, automatisch hinzugefügten Knoten skizziert. Diese dienen im Wesentlichen der Einblendung zusätzlicher Daten (Normalen, Koordinatenachsen) und der Transformation des Objektes. Wie oben beschrieben kann auch die gesamte Szene, d.h. *alle* geladenen Objekte mit Hilfe eines weiteren Transformationsknotens manipuliert werden.
Die Umgebung, die aufgrund der Umsetzung als Würfel auch meist als „Skybox" bezeichnet wird, ist auf der rechten Seite des Graphen dargestellt. Diese Box ist aus sechs einzelnen Quadraten mit nach innen gerichteten Normalen aufgebaut, die dadurch einfach texturiert werden können. Alle Geometrien der Szene befinden sich innerhalb dieser Box, deren Abmessung weit über die des größten geladenen Objektes hinausgeht.[29]

Für den üblichen Ablauf des Programmes wurde ein ähnliches Konzept wie in [Bie03] verwendet, das aber zugunsten der vorliegenden Plattform und der Flexibilität erweitert wurde. Grundsätzlich orientiert sich die Anwendung dabei an folgendem Ablauf:

1. **Erstellen der Szene**
   - Laden von Objekten (VRML-Dateien)
   - Laden von Umgebungsbildern (EXR-Dateien)[30]
   - Laden von Kamerakurven (M-Dateien)[31]
   - Anpassen der Umgebungsdarstellung

[29] Gegebenenfalls wird das geladene Objekt um einen genügenden Faktor herunter skaliert, da eine zu starke Vergrößerung der Box zu Darstellungsfehlern (z-Buffer) führt.
[30] siehe Abschnitt 2.2.1, Seite 26
[31] HDR Shop-Format, siehe `http://www.debevec.org/HDRShop`

2. **Editieren der Szene**
   - Auswählen von Objekten („picking“)
   - Löschen/Verändern selektierter Objekte
   - Wechsel der Darstellung der Objekte
3. **Vorberechnungen**
   - Spherical-Harmonics-Koeffizienten (unverschattet)
   - Ambient Occlusion
4. **Parameter-Einstellungen**
   - Weitere Anpassungen der Umgebungsdarstellung
   - Einstellung von Materialparametern
   - Lichtparameter

Auf die veschiedenen Möglichkeiten, die sich aus diesem exemplarischen Ablauf ergeben, sowie auf Bedienung und Steuerung der Software soll in den folgenden Abschnitten detaillierter eingegangen werden.

6.1.2 Designentscheidungen

Während die Funktionalität des Programms durch die Aufgabenstellung in einem gewissen Rahmen vorgegeben war, ergaben sich dennoch sehr viele Entscheidungsmöglichkeiten für oder gegen die verschiedenen Techniken, die größtenteils in Teil I erläutert wurden. Daher sollen hier kurz die wichtigsten Entscheidungen aufgeführt werden, die zur Umsetzung der ausgewählten Verfahren führten.

Vertex- oder pixelbasierte Berechnungen

Vor allem bei den beiden zentralen, neueren Technologien (Spherical Harmonics und Ambient Occlusion) besteht die Möglichkeit, die Berechnungen auf Basis der Objekt*vertices* oder zugehöriger *Texturen* durchzuführen; im letzteren Fall spricht man auch oft von „pixelbasierter“ Berechnung.[32] Die Unterscheidung der beiden Vorgehensweisen lässt sich relativ leicht am Beispiel der Selbstverdeckung beschreiben. Während beim vertexbasierten Verfahren die Schattenfühler, Kamera etc. (siehe 2.3.2) nur von den *Vertices* des Objektes ausgehen und Werte für diese liefern, basiert die *pixelbasierte* Berechnung auf den Texturpixeln des zugehörigen Objekts. Dies

[32] Korrekterweise müsste man eigentlich von „texelbasierten“ Verfahren sprechen. Da dies aber ohnehin aus dem Kontext hervorgeht und aufgrund der Komplexität der Verfahren nicht mit der Berechnung durch den Fragmentshader realisierbar und verwechselbar ist, wird allgemein nur von „Pixeln“ gesprochen.

stellt natürlich wesentlich höhere Anforderungen an die zu ladenden Objekte als das rein auf Eckpunkten basierende Vorgehen. Allein das Vorhandensein von Texturkoordinaten für jeden Objektvertex erfordert im Allgemeinen eine umfangreiche Aufbereitung der Daten bzw. eine zusätzliche Verwendung der zugrunde liegenden parametrischen Beschreibungen.

Aufgrund der unterschiedlichen Vorgehensweisen erzielt das texturbasierte Verfahren in den meisten Fällen bessere Ergebnisse, da das Objekt durch die Texel wesentlich besser „abgetastet" werden kann und dadurch inhärent unabhängig von der Auflösung des Meshes ist. Dazu würde beispielsweise bei der in „sashimi" verwendeten Technik (Rendern aus Sicht des Samples per Occlusion Querying) mit Hilfe der Texturkoordinaten eine Abbildung des Texels zurück in das Objektkoordinatensystem erfolgen, von dem aus dann die gesamte Objektgeometrie gerendert werden kann. Der resultierende Wert (hier: ein Skalar) würde in der beispielsweise monochromen Textur an der aktuellen Texelposition gespeichert werden.
Andererseits spricht für die Verwendung der Vertices neben der einfacheren Umsetzung auch noch der Fall, dass Meshes auch sehr hochaufgelöst sein können, wie dies beispielsweise bei der Visualisierung im Automobilbau üblich ist.[33] Die meist aus den parametrischen Beschreibungen erzeugten Polygonnetze werden anhand der Topologie tesseliert, d.h. vor allem im Bereich starker Krümmungen (feine Strukturen, Kanten etc.) sind die resultierenden Dreiecke sehr klein, um den benötigten Detailgrad zu erreichen. Um aber die Texturgröße in einem realistischen Rahmen zu halten, würde für diesen Fall mit hoher Wahrscheinlichkeit *ein Texturpixel mehreren Dreiecken* zugeordnet, was wiederum zu Problemen bei der Darstellung führen kann (vor allem bei der Kombination mit weiteren Texturen).

Aus diesen Gründen fiel die Entscheidung für die vorliegende Implementation auf die **vertexbasierte** Berechnung der Koeffizienten für Spherical Harmonics und Selbstverschattung. Dies hat darüber hinaus auch den zusätzlichen Vorteil, dass für die Umsetzung der Verfahren auf der Grafikhardware eine effizientere Verteilung der Last zwischen Vertex- und Fragmentshader möglich ist. Obwohl bereits neueste Entwicklungen, wie sie in Abschnitt 2.1.1 kurz erwähnt werden, auch Texturdaten für Vertexprogramme zugänglich machen, können die benötigten Daten bis zu einem gewissen Umfang auch direkt in den Objektvertices untergebracht werden und sind somit bereits in Vertexshadern der aktuellen Generation von Grafikhardware verwendbar.

Die Verarbeitung der Umgebungsdaten

Entgegen den ersten Planungen und Entwürfen für das System, in denen

[33] Dies wird anhand von realer Geometrie im Abschnitt 7 noch genauer erläutert.

Informationen aus dem geladenen Umgebungsbild mit Hilfe umfangreicher Bildverarbeitungstechniken gewonnen werden sollten, wurden diese letztlich auf einige wenige Operationen reduziert. Der Grund hierfür war, dass die vorgesehenen Segmentierungsverfahren zur Merkmalsextraktion potentieller Lichtquellen wie graphentheoretisches Clustern, Region Growing (siehe z.B. [Jäh97]) oder Structured Importance Sampling [ARBJ03] eine verhältnismäßig hohe Laufzeit haben. Diese kann bei den für eine ausreichende Repräsentation der Umgebung benötigten Bildgrößen von mindestens 1024×512 (LatLong) bzw. 256×1536 (EXR-Cubemap) bereits im Minutenbereich liegen.
Da es aber insbesondere die Technik der von der Beleuchtung *un*abhängigen und damit vorab berechenbaren Spherical Harmonics-Koeffizienten erst ermöglichte, die Umgebung „on-the-fly“ zu wechseln, wäre eine solche zusätzliche Vorberechnung bei jedem Laden der Beleuchtungsumgebung nachteilig. Daher wurde auf den Einsatz spezieller Algorithmen verzichtet, zumal im Rahmen der vorliegenden Beispielimplementation nicht auf direktionale Schattenverfahren, wie sie in Abschnitt 2.3 erläutert wurden, eingegangen werden konnte.

Der praktische Einsatz der Spherical Harmonics

Während bei einführenden Arbeiten zu diesem Thema (wie beispielsweise [SKS02] oder [Bie03]) eine variable Menge von SH-Koeffizienten die Unterschiede in der Repräsentation des Lichtsignals wiedergaben, wird in der vorliegenden Implementation von einer festen Anzahl ausgegangen. Obwohl es für eine flexible Handhabung verschiedenster Beleuchtungsverhältnisse wünschenswert wäre, die Anzahl der verwendeten SH-Bänder an die Umgebung (insbesondere die Größe der Lichtquellen) anzupassen, wurde hauptsächlich aus zwei Gründen davon abgesehen.
Zum einen erfordert eine entsprechend robuste Merkmalsextraktion, wie sie im vorigen Absatz schon angesprochen wurde, einen hohen Berechnungsaufwand und könnte daher nicht mehr flexibel genug gehandhabt werden. Darüber hinaus sind die für eine effektive Erkennung nötigen Algorithmen von hoher Komplexität, so dass sie den Umfang der Arbeit übersteigen würden.
Abgesehen von diesen „weichen“ Faktoren gab es eine grundsätzliche technische Einschränkung. Das verwendete Szenengraphsystem OpenSG begrenzt bei der hier gewählten und weiter oben erläuterten vertexbasierten Anwendung der SH-Beleuchtung die Anzahl der dafür zusätzlich zur Verfügung stehenden Datenfelder pro Vertex auf insgesamt nur 20 `float`-Werte. Folgende Übersicht zeigt die Verwendung der Parameter, die jeweils maximal aus `Vec4f` bestehen können:

