Realistische Darstellung komplexer Szenen mit hardwarebasierten Real-Time-Shadern

Studienarbeit

Universität Rostock, Fachbereich Informatik
Lehrstuhl für Computergraphik

vorgelegt von
Cords, Hilko
geboren am 04.04.1978 in Oldenburg

Gutachter: Prof. Dr.-Ing. H. Schumann
Betreuer: Dipl.-Inf. Th. Nocke

Abgabedatum: 17.10.2003

Problemstellung

Die realistische Szenendarstellung mit Hilfe von Real-Time-Shadern ist ein aktuelles Forschungsgebiet. Ziel der Arbeit ist es, verschiedene Strategien des Real-Time-Shadings zur realistischen Bilddarstellung basierend auf der Graphikkarten-Hardware zu vergleichen.

Im Rahmen einer Literaturrecherche soll ein Überblick über existierende Strategien zur Darstellung von komplexen Beleuchtungsmodellen (Phong, Cock-Torrance, Bank, Ashikin), zur Abbildung von Spiegelungen und Brechungen sowie zur Schattendarstellung mit aktueller Graphikkarten-Hardware gegeben werden, Kriterien zum Benchmarking von der Implementierungen aufgestellt (z.B. Zeitverhalten, Unterstützung von speziellen Effekten, Qualität ...) sowie die Grenzen und Probleme der Verfahren dargestellt werden.

Weiterhin soll eine Testumgebung umgesetzt werden, welche die interaktive Auswahl der oben genannten Verfahren und deren Parameter erlaubt. Hierbei soll auf existierende Implementierungen zurückgegriffen werden, und diese gegebenenfalls um weitere Strategien erweitert werden. Schwerpunkt hierbei liegt auf dem Einsatz von Pixelshadern, Environment- und Shadow-Maps.

Im Rahmen der Arbeit sollen Beispielszenen (Vergleich von In- und Outdoor-Szenen) mittels der umgesetzten Verfahren dargestellt und deren Ergebnis gemäß der aufgestellten Kriterien miteinander verglichen werden. Die praktische Umsetzung der Teilaufgaben soll unter C++/OpenGL/Radeon-GK erfolgen. Ziel ist die Umsetzung einer wiederverwendbaren Klassenbibliothek, welche die verschiedenen Funktionalitäten kapselt. Es wird auf eine gute Quelltext- und Programmdokumentation wertgelegt.

Kurzfassung Die programmierbaren Vertex- und Fragmentshader heutiger Graphikkarten bieten zahlreiche Möglichkeiten zur Realisierung verschiedenster Konzepte in der Echtzeit-Computergraphik. Diese Arbeit stellt Konzepte zur Umsetzung verschiedener Beleuchtungsmodellen (Phong, Banks, Cook-Torrance und Ashikhmin) mit Hilfe von programmierbaren Shadern auf einer per-Pixel Basis vor und analysiert diese im Hinblick auf eine Implementierung. Zudem werden Prinzipien physikalischer Effekte wie Reflektion, Brechung und Schlagschattenwurf im Hinblick auf eine Implementierung auf Basis aktueller Graphikkartenhardware betrachtet. Alle vorgestellten Verfahren wurden im Rahmen dieser Arbeit in weiterverwendbaren Klassen realisiert, um eine Bewertung ihrer Praxistauglichkeit und Umsetzbarkeit vornehmen zu können.

Classification (ACM CCS 1998): I.3.1, I.3.2, I.3.4, I.3.7.

Keywords: Shading, Vertex Shader, Fragment Shader, Phong, Banks, Cook-Torrance, Ashikhmin, Reflection Mapping, Refraction Mapping, Bump Mapping, Glossy Map, Shadowing, Projective Shadowing, Shadowmapping, Soft Shadows

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Die Echtzeit-Computer-Graphik ist aufgrund der stark zunehmenden Leistung verfügbarer Chips einer grossen Veränderung ausgesetzt. So gestatten heutige Graphikkarten neben der Echtzeit-Darstellung einer schnell wachsenden und großen Anzahl von Polygonen die Programmierung der verwendeten Shader. Damit ist der Programmierer nicht mehr an die festverdrahtete Beleuchtungseinheit der Graphikkarten gebunden. Diese gestatteten bisher lediglich eine Beleuchtungsberechnung nach festen Modellen - in der Regel wurde das Phong Modell verwendet. Die Parameter waren dabei zu einem gewissen Grade variabel, sie konnten jedoch nur in einem durch die Graphikkarte fest vorgegebenen Rahmen verwendet werden.

Heutige Graphikkarten bieten mit ihren programmierbaren Vertex- und Fragmentshadern wesentlich felxiblere Möglichkeiten in Bezug auf die Beleuchtungsberechnung aber auch allgemein in Bezug auf die Bilddarstellung.

Diese Arbeit diskutiert einige der neuen Möglichkeiten aktueller Graphikkartenhardware. Der Schwerpunkt liegt dabei auf der Verwendung von programmierbaren Shadern zur Umsetzung verschiedener Beleuchtungsmodelle, Reflektion und Brechung und der realistischen Schlagschattendarstellung. Zudem werden alle hier vorgestellten Verfahren mit Hinblick auf eine Implementierung und die dabei zu beachtenden Besonderheiten betrachtet. Alle hier vorgestellten Konzepte wurden im Rahmen dieser Arbeit beispielhaft auf der Basis eines ATI Radeon 9800 Pro Chips umgesetzt, um die jeweiligen Konzepte in Bezug auf ihre Realisierbarkeit und Praxistauglichkeit bewerten zu können.

Die Arbeit ist wie folgt gegliedert: Im anschliessenden Kapitel werden die grundlegenden Begriffe, Modelle und physikalischen Zusammenhänge erläutert, auf die im weiteren Verlauf der Arbeit aufgebaut wird. In Kapitel 3 werden diese Grundlagen in Modelle mit Blick auf eine Implementierung gefasst. Dabei ist dieses Kapitel noch unabhängig von einer konkreten Implementierung oder etwaigen Graphiklibraries. Die Betrachtung im Rahmen einer konkreten Implementierung schließt sich dann in Kapitel 4 an. Abschliessend folgt ein Ausblick und eine Zusammenfassung der Arbeit.

An dieser Stelle sei auch erwähnt, dass in dieser Arbeit bei einigen Begriffen die englische Bezeichnung gewählt wurde. In diesen Fällen erschien eine Übersetzung nicht sinnvoll, da die jeweiligen Begriffe entweder keine sinnvolle Entsprechnung in der deutschen Sprache besitzen oder aber allgemein in der Computergraphik unter der entsprechenden Bezeichnung verwendet werden.

Kapitel 2

Mathematisch-physikalische Grundlagen

In diesem Kapitel werden die für diese Arbeit grundlegenden Begriffe und Modelle vorgestellt und diskutiert.

Zunächst werden einige formale Definitionen eingeführt, um die Basis zum Verständnis der folgenden mathematischen und physikalischen Modelle zu legen.

Desweiteren werden die vier für diese Arbeit relevanten Beleuchtungsmodelle (Phong-, Banks-, Cook-Torrance- und Ashikhminmodell) vorgestellt und kurz diskutiert. Schliesslich wird eine Einführung in die physikalischen Grundlagen der Reflektion, der Brechung und des Schlagschattens gegeben.

2.1 Definitionen und Begriffe

Mathematische Notation In dieser Arbeit werden mathematische Formeln mit folgender Semantik gesetzt[1]:

Skalare :	$a, b, c, \ldots$
Vektoren :	$\mathbf{x}, \mathbf{y}, \mathbf{z}, \ldots$
Matrizen :	$\mathcal{M}, \mathcal{A} \ldots$
normierte Vektoren :	$\hat{\mathbf{n}}, \hat{\mathbf{h}}, \hat{\mathbf{l}}, \ldots$
Skalarprodukt :	$\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle, \langle \hat{\mathbf{l}}, \hat{\mathbf{h}} \rangle, \ldots$

[1] Damit wird eine weit verbreitete Art der mathematischen Notation verwendet, wie sie auch in zahlreichen physikalischen und mathematischen Veröffentlichungen zu finden und auch in der Computergraphik nicht unüblich ist.

Mathematische Bezeichnungen Die in dieser Arbeit verwendeten Variablen werden wie folgt bezeichnet:

Einfallende Leuchtdichte :	L_i
Abstrahlende Leuchtdichte :	L_o
BRDF :	f
ambienter Reflektionsgrad :	k_a
diffuser Reflektionsgrad :	k_d
spekularer Reflektionsgrad :	k_s
Normale :	$\mathbf{n}$
Tangente :	$\mathbf{t}$
Binormale :	$\mathbf{b}$
Position :	$\mathbf{x}$
Lichtvektor :	$\mathbf{l}$
Kameravektor :	$\mathbf{e}$
Halbvektor :	$\mathbf{h}$
Reflektionsvektor :	$\mathbf{r}$

Tangente und Binormale Die Begriffe Tangente und Binormale werden in dieser Arbeit verwendet, wie es allgemein im Zusammenhang mit programmierbaren Shadern üblich ist: Die Normale definiert eine Tangentialebene an dem jeweiligen Objektoberflächenpunkt. Zwei beliebige nicht kollineare Vektoren, die in der zugehörigen Tangentialebene liegen, werden Tangente und Binormale genannt. Diese werden später, z.B. beim Ashikhmin Modell, verwendet, um inhomogene Materialeigenschaften zu definieren. Zusammen mit der Normalen bilden sie den zugehörigen Tangentenraum (Abbildung 2.1). Dieser wird benötigt, um z.B. beim Bump Mapping (siehe dazu Kapitel 3.1.2.4) Normalenvariationen zu definieren ([19], [2]). In der Regel stehen die Tangente und die Binormale senkrecht aufeinander. Der Begriff Tangentenraum wird in dieser Arbeit in Anlehnung an [2] und in dem im Computergraphikbereich der Vertex- und Fragmentshader allgemein üblichen Zusammenhang verwendet. Dabei ist zu beachten, dass die Bezeichnung Tangentenraum in der Mathematik anders belegt ist.

Fragment vs. Pixel Der im Zusammenhang mit programmierbaren Shadern häufig verwendete Begriff „Fragment“ ist hier gleichzusetzten mit dem Begriff „Pixel“. Einen Unterschied gibt es in diesem Fall nicht.

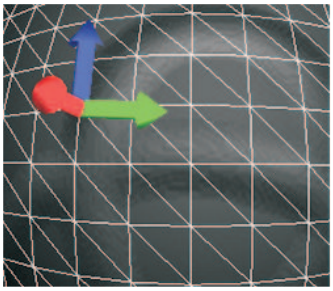

Abb. 2.1: Tangentenraum mit folgenden Axen: Normale (rot), Tangente (grün), Binormale (blau). (Quelle: [2])

Allgemein bezeichnet ein Fragment die jeweiligen Punkte eines Polygons, die auf einer Rasterzeile liegen und mittels Pixel interpoliert werden.

2.2 Beleuchtungsmodelle

Im Folgenden werden die vier in dieser Arbeit als Echtzeitmodelle implementierten Beleuchtungsmodelle vorgestellt. Es wird hier der Begriff Beleuchtungsmodell in Anlehnung an [8] verwendet, um Verwechslungen mit dem Begriff Shadingmodell zu vermeiden. Der Begriff Shadingmodell wird für die konkrete Anwendung eines Beleuchtungsmodells verwendet, ob nun für jeden oder nur einige Pixel[2] hängt von dem jeweiligen Shadingmodell ab.

Alle hier vorgestellten Modelle besitzten Vor- und Nachteile, wie z.B. die Erfüllung der physikalischen Energieerhaltung oder aber die Möglichkeit der Darstellung von Anisotropien. Diese werden für das jeweilige Beleuchtungsmodell kurz diskutiert. Zunächst werden die phänomenologischen Beleuchtugsmodelle Phong und Banks erläutert. Diese sind nicht notwendigerweise an die zugrunde liegenden Physik der Reflektion gebunden - sie nähern die Erscheinung der Reflektionen lediglich an, reichen jedoch für viele Anwendungen aus. Gerade das einfache Phongmodell hat ein große Verbreitung gefunden und ist der „de-facto-Standard“ aller hardwarebeschleunigten Computergraphik.

Des Weiteren werden die beiden physikalisch basierten Beleuch-

[2]In der Regel wird das Beleuchtungsmodell nur auf einige Pixel (Vertexes) angewendet, und alle anderen Farbwerte werden interpoliert [20]. Im Rahmen dieser Arbeit werden unter Verwendung von Vertex- und Fragmentshadern Verfahren vorgestellt, die nahezu für jedes Fragment (anhängig von jew. Texturgrössen) einen eigenen, dem Beleuchtungsmodell entsprechenen Farbwert berechnen.

tungsmodelle Cook-Torrance und Ashikhmin vorgestellt. Diese basieren zumindestens teilweise auf der Physik der Reflektion, erzeugen aber noch kein absolut wirklichkeitsgetreues Abbild der realen Welt. So werden in der Regel Effekte wie Polarisation oder die Reflektionsabhängigkeit von der Wellenlänge des einfallenden Lichtes ausser Acht gelassen oder aber nur approximiert. Da diese Effekte jedoch nur unter bestimmten Rahmenbedingungen wahrnehmbar sind, handelt es sich in den meisten Fällen um keine wesentliche Einschränkung. Nicht zu vergessen ist auch, dass es sich bei allen beschriebenen Verfahren um lokale Beleuchtungsmodelle handelt, so dass globale Effekte, wie z.B. Schlagschatten, noch separat behandelt werden müssen.

2.2.1 Phänomenologische Reflektionsmodelle

2.2.1.1 Phong Modell

Als ein mathematisch einfaches Modell hat das Phong Modell grosse Verbreitung in der Echtzeitcomputergraphik gefunden. Da somit die Prinzipien an vielen Stellen gut dokumentiert zu finden sind (z.B. [8]), folgt hier nur eine kurze Einführung.

Das Phong Modell besteht aus der Summe eines ambienten, eines diffusen und eines spekularen Reflektionsanteils. Mathematisch ist es folgendermassen definiert (siehe dazu auch Abbildung 2.2):

$$L_o(\mathbf{x}, \hat{\mathbf{v}}) = \left(k_a + k_d\langle\hat{\mathbf{n}}, \hat{\mathbf{l}}\rangle + k_s\langle\hat{\mathbf{r}}, \hat{\mathbf{v}}\rangle^q\right)\ L_i(\mathbf{x}, \hat{\mathbf{l}}) \tag{2.1}$$

Eine oft benutzte Variation[3] des klassischen Phong Modells ist das Blinn-Phong Modell. Dieses verwendet statt des Reflektionsvektors den durch einfache Addition und Skalierung zu berechnenden Halbvektor $\hat{\mathbf{h}}$:

$$L_o(\mathbf{x}, \hat{\mathbf{v}}) = \left(k_a + k_d\langle\hat{\mathbf{n}}, \hat{\mathbf{l}}\rangle + k_s\langle\hat{\mathbf{h}}, \hat{\mathbf{n}}\rangle^{q_2}\right)\ L_i(\mathbf{x}, \hat{\mathbf{l}}) \tag{2.2}$$

Dieses Modell wird auch im Rahmen dieser Arbeit genutzt.

Diskussion Das Phong Modell bietet gute spekulare Reflektionsreflexe, ist sehr schnell zu berechnen und sehr anschaulich. Es bietet jedoch keine physikalisch basierte Plausibilität, und es lassen sich nur homogene Materialien

[3] die Berechnung des Reflektionsvektors ist im Vergleich zur Berechnung von $\hat{\mathbf{h}}$ kostenintensiver

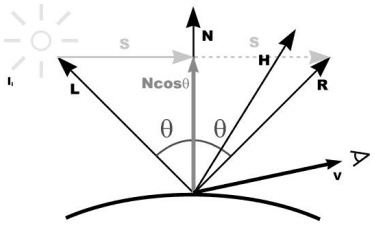

Abb. 2.2: Die für die Beleuchtungsberechnung entscheidenden Vektoren. $\hat{\mathbf{L}}$: Vektor zur Lichtquelle, $\hat{\mathbf{N}}$: Oberflächennormale, $\hat{\mathbf{R}}$: Reflektionsvektor, $\hat{\mathbf{V}}$: Vektor zu Kamerakoordinaten, $\hat{\mathbf{H}}$: Vektor auf Winkelhalbierender von $\hat{\mathbf{L}}$ und $\hat{\mathbf{V}}$. Die hier verwendeten Großbuchstaben entsprechen den jeweiligen Vektoren notiert in Kleinbuchstaben im Text. (Quelle: [2])

darstellen. Es erfüllt damit nicht die physikalischen Eigenschaften der Energieerhaltung und ist auch nicht reziprok[4].

Denoch liefert es im Rahmen dieser Einschränkungen sehr gute und vor allem schnelle Ergebnisse, da es von jeglicher Grafikkartenhardware und diversen Graphiklibraries unterstützt wird. Aus diesem Grunde ist es eines der meist genutzten Beleuchtungsmodelle überhaupt.

2.2.1.2 Banks Modell

Das Banks Modell wurde eingeführt, um auch anisotrope Materialeigenschaften darstellen zu können. Die Grundidee basiert auf der Verwendung von Mikrozylindern. Eine Oberflächentangente $\hat{\mathbf{t}}$ (vgl. Abschnitt 2.1) definiert die Orientierung bzw. die Drehachse des Zylinders (siehe Abbildung 2.4). Diese Orientierung beeinflusst massgeblich die Art der Reflexion.