Daraus folgt, dass selbst bei Verwendung aller ungenutzten Parameter (z.B. zusätzlich die w-Komponente der Normale) nicht die idealerweise

Parametername	Verwendung	Verfügbarkeit
`GeoPosition`	Position (Objektkoordinaten)	benutzt
`GeoNormal`	Normale	benutzt
`GeoColor`	Farbe mit Selbstverschattung	benutzt
`GeoSecondaryColor`	Zweitfarbe	frei
`GeoTexCoord0`	erste Texturkoordinate	frei
`GeoTexCoord1`	zweite Texturkoordinate	frei
`GeoTexCoord2`	dritte Texturkoordinate	frei
`GeoTexCoord3`	vierte Texturkoordinate	frei

Tabelle 1: Die in OpenSG zur Verfügung stehenden Vertexdaten

eingesetzten 25 SH-Koeffizienten ($\ell = 4$) verwendbar sind. Obwohl wie bereits in Kapitel 2.1.1 auf Seite 21 angesprochen, modernste Grafikhardware auch Texturzugriffe im Vertexshader ermöglicht und somit die beschriebene Beschränkung auf die wenigen Vertexparameter praktisch aufgehoben wäre, stand diese Technologie zum Zeitpunkt dieser Arbeit noch nicht zur Verfügung. Somit ist im vorliegenden System die Anzahl der maximal verwendbaren SH-Koeffizienten auf 16 ($\ell = 3$) begrenzt, was zwar im Allgemeinen ausreicht, aber die Beleuchtungsverhältnisse nicht so optimal annähern kann, wie dies mit 25 Koeffizienten möglich wäre; siehe hierzu auch den leichten Pegelausschlag im Bereich des fünften SH-Bandes in Abbildung 3 auf Seite 13.

6.2 Die Bedienung

Da eine komplette Beschreibung der Beispielimplementation und deren grafische Oberfläche den Umfang dieser Ausarbeitung übersteigen würde, soll im Folgenden auf die wesentlichen Schritte und Konzepte eingegangen werden. Dazu wird an einem typischen Szenario aufgezeigt, wie „sashimi“ zu benutzen ist und welche wesentlichen Verfahren dem System zugrunde liegen. Die Reihenfolge, die stellenweise jedoch beliebig ist, orientiert sich dabei grob an dem in Abschnitt 6.1.1 erläuterten Konzept.

6.2.1 Die Programmoberfläche

Die Bedienoberfläche des Programms teilt sich, wie in Abbildung 13 dargestellt, grundsätzlich in einen großen Viewport, in dem die Szene dargestellt wird und in einen Arbeitsbereich, realisiert durch zwei Registerkarten. Zudem können mit der Toolbox verschiedene Ansichtsmodi sowie grundlegende Funktionen zum Laden von Dateien etc. ausgewählt werden. Auf

einige der Funktionalitäten und Steuerungsmöglichkeiten soll im Folgenden kurz eingegangen werden.

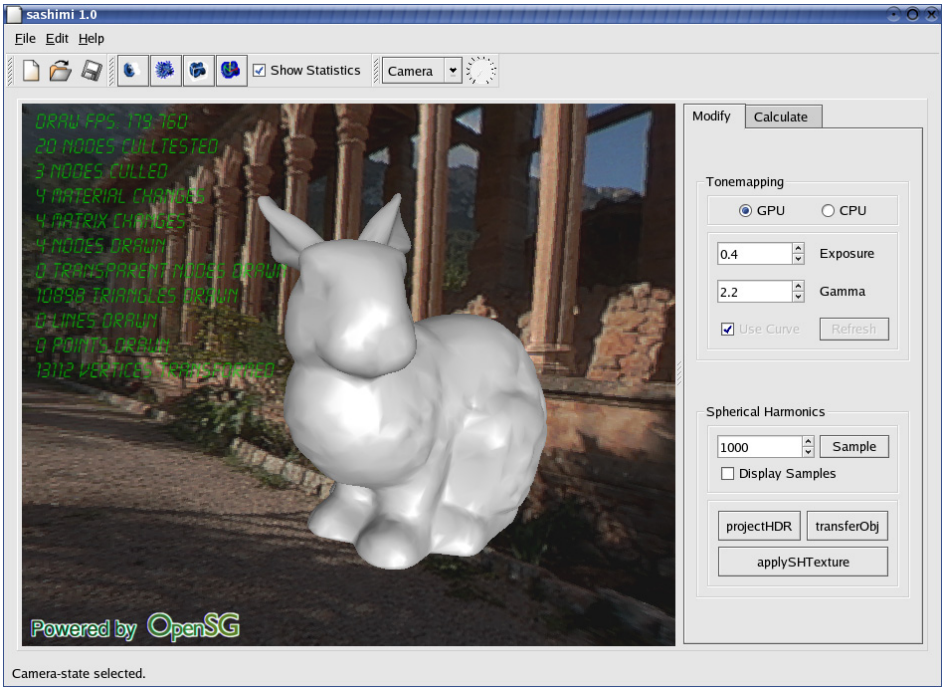

Abbildung 13: Die Programmoberfläche der Beispielimplementation „sashimi“ (Stand: Juni 2004)

6.2.2 Laden von Elementen

Das Beispielprogramm verarbeitet verschiedene Datentypen und ordnet diesen anhand der Dateiendung verschiedene Funktionalitäten zu. Somit ist eine konsistente Möglichkeit gegeben, Komponenten wie Bilder, Objekte und Daten über die gleiche Bedienfolge zu laden. Die Reihenfolge zum Laden der Elemente ist dabei prinzipiell unabhängig voneinander und ermöglicht daher beispielsweise auch das Laden mehrerer Geometrien; lediglich das Laden einer Kamerakurve sollte im Zusammenhang mit einem HDR-Bild geschehen.
Für diese Umgebungsbilder werden die zwei gängigsten Parametrisierungen unterstützt: sowohl das Latlong-Format als auch Cubemaps im EXR-Datenformat können geladen werden. Da das System, wie in Abschnitt 2.2.1 (Seite 26) erläutert, intern mit diesen Cubemaps arbeitet, werden geöffnete Bilder im Latlong-Format beim ersten Laden in Cubemaps konvertiert und als Datei mit dem zusätzlichen Suffix „`_CUBE`“ abgespeichert. Diese sollten

beim nächsten Laden der Bilddatei verwendet werden, da der Ladevorgang dadurch wesentlich beschleunigt werden kann. Zudem werden die Bilder bei ungeeigneter Größe auf die nächste Zweierpotenz *ab*gerundet, da die Bilder der einzelnen Würfelseiten (*engl.:* (cube) faces) als Texturen abgelegt werden. Dabei gilt für die Umrechnung von Latlong in Cubemap folgender Zusammenhang:

$$\text{Breite}_{cubeface} = \lfloor \frac{\text{Breite}_{LatLong}}{4} \rfloor \tag{6.1}$$

Da bei der (standardmäßigen) Verwendung des GPU-basierten Tonemappings aufgrund der Einschränkungen von HDR-Texturen bei der verwendeten Grafikhardware *keine* Interpolationen möglich sind und daher nur `GL_NEAREST` als Filtertyp zur Verfügung steht, sind für eine ausreichende Darstellungsqualität mindestens 512×512 große Texturen pro Würfelseite erforderlich. Dies entspricht 2048×1024 für das Latlong- bzw. 512×3072 für das Cubemapformat, so dass sich der Berechnungsaufwand für diese anfallenden Datenmengen bereits erahnen lässt.

6.2.3 Editieren der Szene

Nachdem die verschiedenen Komponenten der gewünschten Szene geladen sind, stellt sich nun die Frage nach den Möglichkeiten zum Bearbeiten der Objekte. Da meist mehrere Objekte geladen sind und — wie bereits in Abbildung 12 dargestellt — jedes dieser Geometrien unabhängig transformiert werden soll, kann mit Hilfe des sogenannten „pickings“ (Klicken + Leertaste) das gewünschte Objekt selektiert werden; ein umgebender Rahmen (*engl.:* bounding box) gibt dabei entsprechende visuelle Rückmeldung. Für eine Transformation dieses Objektes wird mittels der Auswahlbox in der Werkzeugleiste der Modus der Maussteuerung umgeschaltet und dadurch der in Abschnitt 5.2.3 beschriebene `Trackball` für die Objektmanipulation aktiviert. Nun kann das ausgewählte Element sowohl rotiert (linke Maustaste), in X/Y-Richtung (mittlere Maustaste) als auch entlang der Z-Achse (rechte Maustaste) transliert werden. Die Translation wird dabei im Objektkoordinatensystem durchgeführt, d.h. eine Rotation verändert die Achsen dieses Bezugssystems und damit die Richtung der nachfolgenden Verschiebung. Zusätzlich steuert ein Drehregler die Empfindlichkeit bzw. Schrittweite dieser Bewegung, so dass eine Anpassung an unterschiedliche Objektabmessungen leicht möglich ist.

6.2.4 Vorberechnungen

Nachdem die Szene zunächst entsprechend erstellt und bearbeitet ist, können nun die Vorberechnungsschritte erfolgen. Dabei sind die implemen-

tierten und im Folgenden kurz vorgestellten Techniken — ähnlich dem Vorgängerprogramm „sushi“ — als relativ langwierige Offline-Berechnungen zu verstehen, die im Allgemeinen nur selten durchgeführt werden.