Mathematisch stellt sich das Banks Modell wie folgt dar [20]:

$$L_o(\mathbf{x}, \hat{\mathbf{v}}) = \left(k_a + k_d \langle \hat{\mathbf{n}'}, \hat{\mathbf{l}} \rangle^p + k_s \langle \hat{\mathbf{v}}, \hat{\mathbf{r}} \rangle^q\right) \cdot \cos\alpha \cdot L_i(\mathbf{x}, \hat{\mathbf{l}}) \tag{2.3}$$

Der Parameter q entspricht dem Phong Exponent und der Exponent p verringert unter anderem die überschüssige diffuse Leuchtstärke, die durch die Verwendung von $\hat{\mathbf{n}'}$ statt $\hat{\mathbf{n}}$ entstehen kann. Es wird ein Wert im Bereich $p \in [1, 10]$ empfohlen. Die Normale $\hat{\mathbf{n}'}$ stellt die Projektion von $\hat{\mathbf{l}}$ auf die von $\hat{\mathbf{n}}$ und $\hat{\mathbf{n}} \times \hat{\mathbf{t}}$ gebildeten Ebene dar [13]. $\cos\alpha$ entspricht dem Vektorprodukt $\langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle$. Die Skalarprodukte können wie folgt berechnet werden [20]:

$$\langle \hat{\mathbf{n}'}, \hat{\mathbf{l}} \rangle = \sqrt{1 - \langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle^2} \tag{2.4}$$

[4] aus den Maxwell Gleichungen hervorgehende physikalische Eigenschaft: $f_r(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = f_r(\mathbf{x}, \hat{\mathbf{l}} \leftarrow \hat{\mathbf{v}})$. Bezeichnet also in diesem Fall die Eigenschaft der Symmetrie der BRDF.

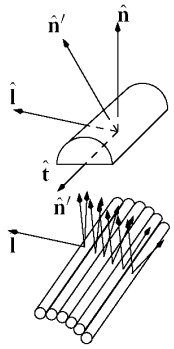

Abb. 2.3: Die Verwendung von Mikrozylindern im Banks Modell. (Quelle: [16])

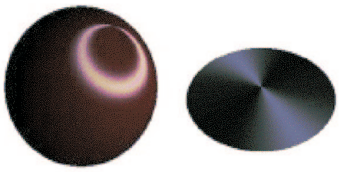

Abb. 2.4: Beispiel der Möglichkeiten zur Darstellung von Anisotropien mit dem Banksmodell. (Quelle: [13])

$$\langle \hat{\mathbf{v}}, \hat{\mathbf{r}} \rangle = \sqrt{1 - \langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle^2} \sqrt{1 - \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle^2} - \langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle \tag{2.5}$$

Ein Aspekt dieser Umformungen, der später interessant wird, sei hier kurz erwähnt: das Banks Modell hängt nur von den zwei Paramtern $\langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle$ und $\langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle$ ab.

Diskussion Das Banks Modell stellt in gewisser Weise die anisotrope Erweiterung des Phong Modells dar. Somit gelten, abgesehen von der Darstellung der Anisotropien, die gleichen Einschränkungen. Ein Beispiel der erweiterten Möglichkeiten ist in Abbildung 2.4 zu sehen. Ein Nachteil des Banks Modells ist, dass nun neben der Normalen auch noch jeweils eine Tangente definiert sein muss. Falls sich diese nicht automatisch erzeugen laesst (haengt von dem jew. anisotropen Modell ab), muss mit erhoehtem Entwicklungsaufwand bei der Erzeugung der Graphikobjekte gerechnet werden. Das Banks Modell ist sehr gut geeignet, um z.B. gefräste Materialien zu simulieren. Eine weitere Anwendung ist das Rendern von Geraden, die als solche keine Volumenausdehnung besitzen, aber mit dem Banksmodell als Aneinanderreihung

von Mikrozylindern aufgefasst werden können. In diesem Fall wird der Faktor $\cos\alpha$ vernachlässigt.

2.2.2 Physikalisch basierte Reflektionsmodelle

2.2.2.1 Cook-Torrance Modell

Das Cook-Torrance Modell ist genau wie das Banks Modell ein Mikrofacettenmodell. Es basiert auf der Annahme, dass Variationen der Reflektivität von Mikrorauhigkeiten der Oberfläche und damit von der zufälligen Anordnung der Mikrofacetten abhängen (vgl. Abbildung 2.5) [6]. Zusätzlich können unterschiedliche Wellenlängen und der Fresnel-Effekt[5] mit Hilfe des Fresnel Termes betrachtet werden. Zusätzlich behandelt das Cook-Torrance Modell Selbstschattierung und Selbstmaskierung [16]. Im Cook-Torrance Modell wird zusammenfassend vorausgesetzt, dass die Variation der Lichtreflektion von folgenden Eigenschaften abhängig ist:

- Wahrscheinlichkeitsverteilung der Orientierung der Mikrofacetten
- Fresnel Term der jew. Mikrofacette
- Schattenwurf/Maskierung der einzelnen Mikrofacetten

Im Folgenden wird die mathematische Definition des Cook-Torrance Modells in Anlehnung an [6] und [20] gegeben. Die mathematischen Hintergründe können dort nachgelesen werden. An dieser Stelle werden nur die für diese Arbeit notwendigen Formeln behandelt. Das Cook-Torrance Modell besteht, wie auch das Phong- und das Banks Modell, aus einem ambienten, einem diffusen und einem spekularen Term:

$$f_r(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = k_a + k_d f_d + k_s f_s(\hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) \tag{2.6}$$

Damit die Energie erhalten wird, muss gelten (Absorption bei Werten kleiner Eins):

$$k_d + k_s \leq 1 \tag{2.7}$$

Der diffuse Term wird als konstant angenommen :

$$f_d = \frac{1}{\pi} \tag{2.8}$$

[5]Die jeweilige Transmission bzw. Reflektion einfallenden Lichtes ist abhängig von dem Winkel, mit dem das Licht auf die Oberfläche trifft [27]. Der Fresnel-Effekt ist wellenlängenabhängig.

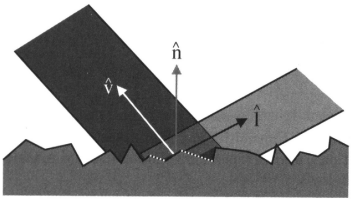

Abb. 2.5: Geometrische Annahmen für das Cook Torrance Modell (Quelle: [20])

Der spekulare Term ist folgendermassen definiert:

$$f_s\left(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}\right) = \frac{1}{\pi} \cdot \frac{F \cdot D \cdot G}{\langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle} \tag{2.9}$$

F repräsentiert die Fresnel-Funktion, D repräsentiert die Verteilungsfunktion der Mikrofacettenorientierung und G repräsentiert die Abschattungs-/Maskierungsfunktion. Diese drei Funktionen hängen jeweils von verschiedenen Paramtern ab, die der Übersichtlichkeit halber nicht mit in die Formel aufgenommen wurden. Diese sind bei ihrer jew. Definition jedoch angegeben:

$$c = \cos\gamma \tag{2.10}$$

$$n = \frac{n_1}{n_2} \tag{2.11}$$

$$g^2 = n^2 + c^2 - 1 \tag{2.12}$$

$$F(c) = \frac{(g-c)^2}{2(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right) \tag{2.13}$$

$\cos\gamma = \langle \hat{\mathbf{v}}, \hat{\mathbf{h}} \rangle = \langle \hat{\mathbf{l}}, \hat{\mathbf{h}} \rangle$ und n_1 , n_2 bezeichnen die jew. Brechungsindizes. Die Wellenlängenabhängigkeit ist durch die Brechungsindexabhängigkeit von der Wellenlänge gegeben. Die Bezeichnung $n(\lambda)$ ist jedoch eher unüblich.

Für die Verteilungsfunktion D wird die Beckmann-Verteilungsfunktion gewählt, wie es für das Cook-Torrance Modell allgemein üblich ist:

$$D(\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle) = \frac{1}{m^2 \langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle^4} \cdot e^{-\frac{1-\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle^2}{\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle^2 m^2}} \tag{2.14}$$

m bezeichnet die Standardabweichung der Ausrichtung der Facetten vom Mittelwert [16].

Für die Abschattungs-/Maskierungsfunktion G wird das Modell, welches in [26] vorgestellt wird, benutzt. Der Vorteil liegt darin, dass es nur von zwei Winkeln abhängt:

$$\begin{aligned} G(\langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle, \langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle) &= G(\cos\alpha, \cos\beta) \\ &= S(\alpha) \cdot S(\beta) \end{aligned} \tag{2.15}$$

mit

$$S(\theta) = \frac{1 - \frac{1}{2}\,\mathrm{erfc}\,(\cot\theta/\sqrt{2}m)}{\Lambda(cot\theta) + 1} \tag{2.16}$$

$$\Lambda(\cot\theta) = \frac{1}{2}\Big(\sqrt{\frac{2}{\pi}} \cdot \frac{m}{\cot\theta} \cdot e^{-\cot^2\theta/2m^2} - \mathrm{erfc}\big(\cot\theta/\sqrt{2}m\big)\Big) \tag{2.17}$$

m besitzt die gleiche Bedeutung wie bei D. erfc(x) bezeichnet die "complementary error function" und ist folgendermassen definiert [4]:

$$\mathrm{erfc}(x) = 1 - \left(\frac{2}{\pi}\right)^{\frac{1}{2}} \cdot \int_0^x e^{-t^2} \cdot dt \tag{2.18}$$

Diskussion Das Cook-Torrance Modell ist physikalisch basiert und bietet Möglichkeiten, um vor allem rauhe Oberflächen zufriedenstellend zu simulieren. Dabei kann eine wellenlängenabhängige Beleuchtung im Rahmen des Fresnel-Effektes vorgenommen werden. Allerdings bietet es keine Unterstützung von anisotropen Materialien, wie etwa das nun folgende Ashikhmin-Modell.

2.2.2.2 Ashikhmin Modell

Das Ashikhmin Modell ist im Grunde genommen eine moderne Version des Blinn-Phong Modells [20]. Es benutzt genau wie dieses den Term $\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle$ zur Berechnung der spekularen Reflexe. Es stellt jedoch eine Erweiterung in Bezug auf physikalische Plausibilität (Energieerhaltung, Reziprokitivität), den Fresnel-Effekt und den Energieaustausch zwischen spekularem und diffusem Reflektionsanteil dar. Zusätzlich unterstützt es die Darstellung von anisotropen Materialien unter Bezugnahme auf die Tangenten und Binormalen und besitzt damit zwei Freiheitsgrade zur Darstellung der Anisotropie[6].

Das Ashikhminmodell basiert ebenso wie die anderen drei vorgestellten Modelle auf der Summe eines ambienten, eines diffusen und eines spekularen Anteils [20]:

$$f(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = k_a + k_d(1 - k_s) f_d(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) + k_s f_s(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) \tag{2.19}$$

Der diffuse Anteil ist blickpunktabhängig:

$$f_d(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = \frac{28}{23\pi} \cdot (1 - (1 - \langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle/2)^5)(1 - (1 - \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle/2)^5) \tag{2.20}$$

[6] zum Vergleich: das Banksmodell benutzt zur Darstellung der Anisotropie lediglich einen Freiheitsgrad (Tangente)

Der spekulare Anteil berechnet sich wie folgt:

$$f_s(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = \frac{\sqrt{(q_t+1)(q_b+1)}}{8\pi} \frac{\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle^{(q_t \langle \hat{\mathbf{h}}, \hat{\mathbf{t}} \rangle^2 + q_v \langle \hat{\mathbf{h}}, \hat{\mathbf{b}} \rangle^2)/(1-\langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle^2)}}{\cos\gamma \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle} F(\cos\gamma) \tag{2.21}$$

Der spekulare Anteil repräsentiert nicht wie z.B. beim Phong Modell einen einfachen Impuls der Spekularen Reflektion. Vielmehr repräsentiert er ähnlich wie beim Bump Mapping einen Blickpunkt- und Oberflächenorientierungsabhängigen Reflektionsanteil.

F steht für die Fresnel-Funktion mit dem Paramter $\cos\gamma = \langle \hat{\mathbf{v}}, \hat{\mathbf{h}} \rangle = \langle \hat{\mathbf{l}}, \hat{\mathbf{h}} \rangle$. Mit den beiden Exponenten q_t und q_b wird die Form des spekularen Reflexes kontrolliert. Mit Hilfe des Faktors $k_d(1-k_s)$ wird die Energieerhaltung sichergestellt. Dabei basiert dieser Ansatz auf dem Fresnel-Effekt: Das nicht spekular reflektierte Licht muss in die Objektoberfläche transmittieren und stellt somit einen zusätzlichen Teil des diffusen Reflektionsanteils. Mit den Paramtern q_t und q_b kann die Form der spekularen Reflektion variiert werden (siehe auch Abb. 3.8).

Der diffuse Anteil kann je nach Anwendung auch als konstant definiert werden. So wird z.B. empfohlen, zur Darstellung von metallischen Oberflächen den diffusen Anteil auf 0 zu setzten.

Diskussion Das Ashikhmin Modell ist ein physikalisch basiertes Beleuchtungsmodell. Es behandelt den Fresnel-Effekt und den Energieaustausch zwischen diffusem und spekularem Beleuchtungsanteil. Zudem kann es Anisotropien darstellen, die mit Hilfe zweier Vektoren in ihrem Erscheinen definiert werden (Binormale und Tangente). Diese erfordern zugleich einen erhöhten Entwicklungsaufwand, da sie einen entscheidenden Faktor für die visuelle Erscheinung des Materials darstellen (vgl. Banks Modell). Gut geeignet ist das Ashikhmin Modell z.B. für metallische Oberflächen.

2.3 Spiegelnde Reflektion

Die physikaliche spiegelnde Reflektion ist mathematisch folgendermassen definiert (siehe auch Abbildung 2.6):

$$\Theta_{in} = \Theta_{out} \tag{2.22}$$

oder anschaulicher:

$$\text{Einfallswinkel} = \text{Ausfallswinkel} \tag{2.23}$$

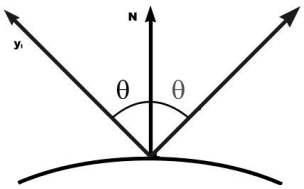

Abb. 2.6: Spiegelnde Reflektion: Einfallswinkel = Ausfallswinkel (Quelle: [22])

In der Anwendung bedeutet dies, dass lediglich eine Oberflächennormale und die Richtung des einfallenden Lichtstrahls benötigt werden, um einen korrekten Reflektionsvektor berechnen zu können.

2.4 Brechung

Lichtbrechung tritt dann auf, wenn ein Lichtstrahl von einem Material in ein anderes Material mit unterschiedlichen Dichten übergeht. Dabei ändert der Lichtstrahl seine Richtung. Die Richtungsänderung wird mit dem Snelliusschen Gesetz beschrieben (siehe dazu auch Abbildung 2.7 und Abbildung 2.8):

$$n_1 \cdot \sin \Theta_{in} = n_2 \cdot \sin \Theta_{out} \tag{2.24}$$

Beim Verlassen des Mediums kommt dieses Gesetzt wieder zum Tragen (vgl. Abbbildung 2.9).

2.5 Schlagschatten

Schlagschatten sind folgendermassen definiert [23]: Der Schlagschatten eines Punktes P auf eine Ebene oder Objekt E ist derjenige Punkt S in welchem der durch P gehende Lichtstrahl das Objekt oder die Fläche E durchdringt. (vgl. Abbildung 2.10).

Alle hinter P und damit im Schattenraum (vgl. Abb. 2.11) liegenden Objekte werden schattiert. Falls ein Objekt nur teilweise im Schattenraum liegt, so ist nur der im Schattenraum befindliche Teil des Objektes schattiert. Dabei ist zu beachten, dass nur die direkt von der Lichtquelle kommende Lichtstrahlung abgeschirmt wird. Eventuelles Umgebungslicht (ambientes Licht) oder Licht von anderen Lichtquellen führt zu den üblichen Reflektions- und Beleuchtungserscheinungen.

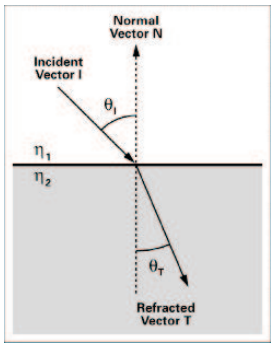

Abb. 2.7: Snelliussches Gesetz: $\eta_1 \cdot \sin\theta_1 = \eta_2 \cdot \sin\theta_2$ (Quelle: [18])

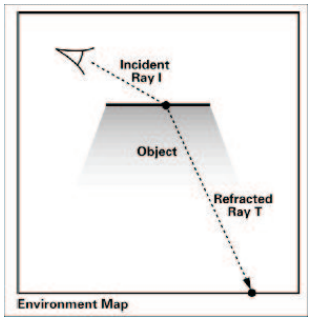

Abb. 2.8: Einfache Brechung (Quelle: [18])

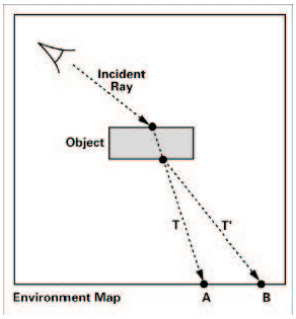

Abb. 2.9: Mehrfache Brechung (Quelle: [18])

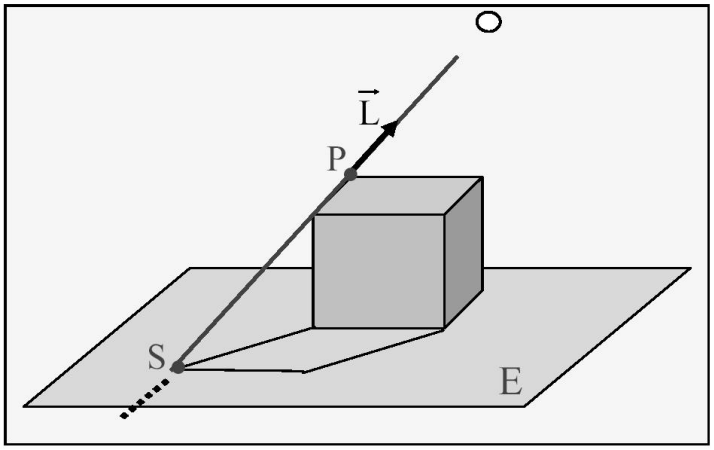

Abb. 2.10: Prinzip des Schlagschattenwurfes (Quelle: [23])

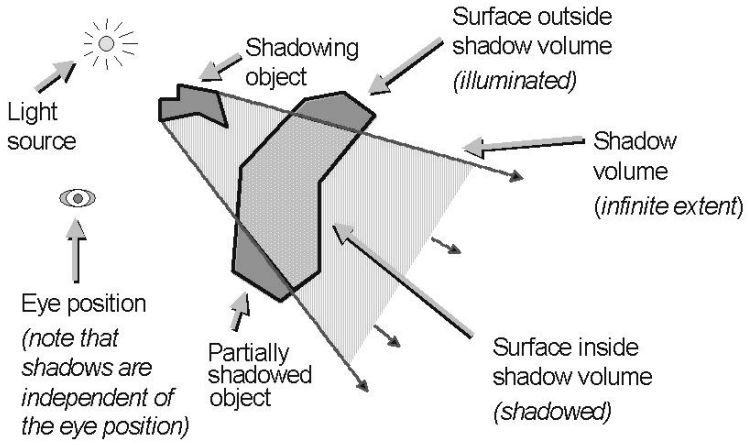

Abb. 2.11: Prinzip des Schlagschattenwurfes: Schattenraum (Quelle: [15])

Kapitel 3

Konzepte zur Realisierung von Beleuchtungsberechnungen und Schattendarstellung mit hardware-basierten Real-Time-Shadern

Dieses Kapitel präsentiert die Konzepte, die für eine Implementierung der im vorhergehenden Kapitel allgemein beschriebenen Modelle und physikalischen Sachverhalte notwendig sind. Dabei ist dieses Kapitel bewusst unabhängig von etwaigen Graphiklibraries und damit von einer konkreten Implementierung gehalten. Die Konzepte gelten im Allgemeinen. Konkrete Hinweise zur Implementierung sind in Kapitel 4 zu finden.