Spherical Harmonics

Wie weiter oben schon erläutert, schränkte die verwendete Plattform die flexible Handhabung der Spherical Harmonics leider etwas ein. Abgesehen von der festen Anzahl der Koeffizienten besteht jedoch keine wesentliche Beeinträchtigung des Umgangs mit dem System z.B. im Vergleich zu [Bie03]. Die einzelnen Schritte zur Initialisierung der SH-Koeffizienten sind dabei weiter modularisiert worden und ermöglichen eine individuelle Anpassung bezüglich Qualität und Verarbeitungszeit.
Dazu werden bei geladener HDR-Umgebung die Anzahl der Samples festgelegt und schließlich berechnet. Dieser Vorgang erzeugt gleichverteilte Samples auf der Einheitskugel, die anhand ihrer sphärischen Koordinaten leicht in kartesische umgerechnet werden können. Mit Hilfe dieser Umrechnung können die OpenEXR-Cubemaps durch 3D-Vektoren sehr komfortabel indiziert werden und bilden die Grundlage für die weiteren Berechnungen. Diese bestehen im Wesentlichen aus der Bestimmung der SH-Basiskoeffizienten, die direkt in den `HDRSamples` abgespeichert werden.
Anschließend können bei Bedarf neue Samples erzeugt werden (z.B. wenn eine Darstellung der Samples deutlich macht, dass mehr Samples erforderlich sind) oder die Bildinformation in die SH-Basis projiziert werden. Dieser Schritt, der aus den theoretischen Grundlagen in Teil I nachvollziehbar sein sollte, resultiert in einer Liste von n^2 RGB-SH-Koeffizienten, die für eine einfachere Weiterverarbeitung als `Vec3f` abgelegt werden und die Lichtfunktion $L(s)$ repräsentieren. Alle bisher durchgeführten Vorgänge führen zu keinem nennenswerten Berechnungsaufwand, da deren Verarbeitungszeiten im einstelligen Sekundenbereich liegen.

Im Gegensatz dazu ist der sich im Allgemeinen anschließende letzte Schritt in der Verarbeitungskette der Spherical Harmonics um einiges aufwändiger. Hierbei werden für jeden Vertex des ausgewählten Objektes dessen SH-Koeffizienten bestimmt und innerhalb der Datenstruktur abgelegt. Wie bereits in Abschnitt 6.1.1 erwähnt, werden die nötigen Informationen zu jedem Eckpunkt (Position und Normale) beim Laden der Geometrie kompakt gesammelt und verwaltet, so dass bei komplexen und sehr modular aufgebauten Objekten deren (Teil-)Szenengraph nicht ständig traversiert werden muss. Lediglich die 16 SH-Koeffizienten werden für die direkte Weiterverwendung beim Rendern in den eigentlichen Geometrieknoten des Objekt-Szenengraphen benötigt und daher als Texturkoordinaten dort abgelegt (siehe Abschnitt 1.3).
Während in anderen Umsetzungen SH-basierter Beleuchtung (z.B. [Gre03],

[SKS02] oder [Bie03]) verschiedene Stufen von Transferfunktionen $T(s)$ realisiert wurden, geht die vorliegende Implementation einen anderen Weg. Ausgehend von der Betrachtung des einfachen, *verschatteten* Transfers — also keine komplexeren Verfahren mit Indirektionen des Lichtweges — ist es möglich, die reine Selbstverschattung des Objektes unabhängig von den Spherical Harmonics zu berechnen. Während bei der Bestimmung der Transferkoeffizienten über den vorderen Halbraum des aktuellen Samples integriert wird,[34] führt eine direkte Einbeziehung der geometrischen Selbstverschattung lediglich zu einer *integrierten* Modulation der Koeffizienten. Um jedoch die gesamte Verarbeitung der Beleuchtungsmodelle der Objekte so modular wie möglich zu halten, wird im vorliegenden System zunächst nur der *unverschattete* Transfer berechnet; die Berücksichtigung der geometrischen Eigenschaften des Objektes werden, wie im Folgenden beschrieben wird, separat berechnet. Die Ergebnisse sind dabei direkt vergleichbar mit den in „sushi" generierten Darstellungen: so kann dort z.B. die reine Selbstverschattung mit Hilfe einer völlig gleichfarbigen (idealerweise weißen) Umgebung simuliert werden, da die SH-Koeffizienten in diesem Fall konstant sind. Zudem ist es durch die Trennung der Ambient Occlusion von den SH-Berechnungen beispielsweise auch möglich, ganz auf die Spherical Harmonics zu verzichten, falls die Beleuchtungssituation aufgrund zu kleiner Lichtquellen dafür ungeeignet ist.
Im Rahmen der technischen Details im nächsten Unterkapitel wird auf die weitere Verarbeitung der erstellten Koeffizienten eingegangen. Insbesondere die Beleuchtung der Objekte im dynamischen Kontext sowie die dafür nötige effiziente Verarbeitung sowohl auf Seiten der Grafikhardware als auf der CPU stehen dabei im Vordergrund.

Ambient Occlusion

In einem weiteren, von den SH-Berechnungen völlig unabhängigem Modul, wird die geometrische Selbstverdeckung eines Objektes vorberechnet. Obwohl aktuellste Entwicklungen zeigen, dass die diese Berechnungen selbst auf relativ komplexen Geometrien ([PG04]) im Sekundenbereich liegen kann, oder mit Hilfe vereinfachter Modelle ([KLA04]) sogar im interaktiven Rahmen liegt, wurde sich hier auf eine feste Vorabberechnung beschränkt. Dies liegt zum einen an der Komplexität, die eine Übertragung der genannten Ansätze auf das vorliegende System sehr aufwändig machen; zum anderen aber auch an Designentscheidungen, die beispielsweise das auf ShadowMapping basierende Verfahren nach [PG04] nicht mit dem gewählten vertexbasierten Ansatz vereinbarem ließen.
Dennoch kann die in „sashimi" implementierte Technik viele andere Herangehensweisen in Bezug auf Geschwindigkeit und Ausnutzung moder-

[34] Aufwändigere Transferberechnungen wie Subsurface Scattering (z.B. [Slo03]) oder ähnliche erfordern hier je nach zugrunde liegendem Verfahren andere Vorgehensweisen.

ner Grafikhardware übertreffen. Dazu wird, wie bereits in Kapitel 2.3.2 technisch im Detail erläutert, aus Sicht jedes einzelnen Eckpunktes der vordere Halbraum gerendert und dabei die gezeichneten Pixel im Verhältnis zur Gesamtauflösung eingesetzt. In Abschnitt 6.3.4 wird noch auf eine eingesetzte Verbesserungsmöglichkeit eingegangen, die die Ungenauigkeiten des Verfahrens verbessern kann. Von Seiten der Bedienung ist der Ablauf vergleichbar mit allen anderen Vorgängen:

- Auswählen des zu bearbeitenden Objektes
- Starten des gewünschten Verfahrens
- Weitere Bearbeitung/Anpassung der Ergebnisse

Abschließend sei im Zusammenhang mit dem zweiten Schritt erwähnt, dass als Einstellungsmöglichkeit bei dem gewählten Verfahren der Öffnungswinkel der Kamera eingestellt werden kann. Dadurch kann bei Verwendung des Standardrenderings (d.h. ohne der unten beschriebenen stereographischen Projektion mittels Vertexshader) stufenlos Einfluss auf die Gewichtung der Selbstverschattung genommen werden; wie aus Abbildung 17 gut hervorgeht, ist dabei jedoch ein Kompromiss zwischen Verschattung und Öffnungswinkel zu treffen.

Parameter-Einstellungen

Anders als die vorangegangenen Ausführungen bezieht sich dieser Abschnitt auf direkt einstellbare Parameter, d.h. die jederzeit und meist auch mit direktem Einfluss auf die Darstellung angepasst werden können.
Hierzu gehören zum einen die weiter unten detailliert beschriebenen Tonemapping-Einstellungen, die standardmäßig mittels Fragmentshader eine Manipulation der HDR-Daten in Echtzeit erlauben. Zum anderen schließen diese Einstellungen aber auch die Anpassung der Materialparameter ein, die selbst wiederum mit Hilfe kombinierter Shaderprogramme unter Ausnutzung moderner, programmierbarer Grafikhardware dargestellt werden können. Wie aus Teil I bereits bekannt sein sollte, bieten sich hierzu insbesondere `uniform`-Parameter an, die eine mächtige aber dennoch effiziente Anpassung des Shaders (im Gegensatz zu den Vertex-Daten selbst) erlauben.

6.3 Technische Details

6.3.1 Cube Mapping

Dieses von Blinn et al. [BN76] entwickelte Verfahren, dass schon sehr lange im Einsatz ist, erzeugt unter der Annahme einer unendlich weit entfernten

Umgebung in sehr effizienter Weise überzeugende Spiegelungseffekte. Dazu wird für den einfallenden Blickwinkel der Reflexionsvektor anhand der Normalen bestimmt und indiziert damit in eine Textur.
Für dieses Environment Mapping existieren verschiedene Parametrisierungen, die in unmittelbarem Zusammenhang mit den in Abschnitt 2.2.1 erläuterten Formatierungen stehen. Insbesondere die Cubemaps bieten viele Vorteile bezüglich der einfachen, hardwarebeschleunigten Verarbeitung und werden daher für die internen Berechnungen der Beispielimplementation verwendet. Dies wird zum einen durch die OpenEXR-API unterstützt, die viele Funktionen sowohl auf Latlong- als auch Cubemaps zur Verfügung stellt. Zum anderen ermöglicht diese Parametrisierung auch die direkte Verwendung der HDR-Daten einzelner Würfelseiten als Textur, da mit der verwendeten Plattform Fließkommatexturen nur in zweidimensionaler Form (und nicht als Cubemap) an die Grafikhardware übergeben werden können. In zukünftigen Generationen von Grafikkarten wird die direkte Unterstützung mit großer Wahrscheinlichkeit realisiert werden, wie aktuelle Entwicklungen (siehe z.B. [NVI04]) zeigen.

Geht man nun noch einen Schritt weiter, kann man diese Technik auch noch wesentlich überzeugender einsetzen, als es mit einer statischen[35] möglich wäre. Das ursprüngliche Verfahren nach Blinn et al. kann keine globalen Änderungen der Szene berücksichtigen, d.h. wenn sich neue Elemente in die Nähe eines spiegelnden Objektes bewegen, werden diese nicht dargestellt — sie existieren schließlich in der umgebenden EM-Textur nicht. Daher liegt es nahe, diese Map *dynamisch* zu erzeugen und damit auch die Umgebung des reflektierenden Objektes zu erfassen; dies wurde auch in der vorliegenden Implementation umgesetzt. Dazu wird nur bei einer sich ändernden Szene[36] eine Cubemap *pro Frame* aus dem Objektursprung heraus erzeugt, wobei das Objekt selbst natürlich nicht enthalten sein darf. Diese dient dann wiederum als Eingabe für die weiteren Verarbeitungsstufen wie dem Fragment(teil-)programm zur Erzeugung der Reflektionseffekte.

6.3.2 Tonemapping per Fragmentshader

Die im vorigen Abschnitt sowie in Kapitel 2.2 (Seite 25) erläuterten Techniken zur Verwendung und Darstellung der geladenen EXR-Cubemaps finden in der Beispielimplementation zweierlei Anwendung. Zum einen ist es ähnlich der in [Bie03] verwendeten Methode möglich, die HDR-Daten der

[35] Statisch bezieht sich hierbei auf den *Inhalt* der Textur, also *was* dargestellt wird; natürlich kann eine solche Darstellung mittels Tonemapping o.ä. „dynamisch“ geändert werden (siehe 6.3.2).

[36] Änderungen, die eine Aktualisierung der Umgebungstextur nicht nötig machen wäre z.B. eine neue Betrachterposition.

Umgebung statisch in ein LDR-Bild zu konvertieren. Diese Textur kann auf Basis einer zusätzlich geladenen Kamerakurve als auch (standardmäßig) per einfacher Gammakurve aus der aktuellen HDR-Skybox erzeugt werden. Der Vorteil dieser Methode ist dabei, dass selbst einfachste Grafikhardware ohne programmierbare Komponenten die Texturen (hardwarebeschleunigt) darstellen kann. Je nach verwendeter Texturauflösung liegt die durchschnittliche Zeit für die Erzeugung der LDR-Texturen im unteren Sekundenbereich, so dass hier keine signifikante Verzögerung in Kauf genommen werden muss.
Als Alternative kann aber auch ein GPU-basiertes Tonemapping verwendet werden, das ebenso eine Kamerakurve oder eine Gammafunktion voraussetzt. Basierend auf dem Verfahren von Kainz et al. [KBH04] werden — unter Ausnutzung des `half`-Datentyps — Lookup-Tabellen vorberechnet und als zusätzliche Texturen dem Fragmentprogramm zur Verfügung gestellt. Diese, für jeden Farbkanal einzeln zu berechnenden Tabellen repräsentieren die aktuelle Abbildungsfunktion von `half`-Daten auf den 8 Bit Wertebereich des Ausgabegerätes. Da es maximal nur 2^{16} verschiedene `half`-Werte gibt, ist diese Vorberechnung überhaupt erst möglich und sinnvoll; eine solche Tabelle für `float`-Daten (2^{32} Werte) wäre dagegen alles andere als praktikabel...

Der erste Ansatz, diese eindimensionale Abbildungsfunktion als normale 1D-Textur zu übergeben, ist jedoch aus zwei Gründen ungünstig. Zum einen unterstützt heutige (und auch zukünftige) Grafikhardware keine Texturen der Dimension 1×65536, obwohl die Datenmenge vergleichsweise gering ist. Zum anderen sind Texturzugriffe auf 1D-Texturen weniger optimiert als zweidimensionale, da letztere die typische Texturform in heutigen Anwendungen darstellen. Daher schlagen Kainz et al. vor, die 2^{16} Werte als 2^{8^2} zu organisieren, d.h. in Form einer 256×256 großen 2D-Textur. Da die Indizierung der Einträge nun nicht mehr so einfach ist wie bei einer eindimensionalen „Abbildung", wird unter Ausnutzung spezieller `pack`- und `unpack`-Funktionen[37] des Fragmentshaders das `half`-Argument in höher- und niederwertige Bits (*engl.:* most/least significant bits) zerlegt und geschickt zusammengesetzt. Somit können von Seiten der Applikation die benötigten Lookup-Tabellen schließlich vorberechnet werden, so dass der Fragmentshader mit Hilfe hochoptimierter (2D)-Texturzugriffe jeden in der HDR-Umgebung vorkommenden `half`-Wert auf 8 Bit abbilden und sehr schnell darstellen kann.
Um die Erzeugung des Tonemappings von der Benutzerseite steuern zu können, verwendet das Fragmentprogramm eine Konstante, die bei Ver-

[37] Diese stehen zur Zeit nur im Cg-Shaderprofil „fp30" zur Verfügung und erfordern daher entsprechende Hardware. Andere Systeme (v.a. ATI-Grafikkarten) bereiten ohnehin bei der Verwendung von `half` im Fragmentshader Probleme; siehe auch 5.1.2, Seite 62

wendung der Kamerakurve der relativen Belichtungszeit entspricht. Dazu werden die RGB-Werte aus der HDR-Textur zunächst mit diesem Wert multipliziert, während die eigentliche Kamerafunktion bereits in den von der CPU vorberechneten Lookup-Texturen enthalten ist. Bei Benutzung der einfachen Gammafunktion von Seiten der Applikation, wird direkt mit Hilfe des Gammawertes das Tonemapping des (ebenfalls per Belichtung skalierten) HDR-Farbwertes durchgeführt. In Abbildung 14 sind die wesentlichen Bestandteile beider Varianten noch einmal aufgeführt.

```
/*
 *  Fragment program for LUT-based tonemapping (see [KBH04])
 */

// subroutine
half lookup( sampler2D lut, half h)
{
    float f = pack_2half( h.xx );
    half4 bytes = unpack_4ubyte( f );

    return h1tex2D( lut, bytes.xy );
}


// the fragment program
void main( in  vp2fp IN, out  fp2fb OUT,

           uniform samplerRECT hdrImg,
           uniform sampler2D redLut,
           uniform sampler2D greenLut,
           uniform sampler2D blueLut,
           uniform float4 hdrControl )      // 1/gamma, exposure, hdrWidth, -
{

    // get half RGBA-pixel (using rescaled texCoords)
    half4 pixel = h4texRECT( hdrImg, IN.texs * hdrControl.z );

    // apply exposure setting to RGB only
    pixel.rgb *= half( hdrControl.y );

    // LUT lookups
    pixel.r = lookup( redLut, pixel.r );
    pixel.g = lookup( greenLut, pixel.g );
    pixel.b = lookup( blueLut, pixel.b );

    // final color
    OUT.color = pixel;
}
```

```
/*
 *  Fragment program for simple gamma-based tonemapping
 */

void main( in  vp2fp IN, out  fp2fb OUT,

           uniform samplerRECT hdrImg,
           uniform float4 hdrControl )      // 1/gamma, exposure, hdrWidth, -
{

    // get half RGBA-pixel (using rescaled texCoords)
    half4 pixel = h4texRECT( hdrImg, IN.texs * hdrControl.z );

    // apply exposure setting to RGB only
    pixel.rgb *= half( hdrControl.y );

    // finally apply the gamma correction
    OUT.color = pow( pixel, half( hdrControl.x ) );

}
```

Abbildung 14: Fragmentprogramme zur Darstellung von HDR-Texturen per Lookup-Tabelle und Gammafunktion

6.3.3 Spherical Harmonics auf der GPU

Wie bereits in den beiden vorangegangenen Teilen der Arbeit erläutert wurde, ist eine Verarbeitung der Spherical Harmonics prinizipiell auf zwei Arten möglich. Während die eine Variante mit Hilfe gleichverteilter Samples die (globale) Lichtfunktion aus den HDR-Bilddaten bestimmt, ermöglicht die in 3.2.2 (Seite 43) besprochene Vorgehensweise eine dynamische Berücksichtigung der näheren Umgebung — inklusive potentieller Lichtquellen. Aufgrund technischer Einschränkungen aktueller Grafikhardware im Zusammenhang mit dem verwendeten Renderingsystem wäre die Umsetzung des zweiten Verfahren mit einem sehr hohen Aufwand verbunden: die fehlende Möglichkeit, die gerenderte Umgebung als *HDR-Daten* für weitere Berechnungen zurückzulesen, würde ein mehrfaches Rendern von LDR-Bildern verschiedener Belichtungszeit erfordern, dass weder unproblematisch noch effizient wäre.
Aus diesem Grund war für die vorliegende Implementation nur die globale Variante realisierbar, die jedoch durch eine umfangreiche Überarbeitung eine erhebliche Erweiterung gegenüber dem Ansatz in [Bie03] darstellt. Hierzu ist auf Basis der Kenntnisse aus Teil I zunächst festzuhalten, dass für eine beschleunigte Berechnung der Spherical Harmonics-Daten nicht pauschal die Grafikhardware geeigneter ist. Insbesondere die Tatsache, dass Vertexprozessoren heutiger Grafikkarten für diese Art von Berechnungen kaum schneller sind als aktuelle CPUs, führt zusammen mit dem vertexbasierten SH-Beleuchtungsansatz zu einer Entscheidung, welche Schritte auf welchen Systemkomponenten sinnvoll sind.
In Abschnitt 1.3.2 (Seite 18) wurde bereits im Detail auf die Rotation der SH-Koeffizienten eingegangen. Die dabei beschriebene Zurückführung der Objekttransformation auf eine ganzheitliche Rotation nur der Umgebung ist hierbei der zentrale Ansatz, der auch durch Szenen aus mehreren Objekten nicht beeinträchtigt wird: lediglich die Kombinationen der Objekt- und Umgebungsmatrix muss dann für jedes Objekt neu durchgeführt werden. Wie jedoch aus der Struktur dieser (resultierenden) Rotationsmatrix der Dimension $n \times n$ (hier: 16×16) leicht erkennbar ist, wäre eine Durchführung dieser Matrix-Vektor-Multiplikation auf der GPU nicht sinnvoll. Zum einen sind selbst bei Ausnutzung aller Eigenschaften dieser Rotationsmatrizen für Shaderprogramme immer noch verhältnismäßig viele Instruktionen notwendig, die im Zusammenhang mit weiteren Techniken die Obergrenze sicher überschreiten würden. Dies liegt insbesondere auch an der Beschränkung der Shader-Bibliotheksfunktionen auf maximal 4×4-Matrizen, die in der „üblichen“ Computergrafik auch vollständig ausreichen.
Zum anderen wäre ein wesentlich höherer Kommunikationsaufwand nötig, da die Informationen über die Änderung der Objekt-/Umgebungsrotation zunächst bei der Applikation (also bei der CPU) ankommen und verarbeitet werden. Zwar wäre die Richtung des Datentransfers von der CPU

zur GPU für die heutige asynchrone Buskommunikation unkritisch, doch die Berechnungen unterliegen ebenfalls der oben genannten Beschränkung auf Computergrafik-typische Formate. Somit ergibt sich zunächst folgende Verteilung, die inhaltlich schon in Kapitel 1.3 erläutert wurden:

- **CPU-Verarbeitung:**
  - Bestimmung der Objekt- bzw. Umgebungsrotation aus der Benutzerinteraktion
  - Erstellung der $n \times n$-Rotationsmatrix[38] R_{SH}
  - Berechnung von $L(s)$, d.h. Projektion der Umgebungssamples in die SH-Basis
  - Berechnung der Transferfunktion $T(s)$ für das Objekt
  - Matrix-Vektor-Multiplikation $L_R(s) = R_{SH} \cdot L(s)$ (pro Frame)
- **GPU-Verarbeitung (Vertexshader):**
  - Berechnung des inneren Produktes („multiply-adds") für die resultierende Farbe $C_{RGB} = L_R(s) \bullet T(s)$
  - Skalierung durch Benutzerparameter
  - Modulation durch den Sichtbarkeitswert

Diese Aufteilung hat zwei große Vorteile: zum einen können die Transformationsdaten der Objekte (Umgebung) „vor Ort" verwertet werden und unterliegen nicht den Einschränkungen des GPU-Programmierparadigmas. Zum anderen sind gerade die parallelen Verarbeitungseinheiten moderner Grafikprozessoren für Skalarprodukte oder ähnliche Operationen geeignet, vor allem weil die gleichzeitige Verarbeitung der drei Farbkomponenten der (rotierten) Lichtkoeffizienten $L_R(s)$ durch das SIMD-Konzept vorgesehen ist und daher keinen zusätzlichen Berechnungsaufwand einführt.
Mit der erweiterten Cg-Schnittstelle (siehe 5.2.2) kann dabei insbesondere bei den als `uniform`-Parameter bereitgestellten Beleuchtungskoeffizienten profitiert werden. Wie aus dem Codeausschnitt in Abbildung 15 hervorgeht, können die RGB-Daten direkt als 3×4-Matrizen für die Multiplikation mit dem Transfervektor eingesetzt werden und erfordern daher weniger Instruktionen im Vertexprogramm.

6.3.4 Stereographische Projektion

Wie weiter oben schon beschrieben wurde, lässt sich das verwendete Verfahren zur Bestimmung der Selbstverschattung eines Objektes noch verbessern. Mit zunehmendem Öffnungswinkel führt die einfache Abbildung, die

[38] n bezeichnet hier die Anzahl der SH-Koeffizienten; im vorliegenden Programm also 16

```
// data from application to vertex program
struct a2vp
{
    float4  position    :   POSITION;
    float4  sh0_3       :   TEXCOORD0;
    float4  sh4_7       :   TEXCOORD1;
    float4  sh8_11      :   TEXCOORD2;
    float4  sh12_15     :   TEXCOORD3;
    float4  color_vis   :   COLOR;
};


// data from vertex program to fragment program
struct vp2fp
{
    float4  combi_color :   TEXCOORD0;
    float4  position    :   POSITION;
};


// the vertex program
void main( in  a2vp IN, out  vp2fp OUT,

           uniform float4x4 lightSH0_3,     // RGB matrices of lightfunction-coeffs.
           uniform float4x4 lightSH4_7,     //
           uniform float4x4 lightSH8_11,    // Actually, only 3x4 data is used:
           uniform float4x4 lightSH12_15,   // rows: r,g,b; cols: coeff i-j
           uniform float4 control    )      // multipliers: Vec4f( visscale, shscale, -, - )
{


    // get the matrix for transformation
    float4x4 ModelViewProj = glstate.matrix.mvp;

    // transform position into screen space
    OUT.position = mul( ModelViewProj, IN.position );


    /*** multiply-adds using matrix-vector-products ***/

    // coefficients 0-3
    // reduce current matrix for multiplication
    float3x4 lightMat = (float3x4) lightSH0_3;
    // multiply-add
    float3 shCol = mul( lightMat, IN.sh0_3 );

    // coefficients 4-7
    lightMat = (float3x4) lightSH4_7;
    shCol += mul( lightMat, IN.sh4_7 );

    // coefficients 8-11
    lightMat = (float3x4) lightSH8_11;
    shCol += mul( lightMat, IN.sh8_11 );

    // coefficients 12-15
    lightMat = (float3x4) lightSH12_15;
    shCol += mul( lightMat, IN.sh12_15 );


    /*** set output color ***/

    // scale visibility
    float vis = IN.color_vis.a * control.x;

    // final_color
    OUT.combi_color.rgb = control.y * ( ( shCol * vis ) + ( IN.color_vis.rgb * vis ) );
    OUT.combi_color.a = 1.0;

}
```

Abbildung 15: Vertexprogramm zur Beleuchtung der Vertices auf Basis der Spherical Harmonics (16 Koeffizienten)

dem Single-Plane-Ansatz ([SP94]) entspricht, einen immer größeren Fehler ein, da die Raumwinkel am Rand immer stärker von denen der Kamerablickrichtung abweichen. Um dies zu vermeiden, haben Coombe et al. [CHL03] im Rahmen ihrer Berechnung von Visibilitätsfaktoren auf der GPU eine stereographische Abbildung mittels Vertexshader realisiert. Dabei wird insbesondere die z-Komponente der 3D-Position in Kamerakoordinaten jedes Vertex so transformiert, dass sie einer Projektion des Punktes auf die obere Hemisphäre (bezüglich der Kamera) entspricht. Dies hat zur Folge, dass der Öffnungswinkel der Kamera nicht mehr von direkter Bedeutung ist; es muss nur sichergestellt werden, dass alle Vertices im vorderen Halbraum sichtbar sind und nicht z.B. durch Beschleunigungstechniken (View-Frustum Culling) vorher entfernt werden. Folgende Abbildungen zeigen den Quellcode des zugehörigen Vertexprogramms und eine Gegenüberstellung der Wirkung zur Bestimmung der Selbstverschattung mit und ohne stereographischer Korrektur.

```
/*
 *  Vertex program for projection hemisphere (see [CHL03])
 */

void main(  in a2vp  IN, out vp2fp  OUT,
            uniform float4  nearFar )       // near, far, -, -
{
    // transform 3D-position
    float4x4 modelView = glstate.matrix.modelview[0];
    float4 vertCam = mul( modelView, IN.position );

    // perform stereographic projection
    float3 vertCamNorm = normalize( vertCam.xyz );

    OUT.position.x = vertCamNorm.x;
    OUT.position.y = vertCamNorm.y;
    OUT.position.z = ( -2.0f * vertCam.z - nearFar.y - nearFar.x ) / ( nearFar.y - nearFar.x );
}
```

Abbildung 16: Vertexprogramm zur Transformation der sichtbaren Vertices zur verbesserten Projektion des vorderen Halbraumes.

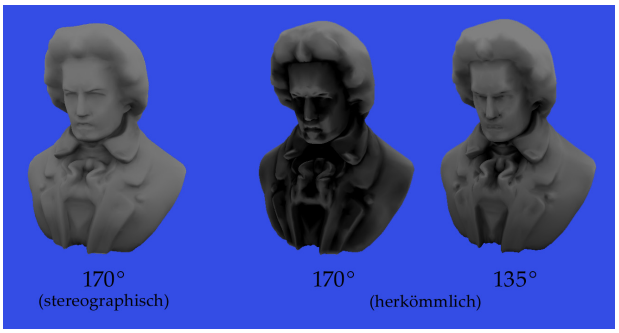

Abbildung 17: Wirkung der stereographischen Projektion nach [CHL03] zur Verbesserung des SinglePlane-Ansatzes.

Abschließend sei noch angemerkt, dass die von den Autoren genannten Nachteile des Verfahrens (geradlinige Rasterisierung gekrümmt abgebildeter Kanten) hier — wenn überhaupt — nur eine untergeordnete Rolle spielen, da die diskretisierten Daten nur zur Auswertung der Anzahl gezeichneter Fragments verwendet werden.

7 Ergebnisse und Bewertung

In diesem letzten Kapitel sollen die praktischen Ergebnisse zusammengefasst und abschließend bewertet werden. Neben konkreten Messwerten zu Berechnungsdauer und/oder Darstellungsgeschwindigkeit sollen einige Bilder aus der Anwendung die besprochenen Aspekte aufzeigen. Im ersten Abschnitt soll die Verarbeitung der HDR-Bilder sowie einfacher Geometrien angesprochen werden, während für den zweiten Teil die von der Volkswagen AG zur Verfügung gestellten Testdaten verarbeitet wurden. In diesem Zusammenhang soll auch auf gewisse Besonderheiten und Probleme eingegangen werden.
Als Testsystem diente ein AMD AthlonXP 1700+ mit 1 GB Arbeitsspeicher, einer NVIDIA GeforceFX 5700 Ultra (NV36) und einer Bildschirmauflösung von 1280x1024 Pixeln. Softwareseitig wurde das System unter Linux (RedHat9, 2.4.20-8) betrieben und verwendete OpenEXR 1.1.0, ein individuell angepasstes OpenSG 1.3.0pre1 sowie Cg 1.2. *(Die angegebenen Frameraten entsprechen* Durchschnittswerten *und müssen nicht mit den in manchen Abbildung angezeigten Momentangeschwindigkeiten übereinstimmen.)*

7.1 Verarbeitung herkömmlicher Daten

7.1.1 Darstellung von HDR-Bildern

Während die Grundlagen zu diesem Thema schon im Laufe der Arbeit geschaffen wurden, sollen hier kurz die Ergebnisse bezüglich der Qualität und der Darstellungsgeschwindigkeit aufgezeigt werden. Prinzipiell unterstützt „sashimi“ die Darstellung der HDR-Bilder per CPU sowie (standardmäßig) mittels Fragmentshader. Je nach Bedarf kann bei individuellen Anpassungen die GPU-Variante zum interaktiven Einstellen der Parameter verwendet werden; die Rendergeschwindigkeit ist dabei aufgrund des intensiven Gebrauchs von Fragmentoperationen von der Füllrate der Grafikkarte begrenzt. Bei dem Testsystem wurde bei dem in Abbildung 13 dargestellten Format eine durchschnittliche Framerate von 70-80 fps erreicht. Dieser auf den ersten Blick gering erscheinende Wert blieb jedoch selbst bei komplexeren Szenen konstant.
Die CPU-Version, die aufgrund des Berechnungsaufwandes (je nach Bildgröße < 3 Sek.) nur per Knopfdruck aktualisiert wird, erreicht aufgrund des Einsatzes einer statischen Textur wesentlich höhere Frameraten; ca. 200-300 Frames waren üblich. Zudem ermöglichte die Unterstützung von Texturfiltern auf diesen LDR-Bildern eine visuell höhere Qualität als ihr HDR-Pendant, dass mit der verwendeten Hardware nicht höherwertig gefiltert werden konnte. Somit stehen dem Benutzer also je nach Bedarf zwei geeignete Optionen zur Auswahl.