Zunächst werden im ersten Abschnitt die erforderlichen Grundkonzepte vorgestellt, die die Basis für das weitere Verständnis darstellen. Kenntnisse über Texturen, Cubemaps und andere grundlegende Graphikverfahren werden vorausgesetzt. Anschliessend werden die in Kapitel 2 beschriebenen Sachverhalte mit Bezug auf eine Implementierung behandelt.

3.1 Grundkonzepte

3.1.1 Tangentenraum

Der Tangentenraum ist ein Raum, der lokal auf der Oberfläche eines Objektes definiert ist. Aufgespannt wird er von den Basisvektoren[1] Tangente, Binormale und Normale (vgl. dazu auch Abschnitt 2.1 und Abbildung 2.1). Diese können beispielsweise als x,y und z-Achse des Tangentenraumes angesehen werden. Die TBN-Matrix stellt eine Matrix dar, die aus diesen drei Basisvektoren zusammengesetzt ist[2]. Gleichzeitig stellt sie die Matrix dar, die Objektraumkoordinaten in Tangentenraumkoordinaten überführt[3]:

$$\mathcal{M}_{\mathrm{TBN}} = \begin{pmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{pmatrix} \tag{3.1}$$

t_x, t_y und t_z sind dabei die Komponenten des Tangentenvektors und entsprechendes gilt für die anderen Matrizenkomponenten.

Eine Multiplikation mit der TBN-Matrix überführt einen Vektor aus dem Objektraum in den Tangentenraum. Diese Transformation wird z.B. beim Bumpmapping benötigt (siehe dazu Abschnitt 3.1.2.4).

3.1.2 Spezielle Texturen

In der Regel werden Texturen genutzt, um Oberflächen Details hinzuzufügen, die ihre Geometrie nicht besitzt. Dabei können Texturen verschiedene Arten von Details wie z.B. Materialoberflächen aufnehmen. Oftmals werden in Texturen auch globale oder lokale Beleuchtungserscheinungen statischer Lichtquellen vorberechnet (z.B. mit Radiosity oder Raytracing), so dass ein Texturemapping zu einer globalen bzw. lokalen aber statischen Beleuchtung führt[4]. Z.T. werden Texturen auch dynamisch erzeugt, um z.B. bewegte Schlagschatten oder Lichtquellen darstellen zu können. In diesem Fall müssen meist nur ausgewählte Texturen dynamisch erzeugt werden, um den gewünschten Effekt zu erziehlen.

Allen diesen Verfahren ist gemeinsam, dass die Texturen lediglich Farb-(RGBA) oder auch Luminanzinformationen(L) aufnehmen. In den nächsten

[1] damit wird die in der Computergraphik übliche Bezeichnungsweise gewählt - diese entspricht nicht der mathematischen Definition des Tangentenraumes

[2] daher auch die Namensgebung

[3] der mathematische Nachweis dieser Koordinatentransformation ist nicht aufwendig und z.B. nachzulesen in [17]

[4] ein populäres Beispiel ist das Computerspiel Quake

Abschnitten werden Konzepte vorgestellt, in denen Texturen über Farbinformationen hinausgehende Daten speichern.

Allgemein ist bei den folgenden Konzepten zu beachten, dass Texturen heutzutage in der Regel lediglich 8 Bit pro Farbwert zulassen[5]. Diese Einschränkung ist bei der Wahl der abzuspeichernden Informationen immer zu berücksichtigen.

3.1.2.1 Normalmaps

Bei der Verwendung von Normalmaps werden statt der üblichen Farbinformationen Normalen in der Textur gespeichert. Es wird angenommen, dass die Textur im Tangentenraum definiert ist und sich in der von Tangente und Binormale aufgespannten Ebene befindet. Nun kann in jeden Farbwert der Textur ein Vektor, in diesem Fall die Normale, codiert werden.

Vektoren und damit auch Normalen haben allgemein die Form

$$\mathbf{n} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{3.2}$$

Bei normierten Vektoren gilt die Eigenschaft

$$(n_x \cdot n_x + n_y \cdot n_y + n_z \cdot n_z) = 1 \tag{3.3}$$

Somit liegen die einzelnen Komponenten n_x, n_y und n_z jeweils im Bereich $[-1, 1]$. Diese Komponenten werden nun in den Texturfarbinformationen R,G,B gespeichert. Diese Farbkomponenten können Werte im Bereich von $[0, 1]$ aufnehmen. Es ergibt sich also die Umrechnung

$$\begin{aligned} R &= \frac{n_x + 1}{2} \\ G &= \frac{n_y + 1}{2} \\ B &= \frac{n_z + 1}{2} \end{aligned} \tag{3.4}$$

Mit Hilfe dieser Umformung kann jeder beliebige Vektor in der Textur gespeichert werden. Nach dem Auslesen der Texturwerte wird die inverse

[5]Es gibt von nVidia und ATI jeweils Ansätze diese Beschränkung mit Hilfe von Kombinationen von Farbwerten (Hi-Lo-Bits) aufzulösen. Zusätzlich gibt es eigene Tiefentexturen, die Werte bis 24 Bit zulassen, diese sind aber in der Regel an Tiefeninformationen gebunden.

Umformung angewendet, um den ursprünglichen Vektor wieder zu rekonstruieren:

$$\begin{aligned} n_x &= 2 \cdot R - 1 \\ n_y &= 2 \cdot G - 1 \\ n_z &= 2 \cdot B - 1 \end{aligned} \tag{3.5}$$

Die senkrechte Normale $\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ erhält zum Beispiel den Farbwert $(0.5, 0.5, 1.0)$. Aus diesem Grunde haben Normalen Maps in der Regel eine bläuliche Farbe (siehe Abbildung 3.1). Ihren praktischen Nutzten entfalten Normalmaps beim Bumpmapping[6] (siehe Abschnitt 3.1.2.4).

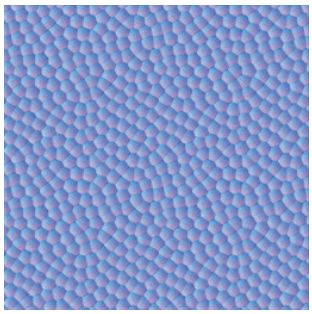

Abb. 3.1: Normalmap: Die Farbwerte enthalten codierte Normalen. Deutlich ist die Blauverschiebung zu erkennen.

3.1.2.2 Depthmaps

Eine Depthmap speichert statt Farbinformationen Tiefenwerte ab. Die Tiefeninformationen einer Szene werden in die Textur gerendert. Somit ist die Textur vergleichbar mit einem kleinen Tiefenpuffer (z-Buffer). Die Depthmap muss nicht notwendigerweise aus der Kameraperspektive erzeugt werden. Üblich ist vorallem ein Rendern aus der Lichtquellenperspektive. Die so entstehende Depthmap wird auch Shadowmap genannt, da sie zur Schlagschattenberechnung mit Hilfe von Shadowmapping (Abschnitt 3.4.2) genutzt werden kann.

3.1.2.3 Funktionen-Maps

Eine weitere für diese Arbeit sehr wichtige Anwendung von Texturen besteht in dem Abspeichern von Funktionen. So können folgende Funktionen in 1-,

[6] in diesem Zusammenhang werden Normalmaps auch Bumpmaps genannt

2- und 3-dimensionalen Texturen abgespeichert werden:

$$\begin{aligned} \text{1D-Textur}: &\quad \mathbf{c} = f(x) \\ \text{2D-Textur}: &\quad \mathbf{c} = f(x,y) \\ \text{3D-Textur}: &\quad \mathbf{c} = f(x,y,z) \\ \text{mit}: &\quad x,y,z \in [0..1] \end{aligned} \tag{3.6}$$

Je nach verwendeter Textur (Luminanz-, RGBA-, RGB-) kann es sich bei **c** um ein Skalar aber auch um einen aus bis zu vier Komponenten bestehenden Vektor handeln.

Die jeweilige Funktion wird vorberechnet und in der Textur abgelegt. Auf die Funktion kann dann ähnlich wie bei einem Feld über die Texturkoordinaten zugegriffen werden. Der Zugriff ist nicht kostenintensiv.

Diese Methode wird genutzt, um relativ komplexe Rechnungen im Fragment-Shader durchzuführen. Dieser ist in der Regel in Bezug auf die Anzahl von Befehlen[7] und auf den Befehlsumfang sehr beschränkt.

3.1.2.4 Bumpmaps

Mit Hilfe von Bumpmapping können planare Oberflächen derart gerendert werden, dass sie nicht mehr planar erscheinen. Der Grundansatz besteht aus einer Variation der Oberflächennormalen. Diese wird z.B. beim Phong-Shading für Oberflächen linear interpoliert. Eine Variation dieser bisherigen linearen oder auch kubischen Interpolation führt im Zusammenhang mit der Beleuchtung zu sehr realistischen

[7] der in dieser Arbeit verwendete Fragmentshader unterstützt beispielsweise lediglich 8 Befehle pro Pass, bei maximal 2 Pässen - also höchtens 16 Befehle

Abb. 3.2: Bumpmapping: Das Quadrat besteht lediglich aus 2 Dreiecken. Dennoch erscheinen sie wie eine rauhe Oberfläche. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `REAL TIME BUMPMAPPING`.

beleuchteten Oberflächenstrukturen. Es werden Strukturen sichtbar, die in der Geometrie des Objektes nicht vorhanden sind (siehe dazu Abbildung 3.2).

Die Variation der Normalen ist in Normalmaps definiert (siehe dazu Abschnitt 3.1.2.1). Die in der Normalenmap gespeicherten Normalen werden zur Beleuchtungsberechnung genutzt. Dabei ist zu beachten, dass die in der Normalenmap gespeicherten Normalen im Tangentenraum definiert sind - also eine senkrecht zur Objektoberfläche stehende Normale unabhängig von der Oberflächenorientierung immer die Form $n = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ besitzt. Aus diesem Grund müssen zur Beleuchtungsberechnung entweder der Kamera- und der Lichtvektor in den Tangentenraum oder umgekehrt die Normale in den Objektraum transformiert werden, um ein mathematisch korrektes Skalarprodukt im gleichen Raum zu gewährleisten. In der Regel ist der Aufwand für die Transformation des Kamera- und des Lichtvektors in den Tangentenraum geringer, da die Transformation mit der TBN-Matrix gegeben ist (Abschnitt 3.1.1). Ein Transformation der Normalen in den Objektraum würde eine Invertierung von M_{TBN} und damit zusätzlichen Aufwand erfordern.

Der Vorteil liegt in der Darstellung von feinen Strukturen, ohne die Anzahl der Polygone einer Szene erhöhen zu müssen. Der jeweilig notwendige Texturzugriff und die anschliessende Umformung der Texturwerte in Normalenform ist nahezu kostenfrei, da entsprechende Befehle in Fragmentshadern zur Verfügung stehen. Dazu ein Beispiel in Abbildung 3.3: Die Anzahl der Dreiecke wurde von 4.000.000 auf 500 verringert. Das anschliessende Bumpmapping lässt das Objekt dennoch nahezu gleichdetailliert erscheinen.

Abschliessend sei noch erwähnt, dass Bumpmapping kein physikalisches Äquivalent besitzt, aber dennoch einen hohen Grad an Realismus erzeugt [1].

3.1.3 Normierende Cube Maps

Mit den im vorhergehenden Abschnitt beschriebenen Methoden und unter Zuhilfenahme von Cubemaps kann ein sehr effektives Verfahren zum Normieren von Vektoren realisiert werden:

Bei einer Cubemap kann die übergebene, aus drei Komponenten bestehende Texturkoordinate als Vektor angesehen werden, der am Ursprung an-

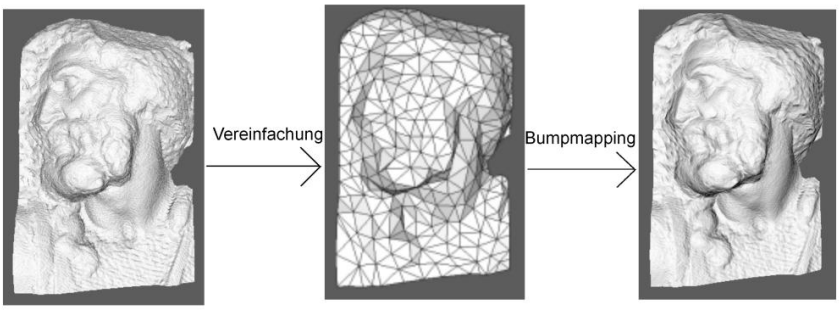

Abb. 3.3: Bumpmapping: Die Geometrie wurde von 4.000.000 auf 500 Dreiecke verringert. Bumpmapping lässt das Objekt dennoch detailliert aussehen. Quelle: [24].

setzt. Zurückgeliefert wird der Farbwert, auf den der Vektor bei Anordnung der sechs Cubemaps um den Ursprung zeigt (vgl. Abbildung 3.4). Die Länge des in Texturkoordinaten übergebenen Vektors ist nicht relevant.

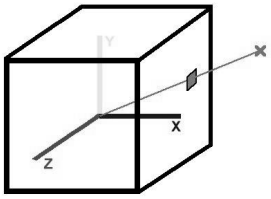

Abb. 3.4: Prinzip einer Cubemap: die übergebene dreikomponentige Texturkoordinate kann als Vektor im Ursprung angesehen werden. Der Farbwert wird über die Richtung dieses Vektors bestimmt.

Wird nun mit Hilfe des im vorhergehenden Abschnittes beschriebenen Verfahrens ein normierter Vektor in die Texturfarbwerte kodiert, der die gleiche Richtung wie die zugehörige Texturkoordinate besitzt, so erhält man über den Texturzugriff den normierten Vektor der übergebenen Texturkoordinate.

Mit Hilfe von normierenden Cube Maps kann ein Vektor mit nur einem einzigen Texturzugriff normiert werden. Zudem ist die Verwendung von normierenden Cube Maps bei vielen Fragmentshadern die einzige Möglichkeit, die beim Übergang vom Vertexshader zum Fragmentshader

interpolierten Vektoren[8] zu normieren. Der im Rahmen dieser Arbeit verwendete Fragmentshader z.B. bietet nicht den Befehlsumfang, um den mathematischen Weg mit $\frac{\mathbf{v}}{|\mathbf{v}|}$ realisieren zu können.

3.2 Beleuchtungsmodelle

In den nächsten Abschnitten werden die in Kapitel 2 vorgestellten Modelle mit den in Abschnitt 3.1 vorgestellten Konzepten in Hinblick auf eine Implementation betrachtet. Konkret wird beschrieben, wie die mathematischen Modelle unter Zuhilfenahme von Funktionenmaps und normierenden Cubemaps innerhalb des Fragmentshaders realisiert werden können. Die jeweiligen Realisierungen sind sich relativ ähnlich, da das Grundprinzip bei allen Shadern darin besteht, die mathematischen Formulierungen mit Hilfe von Funktionenmaps umzusetzten.

3.2.1 Phong

Die in Abschnitt 2.2.1.1 vorgestellte Gleichung 2.2 soll auf einer per-Fragment Basis berechnet werden:

$$L_o(\mathbf{x}, \hat{\mathbf{v}}) = \left(k_d \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle + k_s \langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle^q \right) \; L_i(\mathbf{x}, \hat{\mathbf{l}}) \tag{3.7}$$

k_d und k_s können an dieser Stelle als konstant angesehen werden[9]. Pro Fragment werden die Vektoren $\hat{\mathbf{n}}$, $\hat{\mathbf{l}}$ und $\hat{\mathbf{h}}$ benötigt. Die Berechnung des Skalarproduktes $\langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle$ kann mit dem entsprechenden Befehl[10] direkt im Fragment Shader durchgeführt werden.

Die Berechnung $\langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle^q$ geschieht mit Funktionenmaps, da Fragment Shader in der Regel keinen Befehl zur Berechnung von Exponentialfunktionen bereitstellen. Das Skalarprodukt $\langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle$ liefert lediglich Werte im Bereich $[0, 1]$[11]. Somit könnte prinzipiell eine

[8] in der Regel z.B. Normalen, Tangenten, Lichtquellen- oder Kameravektoren

[9] eine Erweiterung auf per-Fragment-k_d und per-Fragment-k_s kann leicht durchgeführt werden (glossy maps)

[10] alle aktuellen Fragment Shader besitzen einen Befehl zur Berechnung eines Skalarproduktes

[11] Negative Werte ergeben sich lediglich dann, wenn die Normale und damit die Oberflächenorientierung der Lichtquelle oder der Kamera abgewandt ist. In diesem Fall besitzen der spekulare und der diffuse Beleuchtunganteil den Wert 0, können also vernachlässigt werden.

eindimensionale Textur verwendet werden, die die Funktion

$$f(x) = x^q \quad x \in [0,1]; x = \langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle \tag{3.8}$$

berechnet. Wie sich während der Implementierung gezeigt hat, führt dieser Weg zu starken Farb-Aliasing Effekten, da die im 8-Bit Bereich abgespeicherten Normalen und Halbvektoren im Zusammenhang mit hohen Exponenten zu grob abgetastet sind (siehe dazu Abbildung 3.5).