7.1.2 Einfache Modelle

Diese Kategorie von Szenen, zu denen insgesamt nicht mehr als 100.000 Vertices[39] gezählt werden stellten insgesamt keine besonderen Ansprüche an das System und konnten ohne weiteres in echtzeitfähigen Geschwindigkeiten dargestellt werden — unabhängig von den verwendeten Materialien. Selbst die vollständige Rotation der SH-Beleuchtung lag für mehrere Objekte oberhalb interaktiver Frameraten, d.h. mindestens 10 fps.
Die Zeiten für die Vorberechnungen waren dagegen sehr stark von den Parametern abhängig und daher sehr unterschiedlich; im Folgenden sind einige Beispieldaten aufgeführt (alle Angaben in Sekunden):

Modell (Vertices)	AO (Öffnungswinkel)	SH (Samples)
„harley“ (68k)	70 sec. (135°)	90 sec. (5k)
„beethoven“ (29k)	12 sec. (170°)	50 sec. (10k)
„bunny“ (10k)	5 sec. (135°)	2 sec. (1k)

Tabelle 2: Exemplarische Vorberechnungszeiten für einfache BeispielModelle

Während das Ergebnis des „beethoven“-Modells bereits in Abbildung 17 ersichtlich ist, soll an dieser Stelle das etwas komplexere „harley“-Modell abgebildet werden.

Obwohl die Berechnungsdauern für die Selbstverschattung nicht mit dem konzeptuell schnelleren Verfahren nach Pharr et al. [PG04] konkurrieren kann (ca. 4 Sekunden für ein 150k Modell), ist es gegenüber seines Vorgängers „sushi“ bedeutend performanter; die reine Ambient Occlusion-Berechnung hätte für die hier verwendeten Modelle im mittleren Minutenbereich gelegen. Die weiter oben angesprochene Verbesserung der Projektion mit Hilfe des Verfahrens nach Coombe et al. [CHL03] führt dabei aufgrund ihrer sehr geringen Belastung der programmierbaren Pipeline nur zu einem geringfügigen zeitlichen Mehraufwand von ca. 10-15 Prozent.
In Kombination mit der Beleuchtungsumgebung und damit der SH-Repräsentation erreicht die Darstellung eine vergleichbare (wenn nicht sogar bessere) Qualität zum Vorgänger. In diesem Zusammenhang sei nochmals darauf hingewiesen, dass im Gegensatz zu „sushi“ mit der vorliegenden Implementation stets eine *vollständige* Rotation der Umgebung und/oder der Objekte unabhängig voneinander möglich ist und kaum zu Abstrichen in der Performance führt.
Zum Abschluss dieses Abschnittes sei noch die vorangegangene Version des „harley“-Modells kombiniert mit der SH-Beleuchtung dargestellt.

[39] Im Folgenden wird die übliche Abkürzung „k“ für Tausend verwendet.

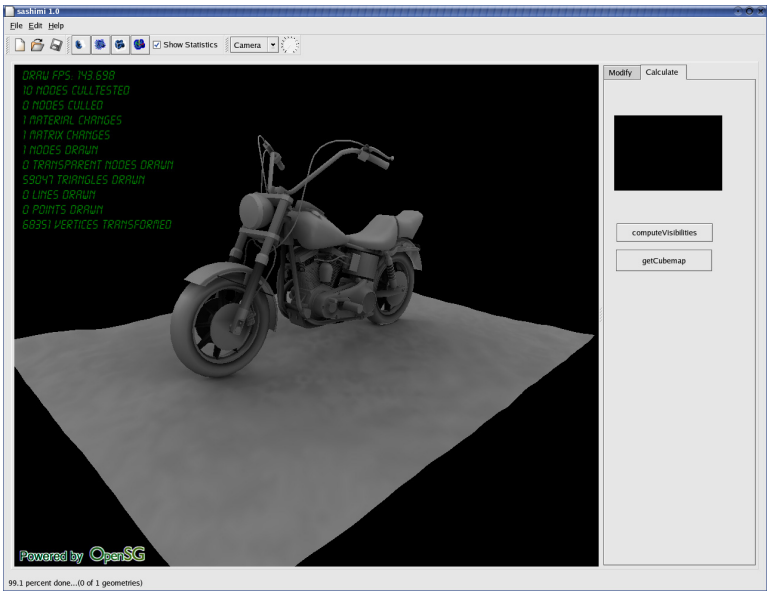

Abbildung 18: Screenshot des mittels vorberechneter Ambient Occlusion dargestellten Modells „harley“.

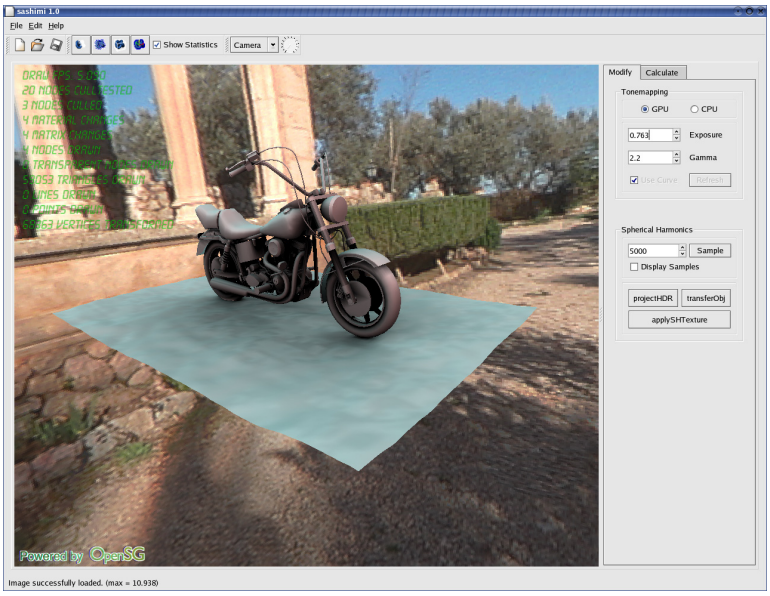

Abbildung 19: Screenshot des mittels vorberechneter Ambient Occlusion und Spherical Harmonics-basierter Beleuchtung kombiniert gerenderten Modells „harley“.

7.2 Der Umgang mit komplexeren Daten

Ein weiteres Anliegen der Arbeit war der Einsatz realistischer Daten(mengen) anhand konkreter Beispielgeometrien. Damit dies nicht durch synthetische Testszenen oder durch eine Vervielfachung vorhandener Modelle geschehen musste, wurde von der Volkswagen AG (insbesondere Hr. Steffen Haupt und Hr. Frank Purschke) freundlicherweise ein Exterieurmodell eines VW Polo zur Verfügung gestellt. Diese modular aufgebaute, polygonale Beschreibung wurde mit einem Editor[40] zu einem Modell handhabbarer Größe und Komplexität zusammengesetzt und liegt bei den hier verwendeten Versionen bei gut 200k Vertices. Natürlich wäre eine Erweiterung auf noch größere Modelle prinzipiell möglich und wünschenswert; die Praxis hat jedoch gezeigt, dass gewisse Konzepte umgestaltet werden müssen, um mit den zusätzlich erforderlichen Datenmengen zurecht zu kommen.
Auch hier soll zunächst eine Übersicht über die gemessenen Werte und technischen Daten gegeben werden, bevor einige Screenshots zu einem kurzen Resümee überleiten:

Modell	**AO (Öffnungswinkel)**	**SH (Samples)**
Polo (198k)	55/317 sec. (150°)	460 sec. (10k)

Tabelle 3: Vorberechnungszeiten des VW Polo Karosserie-Modells.

Erstaunlicherweise erfolgt die reine Berechnung der Visibilitätswerte (erster Wert) sehr schnell; auf dem verwendeten System benötigte jedoch die vertexbasierte Abspeicherung und die damit im Allgemeinen verbundene Allokation von sehr vielen Daten (`Color4f` pro Vertex) den größten Anteil der Gesamtzeit (zweiter Wert).
Im Folgenden zeigen verschiedene Abbildungen das Polo-Modell in verschiedenen Varianten kombiniert mit (dynamischen) EnvironmentMapping, das mit mindestens 15 fps noch durchaus performant dargestellt werden kann.