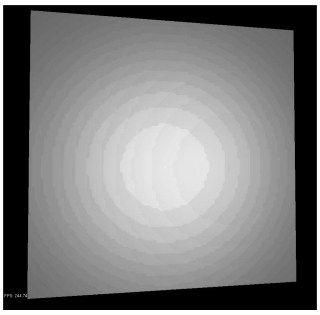

Abb. 3.5: Phong Shading: Die einfache Berechnung von $\langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle^q$ führt zu starken Aliasing Erscheinungen. $k_d = 0.5$, $k_s = 0.5$, $q = 128$. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

Die Verwendung einer zweidimensionalen Textur mit folgender Funktion beseitigt dieses Problem dadurch, dass zur Berechnung der Exponentialfunktion des Skalarproduktes zusätzlich das Skalarprodukt $\langle \mathbf{h}, \mathbf{h} \rangle$ einfliesst:

$$f(x,y) = \min\left(\left(\frac{x}{\sqrt{y}} \right)^q, 1 \right) \quad x = \langle \mathbf{h}, \hat{\mathbf{n}} \rangle; y = \langle \mathbf{h}, \mathbf{h} \rangle \tag{3.9}$$

Ein weiterer Vorteil ist, dass $\mathbf{h}$ nun nicht mehr normiert sein muss, da diese Funktion eine Normierung durchführt. Die Normale $\hat{\mathbf{n}}$ hingegen wird mit einer normierenden Cubemap normiert.

Anschliessend werden im Fragment Shader die jeweiligen Reflektionsgrade k_d und k_s multipliziert, dann der diffuse zum spekularen Anteil addiert und schliesslich mit der einfallenden Leuchtdichte L_i multipliziert.

An dieser Stelle sei nochmal der Vorteil dieses Verfahrens erwähnt: Das vorgestellte Phong-Shading Verfahren ermöglicht eine per-Pixel Beleuchtung mit dem Phongschen Beleuchtungsmodell. Dies ist nicht zu verwechseln mit dem auf aktuellen Graphikkarten unterstützten Phong Modell, bei dem nur für die Vertexes das Phong Modell angewendet wird und die Farbwerte innerhalb eines Polygons interpoliert werden. Das hier vorgestellte Modell ermöglicht z.B. korrekte spekulare Highlights innerhalb eines Polygons (siehe dazu Abbildung 3.6).

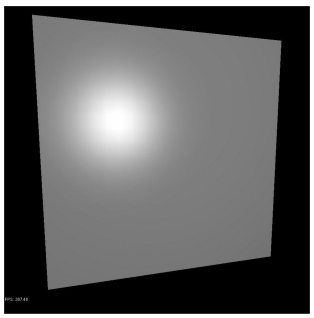

Abb. 3.6: Phong Shading: Obwohl nur zwei Dreiecke gezeichnet werden, wird das spekulare Highlight korrekt gerendert. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `REAL TIME SHADER`.

3.2.2 Banks

Die in Abschnitt 2.2.1.2 vorgestellte Gleichung 2.3 soll auf einer per-Fragment Basis berechnet werden:

$$L_o(\mathbf{x}, \hat{\mathbf{v}}) = \left(k_d \langle \hat{\mathbf{n}'}, \hat{\mathbf{l}} \rangle^p + k_s \langle \hat{\mathbf{v}}, \hat{\mathbf{r}}_{\mathbf{n}'} \rangle^q\right) \cdot \cos\alpha \cdot L_i(\mathbf{x}, \hat{\mathbf{l}}) \tag{3.10}$$

Mit den in Abschnitt 2.2.1.2 vorgestellten Umformungen stellt sich das Banks Modell wie folgt dar [20]:

$$f_d(\langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle) = \sqrt{1 - \langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle^2} \tag{3.11}$$

$$f_s(\langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle, \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle) = \sqrt{1 - \langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle^2}\sqrt{1 - \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle^2} - \langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle \tag{3.12}$$

$$f(\langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle, \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle) = k_d f_d^p(\langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle) + k_s f_s^q(\langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle, \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle) \tag{3.13}$$

$$L_o(\mathbf{x}, \hat{\mathbf{v}}) = f(\langle \hat{\mathbf{l}}, \hat{\mathbf{t}} \rangle, \langle \hat{\mathbf{v}}, \hat{\mathbf{t}} \rangle) \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle \cdot L_i(\mathbf{x}, \hat{\mathbf{l}}) \tag{3.14}$$

Das Skalarprodukt $\langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle$ kann im Fragment Shader direkt berechnet werden. Da f nur von zwei Parametern abhängt, kann sie vorberechnet und in einer zweidimensionalen Textur abgelegt werden. Ein Beispiel wie die Berechnung in C++ aussehen könnte wird im Folgenden gegeben:

```
int BANKS_WIDTH=512;                          // size of texture map

GLubyte *banksTex = (GLubyte *)malloc(sizeof(GLubyte) * BANKS_WIDTH * BANKS_WIDTH);

for (int x = 0; x < BANKS_WIDTH; x++)         // l.t is a scalar in the range [0.0, 1.0]
{
  for (int y = 0; y < BANKS_WIDTH; y++)       // v.t is a scalar in the range [0.0, 1.0]
  {
    float l_dot_t = (((GLfloat)x)/((GLfloat)BANKS_WIDTH-1.0f)); // Range [0,1]
    float v_dot_t = (((GLfloat)y)/((GLfloat)BANKS_WIDTH-1.0f)); // Range [0,1]
```

```
    float f_d = sqrt( 1 - (l_dot_t*l_dot_t) );
    float f_s = f_d * sqrt ( 1 - (v_dot_t*v_dot_t) ) - (l_dot_t*v_dot_t) ;

    float f   = k_d * pow(f_d, p) + k_s * pow (f_s, q);

    f=min(1.0, f);
    f=max(-1.0, f);

    f=f*0.5+0.5;

    banksTex[(y * BANKS_WIDTH) + x] = (GLubyte)(255.0f * f);
  }
}
```

Da die Funktion f Werte im Bereich von [-1,1] annehmen kann, muss noch eine Konvertierung in den Definitionsbereich [0,1] der Textur vorgenommen werden (siehe Listing). In Abbildung 3.7 ist ein Beispiel der von einer Textur aufgenommenen Funktion f dargestellt.

3.2.3 Cook-Torrance

Die Realisierung des Cook-Torrance Modells gestaltet sich prinzipiell nach dem gleichen Schema, welches auch bei dem Phong- bzw. Banks-Modell angewendet wurde. Da das Cook-Torrance Modell mathematisch komplexer ist, werden an dieser Stelle allerdings zwei Funktionenmaps benötigt.

Da der diffuse Anteil des Cook-Torrance Modell als konstant angenommen wird (Abschnitt 2.2.2.1) und seine Berechnung somit trivial ist, sei die Darstellung an dieser Stelle auf den spekularen Anteil beschränkt. Der spekulare Anteil ist im Cook-Torrance Modell definiert mit (vgl. Abschnitt 2.2.2.1):

$$L_{o_s}\left(\mathbf{x}, \hat{\mathbf{l}}\right) = k_s \; f_s(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle \cdot L_i \tag{3.15}$$

$$= k_s \frac{1}{\pi} \frac{F(\cos\gamma) \cdot D(\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle) \cdot G(\langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle, \langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle)}{\langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle} \cdot L_i \tag{3.16}$$

Unter Verwendung von zweidimensionalen Funktionen umgeformt ergibt

Abb. 3.7: Eine zweidimensionale Textur speichert die Banks Funktion f. Parameter: $k_d = 0.5$, $k_s = 0.5$, $p = 10$, $q = 30$

sich [20]:

$$L_{o_s}\left(\mathbf{x}, \hat{\mathbf{l}}\right) = k_s \cdot f(s_f, t_f) \cdot g(s_g, t_g) \cdot L_i \tag{3.17}$$

$$f(s_f, t_f) = F(s_f) \cdot D(t_f) \tag{3.18}$$

$$s_f = \langle \hat{\mathbf{v}}, \hat{\mathbf{h}} \rangle = \langle \hat{\mathbf{l}}, \hat{\mathbf{h}} \rangle = \cos\gamma \tag{3.19}$$

$$t_f = \langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle \tag{3.20}$$

$$g(s_g, t_g) = G(s_g, t_g)/(\pi t_g) \tag{3.21}$$

$$s_g = \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle \tag{3.22}$$

$$t_g = \langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle \tag{3.23}$$

Die Funktionen f und g werden jeweils in einer zweidimensionalen Textur gespeichert. Ihre jeweilige Berechnung wurde bereits in Abschnitt 2.2.2.1 behandelt. Normierungen erfolgen mit normierenden Cubemaps. Die nicht aufwendige Berechnung der abstrahlenden Leuchtdichte erfolgt im Fragment Shader.

3.2.4 Ashikhmin

Auch die Realisierung des Ashikhmin Modells ähnelt denjenigen der vorhergehenden drei Modelle. Zum besseren Verständnis sei die mathematische Beschreibung des Modells in Anlehnung an Abschnitt 2.2.2.2 nochmals gegeben:

$$f(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = k_d(1 - k_s) f_d(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) + k_s f_s(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) \tag{3.24}$$

$$f_d(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = \frac{28}{23\pi} \cdot (1 - (1 - \langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle/2)^5)(1 - (1 - \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle/2)^5) \tag{3.25}$$

$$f_s(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = \frac{\sqrt{(q_t + 1)(q_s + 1)}}{8\pi} \frac{\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle^{(q_t \langle \hat{\mathbf{h}}, \hat{\mathbf{t}} \rangle^2 + q_v \langle \hat{\mathbf{h}}, \hat{\mathbf{s}} \rangle^2)/(1 - \langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle^2)}}{\cos\gamma \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle} F(\cos\gamma) \tag{3.26}$$

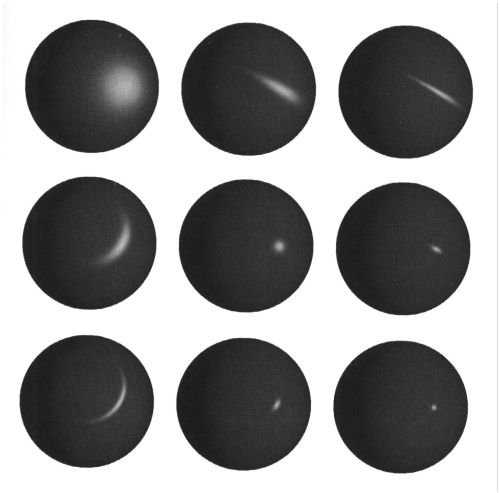

Abb. 3.8: Ashikhmin Modell: Exponenten mit Werten von 10, 200 und 1000 in allen Kombinationen (q_s und q_t). Quelle: [20]

Da der diffuse Term ledliglich von den beiden Parametern $\langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle$ und $\langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle$ abhängt, kann er genau auf diese Weise in einer zweidimensionalen Textur gespeichert werden. Der spekulare Term wird für eine Vorberechung in Texturen folgendermassen umgeformt [20]:

$$f_s(\mathbf{x}, \hat{\mathbf{v}} \leftarrow \hat{\mathbf{l}}) = \frac{\sqrt{(q_t+1)(q_s+1)}}{8\pi} \tag{3.27}$$
$$\cdot \left[\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle^{\frac{q_t \langle \hat{\mathbf{h}}, \hat{\mathbf{t}} \rangle^2}{1-\langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle^2}}\right] \left[\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle^{\frac{q_s \langle \hat{\mathbf{h}}, \hat{\mathbf{s}} \rangle^2}{1-\langle \hat{\mathbf{h}}, \hat{\mathbf{n}} \rangle^2}}\right]$$
$$\cdot \left[\frac{F(\cos\gamma)}{\cos\gamma}\right] \cdot \left[\frac{1}{\max\left(\langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle, \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle\right)}\right]$$
$$= C_a \cdot g_s(\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle, \sqrt{q_t}\langle \hat{\mathbf{t}}, \hat{\mathbf{h}} \rangle) \cdot g_s(\langle \hat{\mathbf{n}}, \hat{\mathbf{h}} \rangle, \sqrt{q_s}\langle \hat{\mathbf{s}}, \hat{\mathbf{h}} \rangle) \tag{3.28}$$
$$\cdot g_f(\cos\gamma) \cdot g_m(\langle \hat{\mathbf{n}}, \hat{\mathbf{v}} \rangle, \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle)$$

mit

$$g_s(u, v) = u^{\left(\frac{v^2}{1-u^2}\right)} \tag{3.29}$$
$$g_f(u) = \frac{F(u)}{u} \tag{3.30}$$
$$g_m(u, v) = \frac{1}{\max(u, v)} \tag{3.31}$$
$$\cos\gamma = \langle \hat{\mathbf{l}}, \hat{\mathbf{h}} \rangle = \langle \hat{\mathbf{v}}, \hat{\mathbf{h}} \rangle \tag{3.32}$$

Diese Art der Darstellung würde drei Texturen (jeweils eine zur Abspeicherung von g_s, g_f und g_m) und zur Berechnung von f_s vier Texturzugriffe benötigen (g_s wird mit zwei unterschiedlichen Parametern aufgerufen). Eine andere, im Rahmen dieser Arbeit gewählte Variante, besteht darin, die Vorfaktoren der Parameter von g_s mit in die Funktion aufzunehmen und zwei verschiedene Texturen mit diesen Funktionen zu erzeugen:

$$g_{s_{q_t}}(u,v) = u^{(\frac{(\sqrt{q_t}v)^2}{1-u^2})} \tag{3.33}$$

$$g_{s_{q_s}}(u,v) = u^{(\frac{(\sqrt{q_s}v)^2}{1-u^2})} \tag{3.34}$$

Diese Variante benötigt eine Textur mehr, besitzt aber den Vorteil, dass die Texturkoordinaten nicht mehr normiert werden müssen, da der jew. Parameter $\langle\hat{\mathbf{h}},\hat{\mathbf{t}}\rangle$ bzw. $\langle\hat{\mathbf{h}},\hat{\mathbf{n}}\rangle$ immer im Bereich $[-1,1]$ liegt. Im Falle der Multiplikation mit $\sqrt{q_t}$ bzw. $\sqrt{q_s}$ würde die Normierung im Fragment Shader durchgeführt werden müssen. Es findet also eine Vergrößerung der Anzahl der benötigten Texturen zugunsten eines veringerten Berechnungsaufwandes innerhalb des Fragment Shaders statt.

Für die Erzeugung der Texturen sollten die Singularitäten der verwendeten Funktionen beachtet werden. So besitzt die Funkion g_s bei $u = \langle\hat{\mathbf{n}},\hat{\mathbf{h}}\rangle = 1$ eine Singularität. Diese kann beseitigt werden, indem die Funktion g_s in der Umgebung von 1 durch ihren dortigen Limes ersetzt wird:

$$\lim_{u\to 1}\left(u^{\frac{v^2}{(1-u^2)}}\right) = e^{\frac{-v^2}{2}} \tag{3.35}$$

Die Fresnel Funktion wird mit Hilfe der Schlickapproximation angenähert:

$$F(\cos\gamma) = k_s + (1-k_s)(1-\cos\gamma)^5 \tag{3.36}$$

Rendering-Beispiele zum Ashikhmin Modell werden in Abbildung 3.8 gezeigt.

3.3 Reflektion und Brechung

3.3.1 Diffuse Reflektion

Diffuse Reflektion tritt vor allem bei matten oder stumpfen Oberflächen auf. Derartige Oberflächen erscheinen mit gleicher Helligkeit unter verschiedenen Blickwinkeln, da sie das Licht mit gleicher Intensität in alle Richtungen reflektieren. Die Berechnung dieser blickpunktunabhängigen Beleuchtungserscheinung erfolgt im Idealfall nach dem Lambertschen Gesetz:

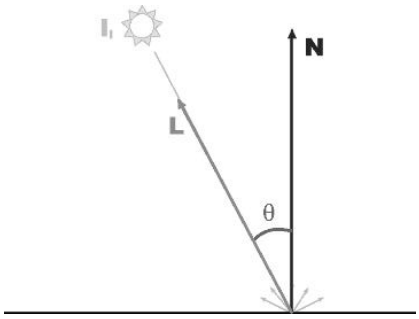

Abb. 3.9: Berechnung der diffusen Reflektion. Quelle: [22]

$$L_{o_{diff}} = k_d \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle \cdot L_i \tag{3.37}$$

Dieses Gesetz kommt z.B. beim Phongschen Beleuchtungsmodell zur Berechnung des ideal diffus reflektierten Lichtanteils zum Tragen.

3.3.2 Spiegelnde Reflektion

Spiegelnde Reflektion wird in der Echtzeitcomputergraphik mit Hilfe von Cubemaps realisiert. Der Reflektionsvektor wird mit Hilfe der in Abschnitt 2.3 besprochenen Gleichung

$$\text{Einfallswinkel} = \text{Ausfallswinkel} \tag{3.38}$$

berechnet. Der derart berechnete Reflektionsvektor wird als Texturkoordinate für die Cubemap benutzt, die die zu reflektierenden Farbinformationen beinhaltet. Somit handelt es sich nicht um eine berechnete Reflektion der dargestellten Szene, wie es z.B. beim Raytracing der Fall ist. Wird die Cubemap jedoch durch Rendern der Szene aus Sicht des reflektierenden Objektes erzeugt, ergibt sich eine korrekte Reflektion der Szene. Mehrfache spiegelnde Reflektion kann durch Multipassrendering beim Erzeugen der Cubemaps realisiert werden. Wird die Cubemap dynamisch erzeugt, so können auch bewegte Objekte auf Kosten der Framerate korrekt spiegelnd dargestellt werden.

Die Berechnung eines Reflektionsvektors nur unter Verwendung von Vektoren wird nachfolgend gegeben (siehe auch Abbildung 3.10):

$$\begin{aligned} \hat{\mathbf{r}} &= \hat{\mathbf{n}} \cos\theta + \hat{\mathbf{s}} \\ &= 2\hat{\mathbf{n}} \cos\theta - \hat{\mathbf{l}} \\ &= 2\hat{\mathbf{n}} \cdot \langle \hat{\mathbf{n}}, \hat{\mathbf{l}} \rangle - \hat{\mathbf{l}} \end{aligned} \tag{3.39}$$

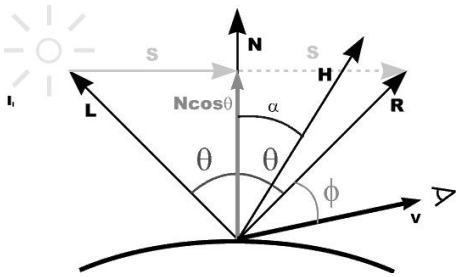

Abb. 3.10: Berechnung des Reflektionsvektors. Quelle: [22]

Wird nun statt des Lichtvektors der Kameravektor verwandt, ergibt sich der blickpunktabhängige spiegelnde Reflektionsvektor.

3.3.3 Brechung

Die Berechnung des Brechungsvektors gestaltet sich ähnlich wie die Berechnung des Reflektionsvektors im vorhergehenden Abschnitt. Es wird wie bei der spiegelnden Reflektion eine Cubemap genutzt, die die Farbinformation für den zugehörigen Brechungsvektor beinhaltet. Auch bei der Brechung bietet es sich an, die Cubemap, wie bei der Reflektion bereits beschrieben, durch rendern aus der Objektperspektive zu erzeugen. Damit wird eine realistisch aussehende Brechung erzeugt.

Der Brechungsvektor wird an dem ersten auftretenden Material- und damit Brechungsindexwechsel bestimmt. Weitere in der Realität auftretenden Brechungen (z.B. beim Verlassen des Mediums) werden vernachlässigt, so dass es sich nicht um eine physikalisch korrekte Brechung handelt. Dies ist eine in Bezug auf physikalische Korrektheit grosse Einschränkung, doch die damit verbundene mathematische Fehlerhaftigkeit der Brechung wird dem Betrachter insbesondere auch bei animierten Szenen kaum auffallen. So liefert das hier vorgestellte Verfahren relativ überzeugende Ergebnisse.

Die physikalischen und mathematischen Grundlagen der Brechung wurden bereits in Abschnitt 2.4 behandelt:

$$n_i \cdot \sin \Theta_i = n_r \cdot \sin \Theta_r \tag{3.40}$$

Die Berechnung eines Brechungsvektors nur unter Verwendung von Vektoren

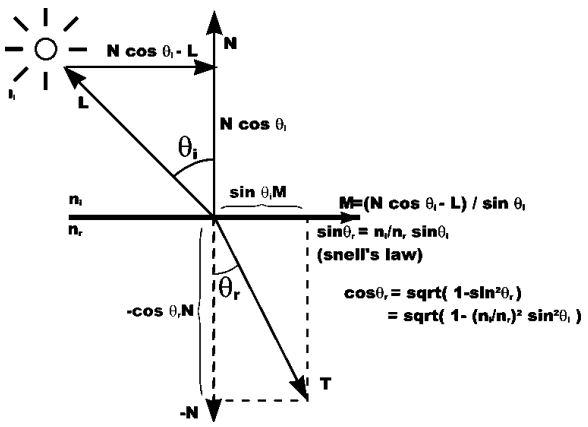

Abb. 3.11: Berechnung des Brechungsvektors. Quelle: [22]

wird nachfolgend gegeben (siehe auch Abbildung 3.11):

$$\hat{\mathbf{r}} = \sin\theta_r \cdot \hat{\mathbf{M}} - \cos\theta_r \cdot \hat{\mathbf{N}} \tag{3.41}$$

$$= \sin\theta_r \cdot \frac{\hat{\mathbf{N}}\cos\theta_i - \hat{\mathbf{L}}}{\sin\theta_i} - \cos\theta_r \cdot \hat{\mathbf{N}} \tag{3.42}$$

$$= \frac{n_i}{n_r} \cdot \left(\hat{\mathbf{N}}\cos\theta_i - \hat{\mathbf{L}}\right) - \cos\theta_r \cdot \hat{\mathbf{N}} \tag{3.43}$$

$$= \left(\frac{n_i}{n_r} \cdot \cos\theta_i - \cos\theta_r\right)\hat{\mathbf{N}} - \frac{n_i}{n_r} \cdot \hat{\mathbf{L}} \tag{3.44}$$

$$= \left(\frac{n_i}{n_r} \cdot \cos\theta_i - \sqrt{1 - \frac{n_i^2}{n_r^2}\sin^2\theta_i}\,\right) \cdot \hat{\mathbf{N}} - \frac{n_i}{n_r} \cdot \hat{\mathbf{L}} \tag{3.45}$$

$$= \left(\frac{n_i}{n_r} \cdot \langle\hat{\mathbf{N}}, \hat{\mathbf{L}}\rangle - \sqrt{1 - \frac{n_i^2}{n_r^2}(1 - \langle\hat{\mathbf{N}}, \hat{\mathbf{L}}\rangle^2)}\,\right) \cdot \hat{\mathbf{N}} - \frac{n_i}{n_r} \cdot \hat{\mathbf{L}} \tag{3.46}$$

Wird nun statt des Lichtvektors der Kameravektor verwandt, ergibt sich der blickpunktabhängige spiegelnde Brechungsvektor. Dieser wird als Texturkoordinate für die Cubemap genutzt.

3.4 Schlagschatten

In den anschliessenden beiden Abschnitten werden zwei Methoden zur Berechnung von Schlagschatten vorgestellt. Zum einen werden projektive Schlagschatten behandelt, die lediglich korrekte Schlagschatten auf Ebenen berechnen und keine Selbstschattierung unterstützen. Zum anderen werden Shadowmaps besprochen. Bei diesem Verfahren werden bei einem

zusätzlichen Renderingpass korrekte Schatten für eine komplette Szene berechnet.

Ein weiteres, sehr populäres Verfahren zur Darstellung von Schlagschatten sei an dieser Stelle der Vollständigkeit halber kurz erwähnt: Shadow Volumes. Dieses Verfahren wurde im Rahmen dieser Arbeit nicht angewendet. Es basiert auf der Generierung von Schattenräumen, und unter Verwendung des Stencil Buffers wird entschieden, ob ein Fragment im Schatten liegt. Dabei ist eine Selbstschattierung möglich. Eine gute Einführung in das Verfahren ist in [7] zu finden.

3.4.1 Projektive Schlagschatten

Bei der Berechnung von projektiven Schlagschatten wird im Grundalgorithmus von folgenden Vereinfachungen ausgegangen:

- die Szene enthält Objekte und ein Ebene,
- lediglich Schlagschatten, die auf dieser Ebene liegen, werden berechnet.

Damit werden Schlagschatten von Objekten auf Objekte und Selbstschattierung vernachlässigt. Die Objekte werden im ersten Pass auf die Ebene projiziert und dort mit der entsprechenden Schattenfarbe oder aber dem ambienten Beleuchtungsanteil der Ebene gezeichnet. Im zweiten Pass werden dann die Objekte selbst gerendert.

Der Projektionsursprung liegt an der Position der jeweiligen Punktlichtquelle und jedes zu zeichnende Polygon wird einzeln auf die Ebene projiziert. Kernstück des Algorithmus ist die Schattenprojektionsmatrix, die im Folgenden entwickelt wird.

Gegeben sei die Ebene, auf die die Schatten projiziert werden sollen und die Gerade, die durch die Lichtquelle und den jeweils zu zeichnenden Vertex verläuft (siehe auch Abbildung 3.12):

$$\text{Ebene} : \langle \mathbf{x}, \hat{\mathbf{n}} \rangle + d = 0 \tag{3.47}$$

$$\text{Gerade} : \mathbf{L} + t \cdot (\mathbf{v} - \mathbf{L}) \tag{3.48}$$

Einsetzen der Geraden- in die Ebenengleichung zur Schnittpunktbestimmung liefert:

$$\langle \mathbf{L} + t \cdot (\mathbf{v} - \mathbf{L}), \hat{\mathbf{n}} \rangle + d = 0 \tag{3.49}$$

$$\tag{3.50}$$

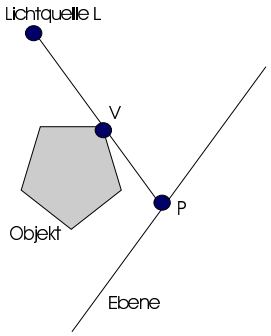

Abb. 3.12: Projektive Schlagschatten: Berechnung der Projektionsmatrix

Es ergibt sich:

$$t = -\frac{d + \langle \hat{\mathbf{n}}, \hat{\mathbf{L}} \rangle}{\langle \hat{\mathbf{n}}, \mathbf{v} - \mathbf{L} \rangle} \tag{3.51}$$

Somit ergibt sich der Schnittpunkt p zu

$$\mathbf{p} = \mathbf{L} - \frac{d + \langle \hat{\mathbf{n}}, \hat{\mathbf{L}} \rangle}{\langle \hat{\mathbf{n}}, \mathbf{v} - \mathbf{L} \rangle} (\mathbf{v} - \mathbf{L}) \tag{3.52}$$

Die gesuchte Schattenprojektionsmatrix muss folgende Eigenschaft erfüllen:

$$\mathbf{p} = \mathbf{L} - \frac{d + \langle \hat{\mathbf{n}}, \hat{\mathbf{L}} \rangle}{\langle \hat{\mathbf{n}}, \mathbf{v} - \mathbf{L} \rangle} (\mathbf{v} - \mathbf{L}) = \mathcal{M}\ \mathbf{v} \tag{3.53}$$

Einige Umformungen erleichtern das Überführen in Matrizenform (der Übersichtlichkeit halber wird in den folgenden Gleichungen das Symbol "$\cdot$" für das Skalarprodukt verwendet):

$$\mathbf{p} = \mathbf{L} - \frac{d(\mathbf{v} - \mathbf{L}) + (\hat{\mathbf{n}} \cdot \mathbf{L})(\mathbf{v} - \mathbf{L})}{\hat{\mathbf{n}} \cdot (\mathbf{v} - \mathbf{L})} \tag{3.54}$$

$$= \frac{\mathbf{L}(\hat{\mathbf{n}} \cdot (\mathbf{v} - \mathbf{L})) - d\mathbf{v} + d\mathbf{L} - (\hat{\mathbf{n}} \cdot \mathbf{L})\mathbf{v} + (\hat{\mathbf{n}} \cdot \mathbf{L})\mathbf{L}}{\hat{\mathbf{n}} \cdot (\mathbf{v} - \mathbf{L})} \tag{3.55}$$

$$= \frac{\mathbf{L}(\hat{\mathbf{n}} \cdot \mathbf{v}) - d\mathbf{v} + d\mathbf{L} - (\hat{\mathbf{n}} \cdot \mathbf{L})\mathbf{v}}{\hat{\mathbf{n}} \cdot \mathbf{v} - \hat{\mathbf{n}} \cdot \mathbf{L}} \tag{3.56}$$

Sei $\mathbf{v}$ ein vierkomponentiger Vektor (xyzw), so ergibt sich:

$$\mathbf{p} = \mathbf{L}(\hat{\mathbf{n}} \cdot \mathbf{v}) + \begin{pmatrix} -d - \hat{\mathbf{n}} \cdot \mathbf{L} & 0 & 0 & d \cdot L_x \\ 0 & -d - \hat{\mathbf{n}} \cdot \mathbf{L} & 0 & d \cdot L_y \\ 0 & 0 & -d - \hat{\mathbf{n}} \cdot \mathbf{L} & d \cdot L_z \\ n_x & n_y & n_z & -\hat{\mathbf{n}} \cdot \mathbf{L} \end{pmatrix} \cdot \mathbf{v} \quad (3.57)$$

$$= \begin{pmatrix} d + \hat{\mathbf{n}} \cdot \mathbf{L} - L_x n_x & -L_x n_y & -L_x n_z & -d \cdot L_x \\ -L_y n_x & d + \hat{\mathbf{n}} \cdot \mathbf{L} - L_y n_y & -L_y n_z & -d \cdot L_y \\ -L_z n_x & -L_z n_y & d + \hat{\mathbf{n}} \cdot \mathbf{L} - L_z n_z & -d \cdot L_z \\ -n_x & -n_y & -n_z & \hat{\mathbf{n}} \cdot \mathbf{L} \end{pmatrix} \cdot \mathbf{v} \quad (3.58)$$

$$= \mathcal{M}\ \mathbf{v} \quad (3.59)$$

Die derart gegebene Matrix $\mathcal{M}$ projiziert einen gegebenen Vertex $\mathbf{v}$ auf die mit $\langle \mathbf{x}, \hat{\mathbf{n}} \rangle + d = 0$ gegebene Ebene. Der Projektionsursprung ist dabei mit der Lichtquellenposition gegeben.

Diese Projektionsmatrix wird auf jeden Vertex angewendet, so dass ein korrekt projiziertes zweidimensionales Abbild (und damit auch Schlagschattenabbild) auf der Ebene entsteht. Die projizierten Objekte werden mit deaktivierter Beleuchtung und der Schattenfarbe bzw. der ambienten Farbe der Ebene gezeichnet

Der Aufwand verdoppelt sich bei dieser Art der Schattendarstellung, da jede Szene zweimal pro Darstellung gerendert werden muss.

Während des Renderingprozesses sind zwei Besonderheiten zu beachten. Zum einen muss sichergestellt werden, dass keine störenden Artefakte bei der Darstellung der Schlagschatten entstehen. Die projizierten Objekte liegen direkt auf der Ebene, besitzen also den gleichen z-Wert wie diese. Damit kann das projizierte Objekt je nach berechneter Position und Fliesskommaungenauigkeit teilweise vor und teilweise hinter der Ebene liegen. Aus diesem Grund sollte entweder der z-Wert jeweils um einen kleinen Wert Δz in Richtung der Kameraposition verschoben, der Tiefenpuffer während des Projektionsrenderingprozesses deaktiviert oder ein Offsetwert[12] für die z-Koordinate gesetzt werden.

Zum anderen kann es passieren, dass die Schlagschatten über die begrenzte Ebene hinausragen, da die Projektion für eine unendliche Ebene definiert

[12] dies wird z.B. von OpenGL und DirectX unterstützt

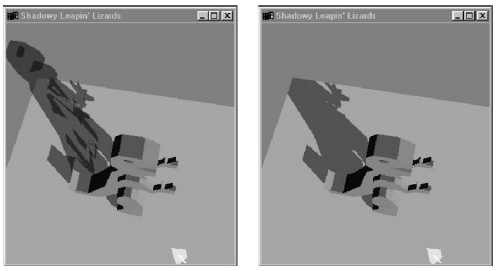

Abb. 3.13: Projektive Schlagschatten: Die Schatten könnten über die Ebene hinausgehen. Dies wird duch die Verwendung des Stencilbuffers vermieden.

ist (siehe Abbildung 3.13). Die Verwendung des Stencil Buffers sorgt dafür, dass die Schlagschatten nicht über die durch Polygone definierte Ebene hinausgehen. Dafür wird vor dem Schattenrenderingpass die Ebene mit aktiviertem Stencil Buffer gezeichnet. Anschliessend wird in dem Projektionspass auf gesetztes Stencil Bit überprüft. So wird sichergestellt, dass die Schatten nur in dem Bereich gezeichnet werden, in dem auch die durch Polygone gegebene Ebene definiert ist.

Eine weitere Problematik kann entstehen, wenn die Punktlichtquelle zwischen Objekt und zu schattierender Ebene liegt. Da die Projektion auch in diesem Fall definiert ist, entstehen sogenannte Antischatten (vgl. Abbildung 3.14). Diese können durch eine Abfrage, die entscheidet ob das

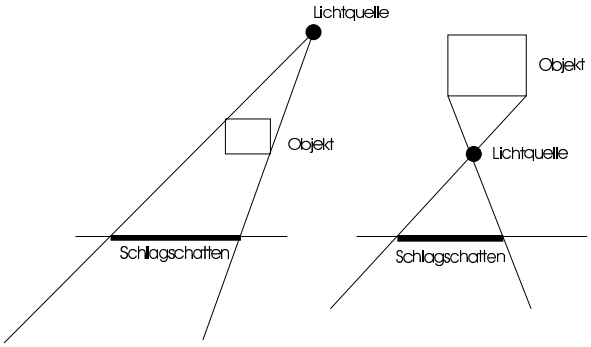

Abb. 3.14: Projektive Schlagschatten: Falls die Lichtquelle zwischen Objekt und zu schattierender Ebene liegt, kommt es zu Antischatten. Hier für den zweidimensionalen Fall dargestellt.

jeweilige Objekt Schlagschatten wirft, beseitigt werden.

Der beschriebene Algorithmus lässt sich ohne grossem Aufwand auf beliebig viele Ebenen/Lichtquellen erweitern. Allerdings ist pro

Ebene/Lichtquelle ein komplettes Rendern der zu schattierenden Szeneteile notwendig, so dass das Verfahren bei zu vielen zu schattierenden Ebenenen/Lichtquellen zu rechenintensiv wird.

3.4.2 Shadowmapping

Shadowmapping ist eine Zweipasstechnik, die Schlagschattenwurf auf andere Objekte und Selbstschattierung unterstützt. Die im Rahmen dieser Arbeit implementierte Form des Shadowmapping lässt lediglich gerichtete Lichtquellen zu. Am Ende dieses Abschnittes wird jedoch ein Verfahren zur Erweiterung auf Punktlichtquellen vorgestellt.

Shadowmapping basiert wie der Name schon sagt auf der Verwendung einer Shadowmap. Im Folgenden wird das Prinzip anhand der Szene in Abbildung 3.15 erläutert.