7.3 Resümee

Wie man insbesondere bei dem Datenmodell höherer Komplexität sehen konnte, können aufwändige Verfahren zur realistischen Darstellung zu Problemen führen, die bei kleineren Szenarien nicht oder nur kaum auftreten. Als erste Konsequenz ließe sich daraus ableiten, dass man entweder mit dem texturbasierten Verfahren einen anderen, von der geometrischen Komplexität der Szene unabhängigen Ansatz wählt und dadurch jedoch

[40] „VRed“ – Virtual Reality Editor; `http://www.vred.org`

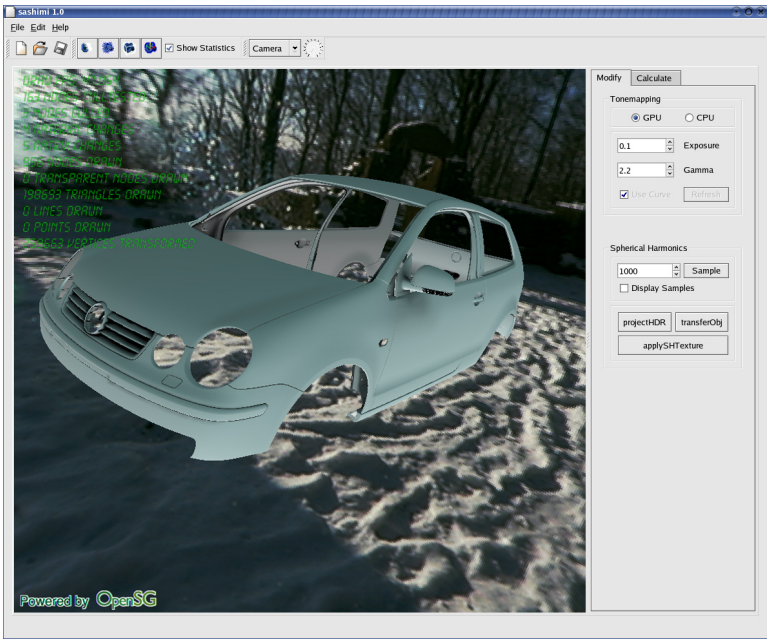

Abbildung 20: Screenshot des Polomodells, dass mittels vorberechneter Ambient Occlusion und Spherical Harmonics-basierter Beleuchtung kombiniert gerendert wird.

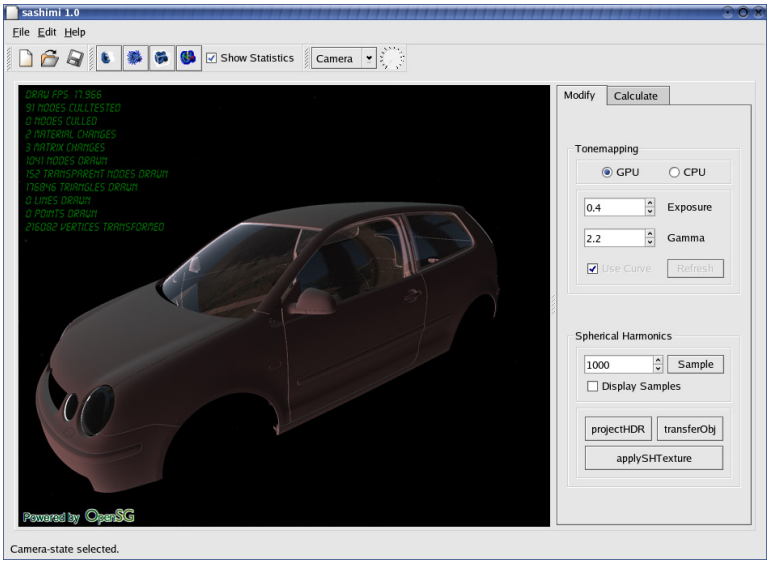

Abbildung 21: Screenshot des Polomodells, dass mittels vorberechneter Spherical Harmonics-basierter Beleuchtung und einem Fresnelterm für transparente Bereiche gerendert wird.

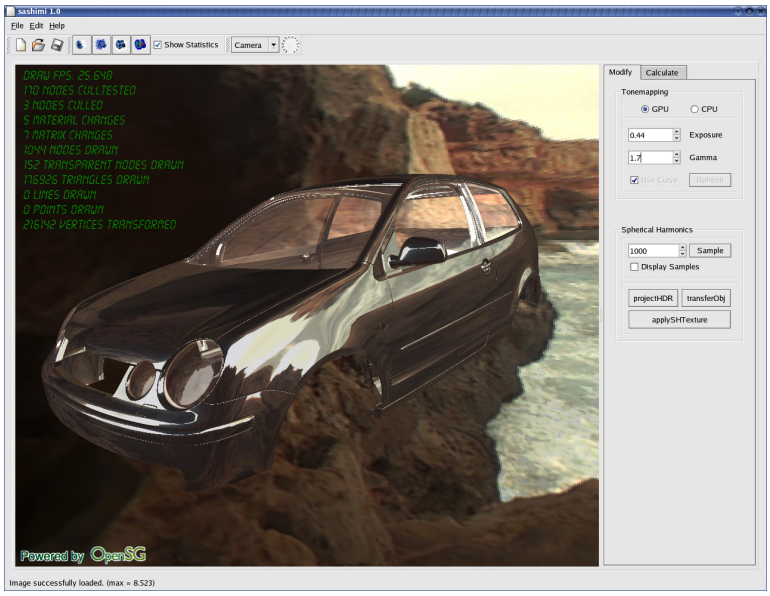

Abbildung 22: Screenshot des Polomodells, das mit einer dynamisch erzeugten Cubemap gerendert wird.

eine ganz andere Art der Berechnung und zugehörigen Fragestellungen betritt; exemplarisch sei hier auf die Problematik der Texturkoordinaten hingewiesen (siehe dazu auch den Ausblick).
Alternativ ließe sich auch eine effizientere Datenhaltung der per-vertex-Informationen anstreben, die aber mit der nächsten Generation von Grafikhardware (vor allem VertexShader 3.0) sowieso möglich ist — letztlich würde dies auch wieder in einen „textur“-basierten Ansatz münden, der jedoch anders als heutige Texturen auch vom Vertexprogramm verwendet werden kann und daher bequem beliebig viele Daten pro Vertex zur Verfügung hat.

Zusammenfassung und Ausblick

Wie aus den vorangegangenen Kapiteln hervorgegangen ist, kann die Realisierung eines hybriden Ansatzes zur realistischen Beleuchtung virtueller Szenen sehr vielseitig und komplex werden. Zu einem solchen System gehört neben den eigentlichen Konzepten, die in der vorliegenden Arbeit unter anderem durch moderne Verfahren wie Spherical Harmonics oder Ambient Occlusion umgesetzt wurden, auch eine Reihe anderer Aufgaben. Dies reicht von dem selbst recht umfangreichen Thema der Verarbeitung und Darstellung von HDR-Bildern über die möglichst intuitive und flexible Benutzerführung bis hin zu der Maßgabe, alle Vorgänge möglichst effizient zu halten. Während für die beiden erstgenannten Aspekte mit OpenEXR und OpenSG zwei sehr moderne und mächtige Systeme zur Verfügung standen, war für eine ausreichende Performance und Echzeitfähigkeit zudem der Einsatz von Grafikhardware unabdingbar. Auch hier konnte die gewählte Plattform Cg in Verbindung mit der guten und ausreichend erweiterbaren Anbindung an das Renderingsystem mehr als überzeugen.
Für den theoretischen Aspekt der Arbeit wurde zunächst eine Gliederung der verschiedenen Formen von Dynamik entworfen. Anhand dieser Unterteilung wurden bestehende Ansätze evaluiert und auf ihre Eignung für den Einsatz in der Beispielimplementation untersucht. Obwohl der Fokus dabei auf Spherical Harmonics basierten Verfahren lag, wurde beispielsweise mit der Ambient Occlusion ein davon prinzipiell unabhängiges, aber sehr gut kombinierbares Verfahren in effizienter Weise eingesetzt. Zudem zeigen aktuelle Konzepte wie [KLA04], dass diese Selbstverschattung zum einen performant berechnet werden kann und zum anderen mit verhältnismäßig einfachen Mitteln einen hohen Grad an Realismus im Kontext der Echtzeitgrafik erreicht.

Obwohl im Rahmen dieser Arbeit zahlreiche Techniken erörtert wurden und einige davon in der Beispielimplementation „sashimi“ zum Einsatz kamen, gibt es natürlich zahlreiche Ansätze für Erweiterungen oder Verbesserungen des Systems.

Zum einen wurden aufgrund der Entscheidung, die wesentlichen Bestandteile Spherical Harmonics und Ambient Occlusion vertexbasiert umzusetzen, gewisse Nachteile in Kauf genommen. Wie jedoch aus den vorangegangenen Kapitel deutlich geworden sein sollte, führt diese Einschränkung bei entsprechenden Modellen, wie sie auch in der Praxis üblich sind, kaum zu deutlichen Qualitätsverlusten. Dennoch wäre auf der Grundlage **parametrischer Geometriedaten**, wie sie häufig im Automobilbau vorkommen eine direkte Umsetzung des pixelbasierten Ansatzes möglich — die benötigten Texturkoordinaten könnten bei Bedarf und zudem wesent-

lich komfortabler generiert werden, als es mit polygonalen Beschreibungen und aufwändigen Mapping-Tools möglich wäre.

Leider wurde an einigen Stellen aber auch deutlich, dass mit der zum Zeitpunkt der Arbeit zur Verfügung stehenden Grafikhardware im Zusammenhang mit dem eingesetzten Renderingsystem nicht alle Konzepte ohne weiteres umgesetzt werden konnten. Die Beschränkung der Anzahl von Spherical Harmonics Koeffizienten sowie die fehlende bzw. unausgereifte Unterstützung von HDR-Rendertargets für die korrekte Berücksichtigung lokaler Lichtquellen seien hier die wichtigsten Aspekte. Wie aber bereits an einigen Stellen erwähnt wurde, bieten neueste (und mittlerweile auch kurz vor der Markteinführung stehende) Entwicklungen die Möglichkeiten zu einer umfangreichen, **hardwarebasierten Erweiterung der Funktionalität und physikalischen Korrektheit**. Insbesondere die Unterstützung von Texturzugriffen im Vertexprogramm oder die direkte und konsistente Verwendung von Fließkommatexturen auf Basis zukünftiger Technologien, bieten sehr viel Lösungsmöglichkeiten aktueller Probleme.