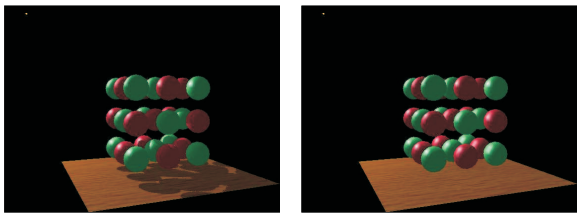

Abb. 3.15: Shadowmapping: Beispielszene. Quelle: [14]

Im ersten Pass wird die Shadowmap, bestehend lediglich aus Tiefenwerten, aus der Lichtquellenperspektive erzeugt und in einer Textur abgelegt (siehe Abbildung 3.16). Die Auflösung der Textur ist für die spätere Qualität der Schlagschatten entscheidend. Sehr gute Ergebnisse werden erzielt, wenn die Auflösung der Textur etwa der Auflösung der dargestellten Szene entspricht. Diese Shadowmap wird nun beim Rendern der Szene von der Lichtquelle aus auf die Szene projiziert (Projektives Texturemapping, siehe Abbildung 3.17). Dazu werden die Texturkoordinaten mit der gleichen Matrix wie beim Rendern aus der Lichtquellenperspektive erzeugt. Lediglich eine Skalierung der Texturkoordinaten auf den entsprechenden Bereich ist zusätzlich erforderlich. Während des Renderingprozesses werden so die entsprechenden Texturkoordinaten (s,t,r,q) generiert. Für den zweidimensionalen Texturzugriff sind lediglich die beiden Koordinaten s/q und t/q erforderlich. Der Wert der Texturkoordinate r/q jedoch enthält aufgrund der Projektion den Abstand der Lichtquelle zum aktuellen Vertex. Diese Eigenschaft wird für einen Tiefenvergleich genutzt:

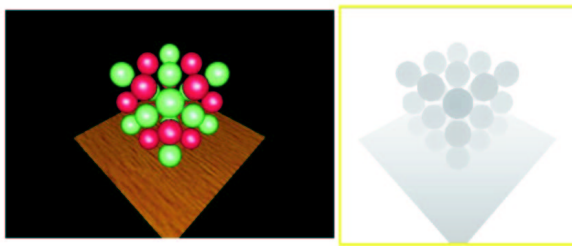

Abb. 3.16: Erzeugen der Shadowmap. Links: Die Szene aus Sicht der Lichtquelle. Rechts: Die zugehörige Shadowmap. Quelle: [14]

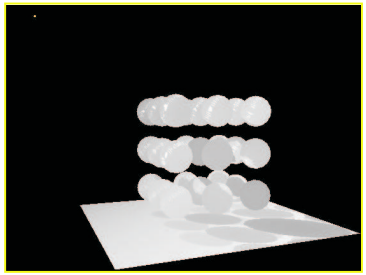

Abb. 3.17: Shadowmapping: Projektion der Shadowmap. Quelle: [14]

Der aktuelle Farbwert der Textur bzw. Tiefenmap beschreibt den Abstand des der Lichtquelle naheliegensten Punktes auf einer Objektoberfläche auf der Geraden liegend, die durch das aktuelle Fragment und die Lichtquelle gegeben ist. Denn auf diese Art und Weise wurde die Tiefenmap generiert. Die interpolierte Texturkoordinate r/q wiederum beinhaltet den Abstand des aktuellen Fragments zur Lichtquelle. Somit kann über einen Vergleich dieser beiden Werte entschieden werden, ob zwischen Lichtquelle und aktuellem Fragment noch ein Objekt liegt. In diesem Fall würde das aktuelle Pixel mit einem Schatten dargestellt werden. Diese Entscheidung sei in Pseudocode gegeben:

- if (texture[s/q, t/q] $\approx$ r/q) then not shadowed
- if (texture[s/q, t/q] $<$ r/q) then shadowed

Dieser Vergleich wird von aktuellen Graphikkarten unterstützt und kann somit sehr effektiv ausgeführt werden. Da das projektive Texturemapping in der Texturprojektionsmatrix ausgeführt werden kann, besitzt es den gleichen Aufwand wie gewöhnliches Texturemapping. Shadowmapping kann also als solches sehr effektiv implementiert werden. Im Falle von

statischen Szenen braucht die Shadowmap nur einmal generiert zu werden und kann dann weiter verwendet werden. In diesem Fall würde es keine Performanceeinbußen geben. Ansonsten halbiert sich die Framerate bei Schattendarstellung etwa, da ein zusätzlicher Pass für die Generierung der Shadowmap erforderlich ist.

Das beschriebene Verfahren kann wegen der Verwendung einer zweidimensionalen Textur lediglich gerichtete Lichtquellen darstellen. Diese Einschränkung würde bei der Verwendung von Cubemaps entfallen, da alle Richtungen durch Shadowmaps abgedeckt werden würden. Bei dynamischen Szenen würde jedoch ein erheblicher Aufwand zur Generierung der sechs Shadowmaps anfallen. In der Praxis allerdings müssen in der Regel nicht für jeden Frame alle sechs Shadowmaps aktualisiert werden, so dass auch hier effektive Lösungen denkbar sind.

Kapitel 4

Implementierung

In den folgenden Abschnitten wird die Umsetzung der in Kapitel 3 vorgestellten Konzepte diskutiert. Im Rahmen dieser Arbeit wurden die Folgenden Konzepte in wiederverwendbaren Klassen umgesetzt:

- Bumpmapping
- Phong Shading
- Banks Shading
- Cook-Torance Shading
- Ashikhmin Shading
- Reflection Mapping
- Refraction Mapping
- Projective Shadowing
- Shadow Mapping.

Zusätzlich wurden mit Hilfe dieser Klassen folgende erweiterte Möglichkeiten in Beispielprogammen realisiert:

- Multiple Lights with Phong Shading
- Soft Shadows
- Multiple Shading Algorithms in One Scene
- Cook-Torrance Shading with different Refraction Indices for different Colors.

Ziel der Implementierung ist es, die Umsetzbarkeit der beschriebenen Verfahren mit aktueller Graphikkartenhardware und ihre Möglichkeiten in der Praxis sowie in Bezug auf ihre Performance, bewerten zu können.

In den anschliessenden Abschnitten wird zunächst eine kurze Einführung in die verwendete Implementierungsumgebung und die verwendeten Hardwareshader gegeben. Darauf folgend werden die einzelnen Implementierungen diskutiert und anschliessend bewertet.

4.1 Auswahl der Implementierungsumgebung

Die Implementierung der beschriebenen Verfahren erfolgte auf einem Pentium IV/1700MHz mit einer auf dem ATI Radeon 9800 pro Chip basierenden Graphikkarte unter Windows 2000. Der ATI Radeon 9800 pro Chip wurde gewählt, um die Möglichkeiten der Verfahren unter Verwendung einer aktuellen Graphikkarte zu testen. Ebensogut hätte eine auf dem nVidia GeForce 5900 Chip basierende Graphikkarte zum Einsatz kommen können, die sich in Bezug auf ihre Performance kaum von der ATI-Version unterscheidet. Nicht zuletzt war die Verfügbarkeit der ATI Graphikkarte ein Entscheidungsgrund.

Mit der Wahl der Graphikkarte wird allerdings auch das zu benutzende SDK vorgegeben, da zum aktuellen Zeitpunkt ein einheitlicher Standard in Bezug auf Vertex- und Fragmentshader[1] unter OpenGL fehlt. Dieser Mangel wird mit OpenGL 2.0 behoben sein, bis aber dieser Standard von den Graphikkartenherstellern umgesetzt sein wird, ist man auf proprietäre Lösungen angewiesen. Dies ist ein grosser Nachteil in Bezug auf die Wiederverwendbarkeit des Codes - da sich jedoch die jew. Umsetzungen der Vertex- und Fragmentshader gerade von nVidia und ATI stark ähneln, ist eine nicht zu aufwendige Portierung möglich.

Das Betriebssystem Windows 2000 wurde aus einem ähnlichen Grund gewählt, denn die SDKs werden in der Regel nur für Windows veröffentlicht.

Als Graphikbibliothek kam OpenGL 1.4 unter VS C++ 6.0 unter Verwendung von Glut 1.7 zum Einsatz. OpenGL ist die am weitesten verbreitete Lösung und wurde somit auch im Rahmen dieser Arbeit verwandt. Zudem ist OpenGL auf den verschiedensten Plattformen verfügbar und bietet damit z.B. einen entscheidenden Vorteil gegenüber DirectX.

[1] DirectX 9.0 besitzt als erste Graphikbibliothek einen Standard.

4.2 `EXT_vertex_shader` und `ATI_fragment_shader`

Im Rahmen dieser Arbeit wurden die OpenGL Erweiterungen `EXT_vertex_shader` und `ATI_fragment_shader` zur Implementierung des Vertex- und des Fragmentshaders genutzt. Wie die Namensgebung `EXT_vertex_shader` schon andeutet, handelt es sich um eine OpenGL Extension, die von Silicon Graphics genehmigt ist und somit von verschiedenen Graphikkartenherstellern unterstützt werden sollte. Bei `ATI_fragment_shader` handelt es sich um eine proprietäre Erweiterung von ATI, die bislang von keinem anderen Hersteller unterstützt wird. Die Portierung der Vertexprogramme sollte also in der Regel keine Probleme bereiten.

Die Dokumentationen zu diesen beiden Erweiterungen bestehen lediglich aus den jew. Spezifikationen ([9], [10]), einem Paper ([11]) und einigen kleinen Beispielprogrammen[2] und Präsentationen. Aus diesem Grunde wird an dieser Stelle eine kurze, aber nicht vollständige Einführung gegeben. Weitere Informationen sind in den oben genannten Veröffentlichungen zu finden.

Vertexshader stellen eine programmierbare Transformations- und Beleuchtungseinheit dar. Das jeweilige Programm erlaubt Berechnungen für jeden Vertex und alle daran gebundenen Informationen (z.B. Normale, MVP-Matrix, etc.). Als Rückgabewert werden Farbwerte und Texturkoordinaten definiert, die interpoliert an den Fragmentshader übergeben werden. Bei einem aktivierten programmierbaren Vertexshader werden die entsprechenden Operationen wie z.B. Normalentransformationen, per-Vertex Lighting und Modelview- und Projektionstransformationen von OpenGl selbst nicht mehr ausgeführt, so dass diese vom programmierbaren Vertexshader durchgeführt werden müssen.

Mit Fragmentshadern können Programme definiert werden, die für jedes Fragment einzeln ausgeführt werden. Als Parameter erhält der Fragmentshader die vom Vertexshader übergebenen und interpolierten Farbwerte. In der Regel ist der Fragmentshader in Bezug auf Anzahl der verfügbaren Befehle aus Performancegründen sehr beschränkt. Als Rückgabewert liefert der Fragmentshader den für das aktuelle Fragment benutzten Farbwert.

[2] die Anzahl der verfügbaren Beispielprogramme auf `http://www.ati.com` erhöhte sich z.B. während der Anfertigung dieser Arbeit von 3 auf 6. Dabei handelt es sich jeweils um sehr einfache Beispiele.

Vertexshader `EXT_vertex_shader` Dieser Vertexshader bietet einen der OpenGL-Befehlssatznamensgebung entsprechenden Befehlssatz. So können Vertexshaderoperationen mit den folgenden Befehlen definiert werden:

- `void ShaderOp1EXT(enum op, uint res, uint arg1)`
- `void ShaderOp2EXT(enum op, uint res, uint arg1, uint arg2)`
- `void ShaderOp3EXT(enum op, uint res, uint arg1, uint arg2, uint arg3)`

Das Ergebnis der Operation `op` wird in `res` abgelegt. Die Argumente werden in `arg1`, `arg2` und `arg3` übergeben. Mögliche Operatoren sind in Tabelle 4.1 aufgeführt. Die Namensgebung der Operatoren ist prinzipiell selbsterklärend. Für nähere Informationen siehe [10].

OP_INDEX_EXT	OP_NEGATE_EXT	OP_DOT3_EXT
OP_DOT4_EXT	OP_MUL_EXT	OP_ADD_EXT
OP_MADD_EXT	OP_FRAC_EXT	OP_MAX_EXT
OP_MIN_EXT	OP_SET_GE_EXT	OP_SET_LT_EXT
OP_CLAMP_EXT	OP_FLOOR_EXT	OP_ROUND_EXT
OP_EXP_BASE_2_EXT	OP_LOG_BASE_2_EXT	OP_POWER_EXT
OP_RECIP_EXT	OP_RECIP_SQRT_EXT	OP_SUB_EXT
OP_CROSS_PRODUCT_EXT	OP_MULTIPLY_MATRIX_EXT	OP_MOV_EXT

Tabelle 4.1: `EXT_vertex_shader` opcodes

Weitere Befehle umfassen das Erzeugen von Variablen und Konstanten und das Definieren von Swizzle- und WriteMask-Argumenten. Als Ausgaberegister stehen die in Tabelle 4.2 dargestellten zur Verfügung.

Fragmentshader `ATI_fragment_shader` Dieser Fragmentshader bietet ebenfalls einen der OpenGL-Befehlssatznamensgebung entsprechenden Befehlssatz. Ein Fragmentshaderprogramm kann aus bis zu zwei Pässen bestehen, wobei jeder Pass maximal acht Befehle umfassen darf. Der Beginn eines jeden Passes wird durch den Zugriff auf Texturen bzw. Texturkoordinaten definiert:

OUTPUT_VERTEX_EXT	OUTPUT_FOG_EXT
OUTPUT_COLOR0_EXT	OUTPUT_COLOR1_EXT
OUTPUT_TEXTURE_COORD0_EXT	OUTPUT_TEXTURE_COORD1_EXT
OUTPUT_TEXTURE_COORD2_EXT	OUTPUT_TEXTURE_COORD3_EXT
OUTPUT_TEXTURE_COORD4_EXT	OUTPUT_TEXTURE_COORD5_EXT

Tabelle 4.2: `EXT_vertex_shader` Ausgaberegister

- `void PassTexCoordATI (uint dst, uint coord, enum swizzle);`
- `void SampleMapATI (uint dst, uint interp, enum swizzle);`

Dabei bezeichnet `dst` eines der sechs zu Verfügung stehenden Register `REG_X_ATI` ($x \in [0, 5]$), `interp` und `coord` eine der 8 zur Verfügung stehenden Texturen `TEXTUR_Y_ARB` ($y \in [0, 7]$) und `swizzle` einen der zur Verfügung stehenden Swizzle-Modes (siehe [9]). Der Unterschied dieser beiden Operationen sei hier kurz erkläutert, da dieser in der Spezifikation nicht sehr deutlich herausgestellt ist:

`PassTexCoord (REG_X_ATI, TEXTURE_Y_ARB, GL_NONE)` übergibt die aktuellen Texturkoordinaten von Textur Y an das übergebene Register X. `SampleMapATI (REG_X_ATI, TEXTURE_Y_ARB, GL_NONE)` übergibt den Farbwert von Textur X(!) an den aktuellen Koordinaten von Textur Y an das Register X. Diese Art der Darstellung kann leicht zu Verwechselungen führen.

Fragmentoperationen können mit den folgenden Befehlen definiert werden:

- `void ColorFragmentOp1ATI ()`
- `void ColorFragmentOp2ATI ()`
- `void ColorFragmentOp3ATI ()`
- `void AlphaFragmentOp1ATI ()`
- `void AlphaFragmentOp2ATI ()`
- `void AlphaFragmentOp3ATI ()`

Der jeweilige übergebene Operator bezieht sich je nach Funktion auf die drei RGB-Farbwerte oder aber den Alpha-Wert. Übergeben wird jeweils ein Resultregister, der Operator, die Anzahl von jew. Operanden und für jeden Operanden und das Resultregister ein Swizzle-Mode.

Die in Tabelle 4.3 aufgeführten Befehle stehen in dem Fragmentshader zur Verfügung.

ADD_ATI	SUB_ATI	MUL_ATI
MAD_ATI	LERP_ATI	MOV_ATI
CND_ATI	CND0_ATI	DOT2_ADD_ATI
DOT3_ATI	DOT4_ATI	-

Tabelle 4.3: `ATI_fragment_shader` opcodes

Die Möglichkeiten innerhalb des Fragmentshaders sind relativ beschränkt, da nur elf einfache Funktionen zur Verfügung stehen und ein Fragmentshaderprogramm lediglich aus max. zwölf Befehlen bestehen darf. Dennoch sind z.B. mit den in Kapitel 3 beschriebenen Verfahren zahlreiche Anwendungsmöglichkeiten gegeben.

Schliesslich sei am Ende noch eine wichtige Eigenschaft des Fragementshaders `ATI_fragment_shader` erwähnt: Die interne Fliesskommaeinheit ist auf den Bereich $[-4, 4]$ beschränkt [9]. Bei der Mehrheit der Anwendungen ist diese Beschränkung keine grosse Einschränkungen, da in der Regel hauptsächlich mit normierten Vektoren in Skalarprodukten gerechnet wird, die somit immer im Bereich $[-1, 1]$ liegen. Der gegebene Beschränkungsbereich von $[-4, 4]$ lässt somit noch einige Additionen und auch Multiplikationen mit kleinen Zahlenwerten zu. Dennoch kann es aufgrund dieser Beschränkung zu Problemen kommen - dies ist z.B. bei der Umsetzung des Ashikhmin Modells der Fall (vgl. dazu 4.4.4).

4.3 Grundlagen der Implementierung

Im Rahmen dieser Arbeit wurden sieben Real Time Shader Klassen entwickelt:

- `rts_phong`
- `rts_banks`
- `rts_cooktorrance`
- `rts_ashikmin`

- `rts_reflectionrefraction`
- `rts_shadow`
- `rts_shadowmap`

Diese Klassen können ohne grossen Aufwand in bestehende OpenGL Programme eingebunden werden und erweitern damit die Möglichkeiten der Graphikdarstellung um den jeweiligen Shader bzw. die jeweilige Schlagschattendarstellung.

Alle diese Klassen werden ähnlich dem OpenGL Paradigma `glBegin()` und `glEnd()` benutzt. Zunächst muss die jeweilige Klasse instanziiert und initialisiert werden. Aktiviert wird der jeweilige Shader vor dem Rendern mit `bind()` und freigegeben nach dem Rendern mit `release()`.

In den anschliessenden Abschnitten werden die jeweiligen Klassen einzeln besprochen. Die Klasse `rts_phong` wird mit Source-Code Beispielen am ausführlichsten behandelt, da die anderen Klassen im Prinzip die gleichen Verfahren benutzen. Bei diesen Klassen wird auf den Sourcecode der auf der beiliegenden CD zu finden ist verwiesen. Diese Arbeit kann keine vollständige Beschreibung der entstandenen Programme geben, ganz ohne Programm-Auszüge hingegen kann sie auch nicht auskommen. Zudem ist eine Beschreibung der prinzipiellen Abläufe sinnvoll, da die Dokumentation der Vertex- und Fragmentshader - wie bereits erwähnt - nicht sehr ausführlich ist.

An dieser Stelle sei noch erwähnt, dass die entwickelten Klassen Beispiele für die Möglichkeiten von Vertex- und Fragmentshadern darstellen. In ihrem jetzigen Funktionsumfang sind sie ohne Änderung weiterverwendbar. Denoch empfiehlt es sich, die Klassen für den jeweiligen konkreten Anwendungsfall anzupassen, um aus Performancegründen Erweiterungen wie z.B. Texturemapping oder per-Fragment spekulare Maps zu entfernen oder je nach Bedarf die entsprechenden Erweiterungen hinzuzufügen. Da sie damit einen prinzipiell beispielhaften Charakter besitzen, wurde auch auf eine Ableitung von evtl. Oberklassen verzichtet, um die grundsätzlichen Prinzipien klar darzustellen.

Zum Einlesen der Texturen werden die Funktionen aus `tga.c` genutzt, entwickelt und bereitgestellt von ATI. Die 3D-Studio Objekte werden mit den Funktionen der Library `3DS.lib` eingelesen - ebenfalls entwickelt von ATI. 3D-Studio Files können jedoch keine Tangenten abspeichern. Aus diesem Grund werden die Tangenten generiert. Durch diese homogene Erzeugung verlieren leider auch die auf Tangenten aufbauenden

Shadingverfahren Banks und Cook-Torrance ihre Mächtigkeit. Aus diesem Grund wurde für die Beispielprogramme noch jeweils ein Quadrat mit stark variierenden Tangenten erzeugt, so dass zumindest bei diesem Objekt die Vorteile der beiden genannten Verfahren ersichtlich sind. Zusätzlich besitzen einige der 3D-Studio-Objekte keine abgespeicherten Texturkoordinaten, so dass diese immer ohne Texturierung dargestellt werden.

Ein weiterer Nachteil der 3D-Studio Files ist, dass die Normalen in Objektraumkoordinaten abgespeichert sind, diese aber für Bump-Mapping z.B. in Tangentenraumkoordinaten vorliegen sollten. Eine prinzipielle Lösung könnte sein, die Normalen während des Einlesens in den Tangentenraum zu transformieren. In dieser Arbeit wurden die jeweiligen Skalarprodukte jedoch im Objektraum berechnet - lediglich beim Bumpmapping wurden die Skalarprodukte im Tangentenraum durchgeführt. Um die Transformation in den Tangentenraum zu vermeiden, wurde für das Bumpmapping Beispiel ein nahezu ebenes 3D-Studio Objekt gewählt, bei dem als solches die Objektraumnormalen den Tangentenraumnormalen nahezu entsprechen.

Die im einfachen `.ppm` Format vorliegenden Cubemap-Texturen werden mit Hilfe einer kurzen im Rahmen dieser Implementierung entwickelten Funktion eingelesen, um die notwendigen Umformungen besser vornehmen zu können.

4.4 Implementierung: Beleuchtungsmodelle

In diesem Abschnitt werden die entwickelten Klassen `rts_phong`, `rts_banks`, `rts_cooktorrance`, `rts_ashikmin` und `rts_reflectionrefraction` vorgestellt und die während der Implementierung aufgetretenen Probleme diskutiert.

4.4.1 Phong

Die Klasse `rts_phong` besitzt den folgenden Konstruktor, Destruktor und die folgenden Methoden (siehe `rts_phong.h`):

```
class rts_phong
{
    public:

    rts_phong(float diffuse=0.5,            // diffuse parameter
              float specular=0.5,           // specular parameter
              int specular_exponent=128,    // exponent
              GLfloat lightColorR=1.0,
              GLfloat lightColorG=1.0,
              GLfloat lightColorB=1.0,      // light color
              char textureFile[80]="\n",    // material texture, else white (1,1,1)
              char specularFile[80]="\n",   // specular parameter map, else specular
```

```
            char bumpFile[80]="\n",        // bump map
            int nhhhMapSize_param=1024);   // size of the (n.h, h.h) map (XxX),
                                           // higher->better quality

    ~rts_phong() {Destroy();};

    bool Init (GLuint tangent, GLuint binormal, GLuint normal);
    bool Destroy (void);

    void Bind (float lightPos[3]);
    void Release (void);
}
```

Dem Konstruktor können zahlreiche Parameter übergeben werden. Die für die Berechnung des Phong Modells wichtigsten sind die ersten drei Paramter. Zusätzlich kann die Farbe der Lichtquelle definiert werden. Wahlweise können `.tga` Files übergeben werden, die jew. eine Texturemap, Specularmap oder Bumpmap enthalten. Ein übergebener Parameter `'/n'` bedeutet, dass die jew. Map nicht verwendet werden soll. Schliesslich kann die Grösse der nhhh-Map definiert werden.

Die Methode `Init()`, die notwendige Initialisierungen von Parametern, Vertex- und Fragmentshader und z.B. notwendigen Funktionenmaps vornimmt, benötigt die Paramter `tangent`, `binormal` und `normal`, die den jeweiligen Vertexshader mit dem entsprechenden Vektor im OpenGL `Object Buffer` verknüpft. Aus Standardisierungsgründen werden diese drei Parameter bei allen hier vorgestellten Shadern übergeben, auch wenn die Tangente und Binormale, wie z.B. beim Phong-Shader, nicht genutzt werden.

Der Methode `Bind()` schliesslich wird die aktuelle Lichtquellenposition übergeben und bindet die jew. Fragment- und Vertexshader und etwaige notwendige Texturen. Diese werden von `Release()` wieder freigegeben.

Das Einbinden der Klasse `rts_phong` geschieht also mit folgenden Schritten:

1. Instanziierung, z.B.:

```
rts_phong my_phong = new rts_phong
            (gl_phong_diffuse,
             gl_phong_specular,
             gl_phong_specular_exp,
             gl_light_r, gl_light_g, gl_light_b,
             textureFileName,
             specularMapFileName,
             "\n");
```

2. Initialisierung, z.B.:

```
if (!(my_phong->Init(tangentVariantID,
                     binormalVariantID,
                     normalVariantID)))
   {exit(0);}
```

3. Rendern, z.B.:

```
my_phong->Bind(lightpos);
draw_scene();
my_phong->Release();
```

Während der Initialisierung werden wie bereits oben angedeutet die Texturen und Maps geladen, die `nhhh`-Map und der Vertex- und Fragmentshader erzeugt. Die `nhhh`-Map wird entsprechend Abschnitt 3.2.1 generiert (dazu sei nochmals die entsprechende Formel gegeben):

$$f(x,y) = \min\left(\left(\frac{x}{\sqrt{y}}\right)^q, 1\right) \quad x = \langle \mathbf{h}, \hat{\mathbf{n}} \rangle; y = \langle \mathbf{h}, \mathbf{h} \rangle \tag{4.1}$$

```
#define N_DOT_H (((GLfloat)x)/((GLfloat)nhhhSize-1.0f))
#define H_DOT_H (((GLfloat)y)/((GLfloat)nhhhSize-1.0f))

for (int x = 0; x < nhhhSize; ++x)
        // N.H is a scalar in the range [0.0, 1.0]
{
  for (int y = 0; y < nhhhSize; ++y)
        // H.H is a scalar in the range [0.0, 1.0]
  {
    nhhhTex[(y * nhhhSize) + x] = (GLubyte)(255.0f *
                    min((GLfloat) pow(N_DOT_H/(sqrt(H_DOT_H)),
                                      (float)k)
                        , 1.0f ));
  }
}
```

Der Vertexshader wird folgendermassen erzeugt:

```
// creates the vertex shader
pointLightVertexShaderID = glGenVertexShadersEXT(1);
if (pointLightVertexShaderID == 0)
{
  printError("phongShaderInit():
              unable to generate a vertex shader\n");
  return(false);
}

// bind and define vertex shader
glBindVertexShaderEXT(pointLightVertexShaderID);

// bind some gl state to vertex shader parameters
GLuint mvpMatrix = glBindParameterEXT(GL_MVP_MATRIX_EXT);
GLuint vertex    = glBindParameterEXT(GL_CURRENT_VERTEX_EXT);
GLuint lightPos  = glBindTextureUnitParameterEXT
                   (GL_TEXTURE2_ARB, GL_CURRENT_TEXTURE_COORDS);
```

```
GLuint eyePos    = glBindTextureUnitParameterEXT
                   (GL_TEXTURE1_ARB, GL_CURRENT_TEXTURE_COORDS);
GLuint texCoords = glBindTextureUnitParameterEXT
                   (GL_TEXTURE0_ARB, GL_CURRENT_TEXTURE_COORDS);

glBeginVertexShaderEXT();
{
    // Vertexshader Code
}
glEndVertexShaderEXT();
```

Zunächst wird ein Vertexshader mit entsprechender Zugriffsvariable generiert. Darauf folgend wird zunächst der Shader und dann einige OpenGL Zustandsvariablen gebunden. Im Vertexshader selbst werden zunächst einige Skalar- und Vektorvariablen definiert, z.B. mit

```
GLuint tempS = glGenSymbolsEXT(GL_SCALAR_EXT, GL_LOCAL_EXT,
                               GL_FULL_RANGE_EXT, 1);
GLuint tempV = glGenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_EXT,
                               GL_FULL_RANGE_EXT, 1);
```

Anschliessend werden der Normalen-, der Lichtquellen- und der Kameravektor transformiert und gegebenenfalls normiert. Ein Beispiel für die Normierung der Normalen sei gegeben:

```
// transform normal to world space
glShaderOp2EXT(GL_OP_MULTIPLY_MATRIX_EXT, normal_ws, mvpMatrix, normal);
// normalize world normal
glShaderOp2EXT(GL_OP_DOT3_EXT, tempS, normal_ws, normal_ws);
glShaderOp1EXT(GL_OP_RECIP_SQRT_EXT, tempS, tempS);
glShaderOp2EXT(GL_OP_MUL_EXT, normal_norm, normal_ws, tempS);
```

Schliesslich wird der Halbvektor **h** bestimmt und folgende Ausgaben werden getätigt:

```
// output to fragment shader
glShaderOp1EXT(GL_OP_MOV_EXT, GL_OUTPUT_TEXTURE_COORD0_EXT, texCoords);
glShaderOp1EXT(GL_OP_MOV_EXT, GL_OUTPUT_TEXTURE_COORD1_EXT, normal);
glShaderOp1EXT(GL_OP_MOV_EXT, GL_OUTPUT_TEXTURE_COORD3_EXT, halfVec);
glShaderOp1EXT(GL_OP_MOV_EXT, GL_OUTPUT_TEXTURE_COORD4_EXT, lightVec);

// transform vertex by modelview and projection matrix
glShaderOp2EXT(GL_OP_MULTIPLY_MATRIX_EXT, GL_OUTPUT_VERTEX_EXT, mvpMatrix, vertex);
```

Im Fragmentshader werden schliesslich die in Abschnitt 3.2.1 vorgestellten Rechnugen durchgeführt. Zunächst wird der Fragmentshader vergleichbar mit dem Vertexshader generiert.

Anschliessend werden die benötigten interpolierten Vektoren eingelesen:

```
glPassTexCoordATI(GL_REG_1_ATI, GL_TEXTURE1_ARB,
                  GL_SWIZZLE_STR_ATI); // get N from interpolation
glPassTexCoordATI(GL_REG_3_ATI, GL_TEXTURE3_ARB,
                  GL_SWIZZLE_STR_ATI); // half angle vec (note
```

```
                                        // this doesn't have to be
                                        // normalized since we are
                                        // using an nhhh map)
glSampleMapATI(GL_REG_5_ATI, GL_TEXTURE4_ARB,
               GL_SWIZZLE_STR_ATI);     // use cubemap to normalize L
```

Dann werden die benötigten Skalarprodukte berechnet:

```
// reg5 = N.L
glColorFragmentOp2ATI(GL_DOT3_ATI,
                      GL_REG_5_ATI, GL_NONE, GL_NONE,
                      GL_REG_1_ATI, GL_NONE, GL_NONE,
                      GL_REG_5_ATI, GL_NONE,
                            GL_2X_BIT_ATI|GL_BIAS_BIT_ATI);

// reg1 = N.H
glColorFragmentOp2ATI(GL_DOT3_ATI,
                      GL_REG_1_ATI, GL_NONE, GL_NONE,
                      GL_REG_1_ATI, GL_NONE, GL_NONE,
                      GL_REG_3_ATI, GL_NONE, GL_NONE);

// reg1(green) = H.H = |H|^2
glColorFragmentOp2ATI(GL_DOT3_ATI,
                      GL_REG_1_ATI, GL_GREEN_BIT_ATI, GL_NONE,
                      GL_REG_3_ATI, GL_NONE, GL_NONE,
                      GL_REG_3_ATI, GL_NONE, GL_NONE);
```

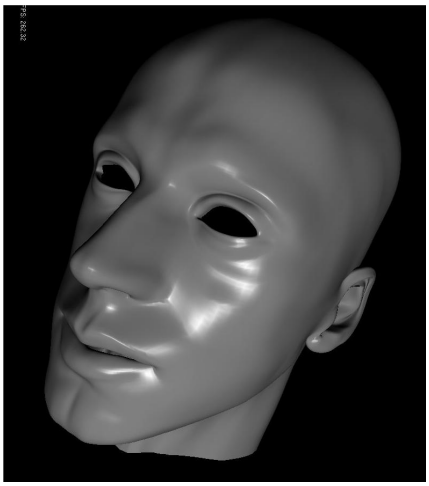

Abb. 4.1: Phong Shading. Parameter: $k_d = 0.5$, $k_s = 0.5$, $q = 128$, nhhh-map Size : 1024. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

Diese werden im zweiten Pass zum `nhhh`-Map Zugriff ($\langle \hat{\mathbf{n}}, \mathbf{h} \rangle$ und $\langle \mathbf{h}, \mathbf{h} \rangle$) genutzt oder aber in den zweiten Pass weitergereicht (Die Register werden mit Beginn des Zweiten Passes gelöscht):

```
glSampleMapATI(GL_REG_4_ATI, GL_REG_1_ATI,
               GL_SWIZZLE_STR_ATI); // raise N.H to q th power
glPassTexCoordATI(GL_REG_5_ATI, GL_REG_5_ATI,
               GL_SWIZZLE_STR_ATI); // pass thru diffuse term
```

Im zweiten Pass werden schliesslich die entsprechenden Terme addiert und multipliziert.

Die Methode `Bind()` bindet vor dem Rendern den Vertex- und den Fragmentshader und die notwendigen Texturen. Freigegeben werden sie nach dem Rendern mit `Release()`.

Ein Beispiel der im Rahmen dieser Arbeit entwickelten Klasse `rts_phong` ist in Abbildung 4.1 dargestellt.

4.4.1.1 Bumpmapping

Die Klasse `rts_phong` ist neben der oben beschriebenen Funktionalität um die Möglichkeit des Bump Mappings erweitert. Dabei gelten die in der Einführung erwähnten Einschränkungen bezüglich der Normalen der eingelesenen 3D-Studio Objekte. Die Bumpmapping Funktionalität ist mit Hilfe von `if then else` Befehlen in den Fragmentshader integriert, so dass die zugehörigen Befehle nur dann ausgeführt werden, wenn dem Konstruktor eine Bumpmap übergeben wird. Dadurch wird gewährleistet, dass zusätzliche Kosten nur dann entstehen, wenn sie tatsächlich benötigt werden.

Ein Beispiel des im Rahmen dieser Arbeit entwickelten Bumpmappings wurde bereits in Abbildung 3.2 dargestellt.

4.4.1.2 Specular Coefficient Mapping

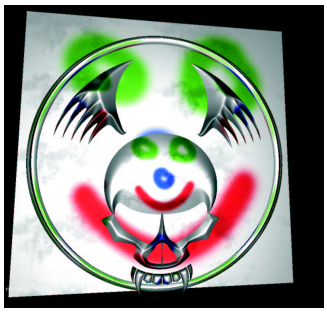

Abb. 4.2: Phong Shading mit Specular Coefficient Mapping (glossy map). Parameter: $k_d = 0.5$, $k_s = 0.5$, $q = 128$, nhhh-map Size : 1024. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

Das Spekulare Koeffizientenmapping ist prinzipiell mit der gleichen Vorgehensweise realisiert, wie das Bumpmapping: Die entsprechenden Befehle

werden lediglich dann ausgeführt, wenn sie tatsächlich benötigt werden. Als Koeffizient wird nicht der konstante, dem Konstruktor übergebene Wert genutzt, sondern der aus der Map ausgelesene. Vergleiche dazu auch Abbildung 4.2.

4.4.2 Banks

Abb. 4.3: Banks Shading. Parameter: $k_d = 2.8$, $k_s = 4.2$, $p = 4$, $q = 20$. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

Die Realisierung des Banks Modells entspricht im Wesentlichen der Realisierung des Phong Modells. Der Unterschied besteht darin, dass in `rts_banks` die in Abschnitt 3.2.2 beschriebenen Funktionenmaps erzeugt werden. Zudem ist für das Banks Modell kein Bumpmapping und Koeffizientenmapping implementiert.

Einschränkungen des Modells im Rahmen der Realisierung sind nicht gegeben. Zwei Beispiele des implementierten Banks Modells seien in Abbildung 4.3 und 4.4 gegeben.