Weniger im Bereich der Computergrafik angesiedelt und daher auch nicht zentrales Thema dieser Arbeit wäre auch die Einbindung von Bildverarbeitungstechniken zur umfangreichen **Adaption an verschiedene Beleuchtungsverhältnisse**. Dies reicht von der Analyse und Merkmalsextraktion geladener Umgebungsbilder bis hin zu physiologisch basierten Filterungstechniken der erzeugten Bilder, die beispielsweise die menschliche Wahrnehmung bei der Repräsentation von hellen Lichtquellen berücksichtigen. Auch hier befinden sich Technologie und insbesondere die Forschung im Bereich der Bildverarbeitung im Umbruch, der neben den allgemeinen Entwicklungen insbesondere auf das unerschöpfliche Potential moderner Grafikhardware zurückzuführen ist; der zunehmende (hardwaregestützte) Einsatz von High-Dynamic-Range-Farbdaten in vielen Bereichen der Bildverarbeitung ist hierfür ein gutes Beispiel.

Um den Bogen zur Computergrafik wieder zu schließen, wäre mit Hilfe dieser Bildverarbeitungstechniken eine weitere Verbesserung der visuellen Qualität durch den gezielten und **optimierten Einsatz herkömmlicher Schattenverfahren** möglich. In der jüngeren Vergangenheit haben die Fortschritte der Grafikhardware auch hier neue Verfahren besonders zur Erzeugung realistischer Schatten in Echtzeit ermöglicht,[41] was durchaus auch ein Indiz für die Komplexität dieses Themengebietes ist.

[41] Für einen umfangreichen Überblick sei auf [AMH02] verwiesen.

Mit dem Wissen über die Umgebungsbeleuchtung und einer Sammlung verschiedener kombinierbarer Techniken zur korrekten Beleuchtung *und* Verschattung, wie sie im Rahmen zahlreicher anderer Arbeiten der Universität Koblenz realisiert wurden und aktuell erstellt werden, lässt sich die Echtzeitgrafik zunehmend der photorealistischen Computergrafik annähern — und damit letztlich der Realität selbst.

Anhang

Literatur

[AMH02] Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. AK Peters Ltd., 2nd edition, 2002. `http://www.realtimerendering.com`.

[ARBJ03] Sameer Agarwal, Ravi Ramamoorthi, Serge Belongie, and Henrik Wann Jensen. Structured importance sampling of environment maps. *ACM Trans. Graph.*, 22(3):605–612, 2003.

[BB03] Michael Bender and Manfred Brill. *Computergrafik*. Carl Hanser Verlag München Wien, 2003.

[Bie03] Matthias Biedermann. *Effiziente Beleuchtung virtueller Objekte unter realen Lichtverhältnissen mit Hilfe Sphärisch-Harmonischer Funktionen*, 2003. Studienarbeit an der Universität Koblenz-Landau.

[BN76] Jim F. Blinn and M.E. Newell. Texture and reflection in computer generated images. In *Communications of the ACM, vol. 19, no. 10*, pages 542–547, October 1976.

[CHL03] Greg Coombe, Mark J. Harris, and Anselmo Lastra. Radiosity on graphics hardware. *UNC TR03-020*, June 2003. `http://www.cs.unc.edu/~coombe/research/radiosity`.

[CIGR99] Cheol Ho Choi, Joseph Ivanic, Mark S. Gordon, and Klaus Ruedenberg. Rapid and stable determination of rotation matrices between spherical harmonics by direct recursion. In *Journal of Chemical Physics*, pages 8825–8831, November 1999.

[CMS87] B. Cabral, N. Max, and R. Springmeyer. Bidirectional reflection functions from surface bump maps. In *ACM SIGGRAPH Proceedings*, pages 273–281, July 1987.

[DDM03] Cyrille Damez, Kirill Dmitriev, and Karol Myszkowski. State of the art in global illumination for interactive applications and high-quality animations. 22(1):55–77, March 2003.

[Dev02] Kate Devlin. A review of tone reproduction techniques. Technical Report CSTR-02-005, Department of Computer Science, University of Bristol, November 2002.

[DM97] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *ACM SIGGRAPH Conference Proceedings*, pages 369–378, August 1997.

[F+04] Randima Fernando et al. *GPU Gems, Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004.

[Fat03] Kayvon Fatahalian. Real-Time Global Illumination of Deformable Objects, May 2003.
`http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/mjs/ftp/thesis-03/fatahalian.pdf`.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial – The Definite Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.
`http://www.nvidia.com/object/cg_toolkit.html`.

[Gre03] Robin Green. Spherical Harmonic Lighting: The Gritty Details, January 2003.
`http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.html`.

[GWWH03] Nolan Goodnight, Rui Wang, Cliff Woolley, and Greg Humphreys. Interactive time-dependent tone mapping using programmable graphics hardware. In *Proceedings of the 14th Eurographics workshop on Rendering*, pages 26–37. Eurographics Association, 2003.

[HK+02] Drew Hess, Florian Kainz, et al. OpenEXR – Platform independent high dynamic range image format, 2002.
`http://www.openexr.com`.

[IR96] Joseph Ivanic and Klaus Ruedenberg. Rotation Matrices for Spherical Harmonics. Direct Determination by Recursion. In *Journal of Physical Chemistry*, pages 6342–6347, January 1996.

[JF03] Doug L. James and Kayvon Fatahalian. Precomputing interactive dynamic deformable scenes. *ACM Trans. Graph.*, 22(3):879–887, 2003.

[Jäh97] Bernd Jähne. *Digitale Bildverarbeitung, 4. Auflage*. Springer-Verlag Berlin Heidelberg, 1997.

[KBH04] Florian Kainz, Rod Bogart, and Drew Hess. The OpenEXR Image File Format. In Randima Fernando, editor, *GPU Gems, Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 425–444. Addison-Wesley, 2004.

[KLA04] Jan Kautz, Jaakko Lehtinen, and Timo Aila. Hemispherical rasterization for self-shadowing of dynamic objects. In *Proceedings of Eurographics Symposium on Rendering 2004*.

Eurographics Association, 2004. Noch nicht erschienen; siehe auch http://graphics.csail.mit.edu/~kautz/publications/egsr_rast.pdf.

[KSS02] Jan Kautz, Peter-Pike Sloan, and John Snyder. Fast, Arbitrary BRDF Shading for Low-Frequency Lighting Using Spherical Harmonics. In *13th Eurographics Workshop on Rendering*, pages 301–308, 2002.

[Lan02] Hayden Landis. Production-Ready Global Illumination, Course 16 notes. *ACM SIGGRAPH Computer Graphics*, 2002. http://www.renderman.org/RMR/Books/sig02.course16.pdf.gz.

[Mül02] Prof. Dr. Stefan Müller. *Vorlesungsskripte zur Computergrafik und Photorealistischen Computergrafik*, 2002. https://www.uni-koblenz.de/FB4/Institutes/ICV/AGMueller/Teaching.

[NRH03] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. All-Frequency Shadows Using Non-linear Wavelet Lighting Approximation. In *ACM SIGGRAPH Computer Graphics*, pages 376–381, July 2003.

[NVI04] NVIDIA Corporation. *Technical Brief, The GeForce 6 Series of GPUs: High Performance and Quality for Complex Image Effects*, April 2004. http://www.nvidia.com/object/IO_12394.html.

[Ope00a] OpenGL Architecture Review Board. *The Official OpenGL Library. OpenGL Programming Guide & Reference Manual, Version 1.2*. Addison-Wesley, 3rd edition, 2000. http://www.opengl.org.

[Ope00b] OpenSG – Open Source Scenegraph, 2000. http://www.opensg.org.

[Pau02] Prof. Dr. Dietrich Paulus. *Vorlesungsskript zur Bildverarbeitung*, 2002. https://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/Teachings.

[PG04] Matt Pharr and Simon Green. Ambient Occlusion. In Randima Fernando, editor, *GPU Gems, Programming Techniques, Tips, and Tricks for Real-Time Graphics*, pages 279–292. Addison-Wesley, 2004. http://download.nvidia.com/developer/presentations/GDC_2004/GDC_2004_Ambient_Occlusion.pdf.

[PTVF02] William H. Press, Saul A. Teukolsky, William T. Vettering, and Brian P. Flannery. *Numerical Recipes in C++. The Art of Scientific Computing.* Cambridge University Press, 2nd edition, February 2002.
`http://www.nr.com`.

[Qt94] Qt – Platform independent API for C++, 1994.
`http://www.trolltech.com`.

[RH01a] Ravi Ramamoorthi and Pat Hanrahan. An Efficient Representation for Irradiance Environment Maps. In *ACM SIGGRAPH Conference Proceedings*, pages 497–500, 2001.

[RH01b] Ravi Ramamoorthi and Pat Hanrahan. On the relationship between radiance and irradiance: Determining the illumination from images of a convex lambertian object. *Journal of the Optical Society of America*, 2001.

[RH02] Ravi Ramamoorthi and Pat Hanrahan. Frequency Space Environment Map Rendering. In *ACM Transaction on Graphics*, pages 517–526, July 2002.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed Radiance Transfer for Real-Rime Rendering in Dynamic, Low-Frequency Lighting Environments. In *ACM Transactions on Graphics*, pages 527–536, 2002.

[Slo03] Peter-Pike Sloan. Spherical Harmonics Precomputed Radiance Transfer, Microsoft DirectX 9.0 (Summer update), 2003.
`http://www.microsoft.com/directx`.

[SP94] Francois Sillion and Claude Puech. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers, August 1994.

[Str00] Bjarne Stroustrup. *Die C++ Programmiersprache, 4. Auflage*. Addison-Wesley, 2000.

[WAT92] S.H. Westin, J.R. Arvo, and K.E. Torrance. Predicting Reflectance Functions from Complex Surfaces. In *ACM SIGGRAPH Computer Graphics*, pages 255–263, July 1992.