4.4.3 Cook-Torrance

Auch das Cook-Torrance Modell kann nach dem gleichen Muster wie das Phong und das Banks Modell umgesetzt werden. Die notwendigen Fresnel- und Distributionsfunktion können direkt umgesetzt werden. Für die Abschattungs-/Maskierungsfunktion G jedoch ist eine numerische Berech-

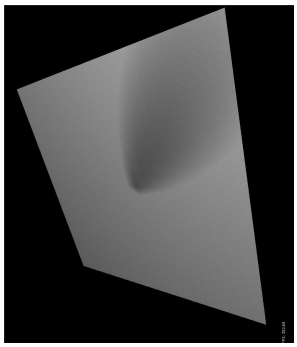

Abb. 4.4: Banks Shading. Die Variation der Tangenten führt zu anisotropen Beleuchtungserscheinungen. Parameter: $k_d = 2.8$, $k_s = 4.2$, $p = 4$, $q = 20$. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

nung von

$$\operatorname{erfc}(x) = 1 - \left(\frac{2}{\pi}\right)^{\frac{1}{2}} \cdot \int_0^x e^{-t^2} \cdot dt \tag{4.2}$$

notwendig. In der Realisierung wurde eine Approximation mit Hilfe einer Taylor-Entwicklung bis zur 24. Ordnung gewählt:

$$\begin{aligned}\operatorname{erfc}(x) \approx 1 &- 2\,\frac{1}{\sqrt{\pi}}\,x + \frac{2}{3}\,\frac{1}{\sqrt{\pi}}\,x^3 - \frac{1}{5}\,\frac{1}{\sqrt{\pi}}\,x^5 + \frac{1}{21}\,\frac{1}{\sqrt{\pi}}\,x^7 - \\ &\frac{1}{108}\,\frac{1}{\sqrt{\pi}}\,x^9 + \frac{1}{660}\,\frac{1}{\sqrt{\pi}}\,x^{11} - \frac{1}{4680}\,\frac{1}{\sqrt{\pi}}\,x^{13} + \\ &\frac{1}{37800}\,\frac{1}{\sqrt{\pi}}\,x^{15} - \frac{1}{342720}\,\frac{1}{\sqrt{\pi}}\,x^{17} + \frac{1}{3447360}\,\frac{1}{\sqrt{\pi}}\,x^{19} \\ &- \frac{1}{38102400}\,\frac{1}{\sqrt{\pi}}\,x^{21} + \frac{1}{459043200}\,\frac{1}{\sqrt{\pi}}\,x^{23} + O(x^{25})\end{aligned} \tag{4.3}$$

Die $\operatorname{erfc}(x)$ Funktion ist streng monoton fallend und besitzt an der Stelle $x = 2$ den Wert $\operatorname{erfc}(2) = 0.004677734981$. Die Funktion $\operatorname{erfc}(x)$ und ihre Differenz zum Taylor-Polynom sind in Abbildung 4.6 dargestellt. Im Bereich oberhalb von 2 wird im Rahmen der Approximation der Wert 0 angenommen. Somit beträgt die Abweichung mit Hilfe dieser Approximation in keinem Fall mehr als 0.005. Diese Abweichung ist für die Echtzeitcomputergraphik akzeptabel. Man bedenke, dass diese Abweichung selbst bei direkter multiplikativer Einflussnahme, die in diesem Falle nicht einmal vorliegt, eine Farbweichung von $255 \cdot 0.005 = 1,28$, also kaum mehr als einem Farbwert, verursacht.

Abb. 4.5: Cook-Torrance Shading. Parameter: $k_d = 0.2$, $k_s = 1.8$, $n1 = 0.6$, $n2 = 1.1$. Die Energieerhaltung wurde an dieser Stelle ausser Aucht gelassen, um eine geeignete Helligkeit zu erziehlen. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

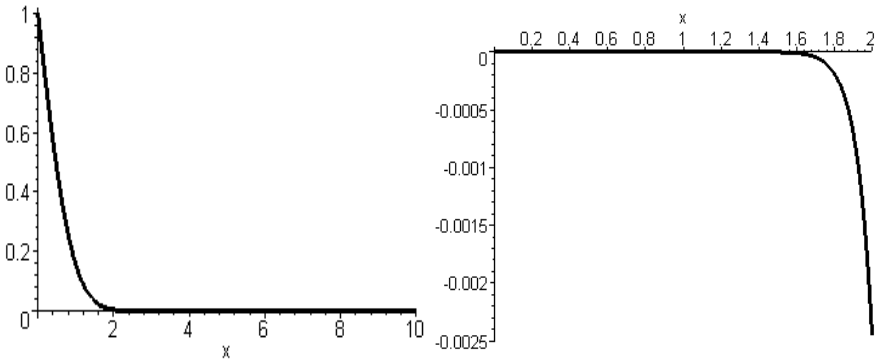

Abb. 4.6: Links: die erfc(**x**)-Funktion ($x \in [0, 10]$) Rechts: Differenz von erfc(**x**) und Taylorpolynomapproximation ($x \in [0, 2]$)

Die Erzeugung der Funktionenmaps f und g (siehe Abschnitt 3.2.3) erfolgt der Erzeugung der nhhh-Map in der Klasse `rts_phong` entsprechend. Mit Hilfe der Klasse `rts_cooktorrance` gerenderte Beispiele sind in Abbildung 4.5 und 4.19 dargestellt.

Abb. 4.7: Ashikhmin Shading. Parameter: $k_d = 0.8$, $k_s = 0.2$, $q_t = 1$, $q_s = 100$. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

4.4.4 Ashikhmin

Die prinzipielle Umsetzung des Ashikhmin Modells (vgl. Abschnitt 3.2.4) geschieht ebenso, wie bei den vorhergehenden drei Modellen beschrieben.

Jedoch werden die durch den benutzten Fragmentshader `ATI_fragment_shader` vorgegebenen Einschränkungen (vgl. dazu Abschnitt 4.2) in Bezug auf Anzahl der möglichen Befehle und Definitionsbereich der Fliesskommazahlen ($[-4, 4]$) bei der Realisierung des Ashikhmin Modells relevant:

1. es nicht möglich mit nur einem Fragmentshader sowohl den diffusen als auch den spekularen Anteil des Ashikhmin Modells mit nur 12 Befehlen (bei zwei Pässen) zu berechnen

2. die Wahl der Parameter des Ashikhmin Modells ist aufgrund des gegebenen Definitionsbereiches der Fliesskommazahlen stark eingeschränkt

Ein Beweis für diese beiden Thesen sei gegeben (vgl. dazu auch Abschnitt 3.2.4 und die zugehörigen Formeln):

1.
   - für die Umsetzung des spekularen Anteils des Ashikhmin Modells werden unter anderem die folgenden Vektoren benötigt: $\hat{\mathbf{h}}$, $\hat{\mathbf{v}}$
   - diese müssen mit Hilfe von Cubemaps normiert werden, ansonsten kommt es zu starken Artefakten in der Bilddarstellung
   - für die Umsetzung des spekularen Anteils des Ashikhmin Modells werden vier Funktionen Maps benötigt

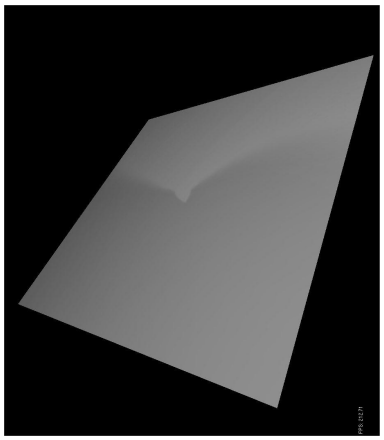

Abb. 4.8: Ashikhmin Shading. Durch Variationen der Tangenten und Binormalen werden anisotrope Beleuchtungserscheinungen erziehlt. Parameter: $k_d = 0.8$, $k_s = 0.2$, $q_t = 1$, $q_s = 100$. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

- damit sind die 6 im Fragmentshader möglichen Texturen ausgenutzt
- für die Funktionen Map des diffusen Anteils ist also keine Textur mehr frei
- q.e.d.[3]

2.
- zur Berechnung des spekularen Anteils ist die Berechnung von $C_a = k_s \frac{\sqrt{(q_t+1)(q_s+1)}}{8\pi}$ notwendig
- C_a muss im Fragmentshader zum spekularen Anteil multipliziert werden
- somit ergibt sich die folgende Beschränkung für das umgesetzte Ashikhmin Modell:

$$C_a \leq 4 \tag{4.4}$$

$$\Leftrightarrow \qquad \sqrt{(q_t+1)(q_s+1)} \leq \frac{32\pi}{k_s} \tag{4.5}$$

Das erste Problem lässt sich durch die Realisierung eines Zweipassrenderingverfahrens lösen. Der diffuse Anteil wird in einem zweiten Fragmentsha-

[3] auch das in Abschnitt 3.2.4 vorgestellte Verfahren unter Verwendung von lediglich drei Texturen kann nicht umgesetzt werden, da hierfür die 12 möglichen Befehle nicht ausreichend sind

der berechnet und additiv dem spekularen Anteil hinzugefügt. Die Performance halbiert sich dadurch annähernd. In der Realisierung wurde für den spekularen und den diffusen Anteil jeweils eine eigene Klasse erzeugt.

Das zweite Problem lässt sich prinzipiell nicht lösen. Die Wahl der Parameter q_s und q_t ist bei der hier vorliegenden Realisierung sehr stark eingeschränkt. Schöne spekulare Beleuchtungseffekte ergeben sich erst bei Werten von q_s und q_t, die mit dieser Einschränkung nicht möglich sind. Somit sind die Anwendungsmöglichkeiten des im Rahmen dieser Arbeit implementierten Ashikhmin Modells stark eingeschränkt. Es sind Ansätze denkbar, die C_a von vornherein durch einen Wert teilen, um C_a unter den Wert 4 zu bringen und anschliessend diesen Wert wieder durch Multiplikation hinzufügen. Allerdings ist auch die Wahl dieses Wertes auf den Bereich $[-4, 4]$ beschränkt[4], und entsprechende Versuche im Rahmen dieser Arbeit führten zu keinem befriedigenden Ergebnis.

Beispiele des Ashikhmin Shadings sind in Abbildung 4.7 und 4.8 dargestellt.

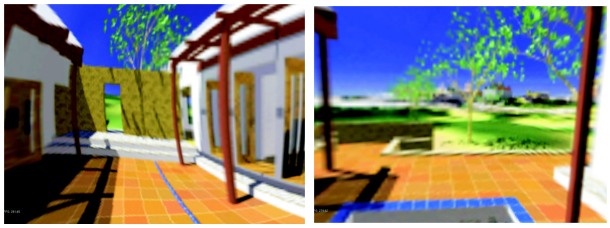

Abb. 4.9: Die in den folgenden Abbildungen verwendete Cubemap-Szene ist hier ausschnittsweise dargestellt. Links: Front. Rechts: Back. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

4.4.5 Reflektion und Brechung

Die Realisierung von Reflektion und Brechung unterscheidet sich ein wenig von den beschriebenen vorhergehenden Realisierungen: Die aufwendigen Rechnungen, die in Anlehnung an Abschnitt 3.3.2 und 3.3.3 durchgeführt werden müssen, könnten prinzipiell auch mit Hilfe von Funktionen-Maps durchgeführt werden. In dieser Arbeit wurde jedoch eine Berechnung des

[4]die Verwendung des Swizzleoperators lässt an dieser Stelle bitweises Shiften um 3 Bit, also den Wert 8 zu

Reflektions- bzw. Brechungsvektors im Vertexshader gewählt, wobei die anschliessende Interpolation beim Übergang vom Vertex- zum Fragmentshader den zum aktuellen Fragment gehörigen Reflektions- bzw. Brechungsvektor bestimmt. Diese Vorgehensweise stellt sicher, dass keine ungewollten Artefarkte durch die 8-Bit Beschränkung der Funktionen-Maps entstehen.

Abb. 4.10: Reflection Mapping: Eine Kugel reflektiert die in Abbildung 4.9 dargestellte Szene. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

Die Berechnung des Brechungsvektors im Vertexshader sei im Folgenden in Anlehnung an Gleichung 3.46 gegeben:

$$\hat{\mathbf{r}} = \left(\frac{n_i}{n_r} \cdot \langle \hat{\mathbf{N}}, \hat{\mathbf{E}} \rangle - \sqrt{1 - \frac{n_i^2}{n_r^2}(1 - \langle \hat{\mathbf{N}}, \hat{\mathbf{E}} \rangle^2)} \right) \cdot \hat{\mathbf{N}} - \frac{n_i}{n_r} \cdot \hat{\mathbf{E}} \qquad (4.6)$$

```
// refraction in object space
float n_c[4] = {refractionIndicesRatio, refractionIndicesRatio, refractionIndicesRatio, .0};
GLuint n = glGenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_CONSTANT_EXT, GL_FULL_RANGE_EXT, 1);
glSetLocalConstantEXT(n, GL_FLOAT, n_c);   // n=n1/n2

float one_c[1] = {1.0};
GLuint one = glGenSymbolsEXT(GL_SCALAR_EXT, GL_LOCAL_CONSTANT_EXT, GL_FULL_RANGE_EXT, 1);
glSetLocalConstantEXT(one, GL_FLOAT, one_c);

GLuint v1 = glGenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);
GLuint v2 = glGenSymbolsEXT(GL_VECTOR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);
GLuint s1 = glGenSymbolsEXT(GL_SCALAR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);
GLuint s2 = glGenSymbolsEXT(GL_SCALAR_EXT, GL_LOCAL_EXT, GL_FULL_RANGE_EXT, 1);

glShaderOp2EXT(GL_OP_DOT3_EXT, s1, eyeVec, normalObjectSpace);// s1 = N.L

glShaderOp2EXT(GL_OP_MUL_EXT,  s2, s1, s1);                   // s2 = (N.L)^2
glShaderOp2EXT(GL_OP_MUL_EXT,  s1, s1, n);                    // s1 = n (N.L)

glShaderOp2EXT(GL_OP_SUB_EXT,  s2, one, s2);                  // s2 = 1-(N.L)^2

glShaderOp2EXT(GL_OP_MUL_EXT,  s2, s2, n);                    // s2 = (1-(N.L)^2) * n
glShaderOp2EXT(GL_OP_MUL_EXT,  s2, s2, n);                    // s2 = (1-(N.L)^2) * n^2
```

```
glShaderOp2EXT(GL_OP_SUB_EXT,  s2, one, s2); // s2 = 1 - (1-(N.L)^2) * n^2

glShaderOp2EXT(GL_OP_SUB_EXT,  s1, s1, s2);// s1 = n(N.L) - [sqrt] (1 - (1-(N.L)^2) * n^2)

glShaderOp2EXT(GL_OP_MUL_EXT,  v1, normalObjectSpace, s1);    // v1 = N * s1

glShaderOp2EXT(GL_OP_MUL_EXT,  v2, eyeVec, n);                // v2 = V * n1/n2

glShaderOp2EXT(GL_OP_SUB_EXT, refraction, v1, v2);            // v1 = v1 - v2
```

Wie im Codebeispiel zu erkennen ist, wurde an dieser Stelle mangels eines entsprechenden Befehls die Wurzelfunktion im Bereich von $[0, 1]$ durch die Funktion $f(x) = x$ ersetzt. Eine Verwendung der existierenden und eigentlich zu Normierungen genutzten Funktion $\frac{1}{\sqrt{(x)}}$ und anschliessender Division mit Hilfe von $\frac{1}{x}$ führte nicht zu den gewünschten Ergebnissen, da diese Verknüpfung aufgrund von Rundungsfehlern fast immer den definierten Wert Unendlich liefert. Diese lineare Approximation erscheint im Rahmen der Echtzeitcomputergraphik legitim und äussert sich in keinster Weise in stark wahrnehmbaren Effekten.

Die Reflektion und Brechung wurden in nur einem einzigen Objekt implementiert, um eine Kombination dieser Erscheinungen, wie sie in der Realität häufig vorkommen, ohne einen starken Performanceverlust[5] zu ermöglichen. Dabei wird die entsprechende Berechnung nur dann durchgeführt, wenn der jeweilige Intensitätskoeffizient grösser als 0 ist.

Zusätzlich wurde die Möglichkeit des additiven Texturemappings im Rahmen der Klasse `rts_reflectionrefraction` realisiert. Beispiele sind in den Abbildungen 4.9 bis 4.11 dargestellt.

4.4.6 Diskussion

In den vorhergehenden Abschnitten wurden verschiedene Aspekte der Implementierung der unterschiedlichen Beleuchtungsmodelle vorgestellt. An dieser Stelle sei nochmals der große Vorteil der vorgestellten Prinzipien erwähnt: Mit den beschriebenen Verfahren wird nahezu eine per-Pixel[6] Berechnung des entsprechenden Beleuchtungsmodells ermöglicht. Dabei sollte aber nicht vergessen werden, dass die beschriebenen Verfahren jeweils nur eine Beleuchtung mit einer einzigen Lichtquelle gestatten. Weitere Lichtquellen müssen entweder, wenn es der verwendete Fragmentshader erlaubt, direkt im Rahmen einer Erweiterung implementiert oder durch

[5] etwa durch Verwendung eines Zwei-Pass-Renderingverfahrens

[6] hängt auch von der Größe der entsprechenden Funktionen-Map ab

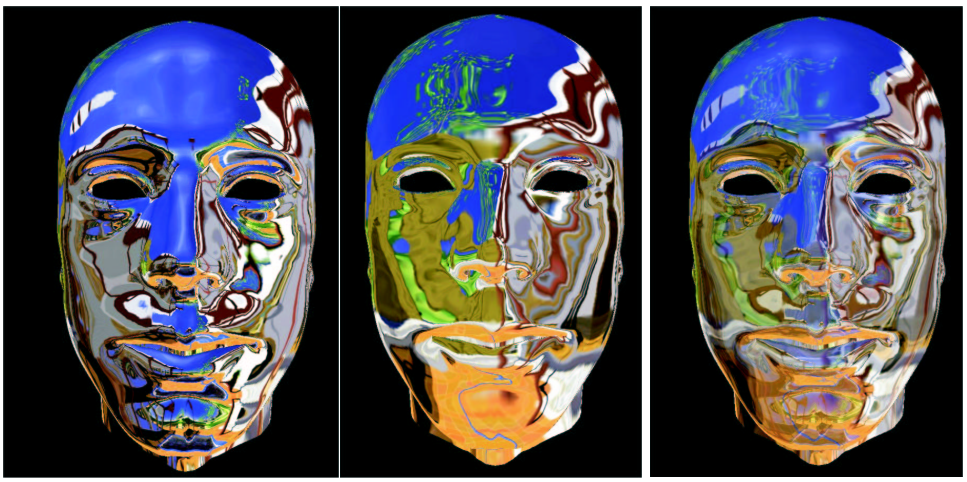

Abb. 4.11: Reflection/Refraction Mapping: Die in Abbildung 4.9 dargestellte Szene wird reflektiert (links), gebrochen (mitte) und reflektiert und gebrochen (rechts). Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shader`.

Verwendung von Multipass-Renderingläufen realisiert werden. Die letztere Möglichkeit verringert dabei die möglichen Frameraten erheblich (siehe dazu auch das Beispielprogramm `simpleMultipleLights`).

Die Vorteile liegen in den Möglichkeiten verschiedener Beleuchtungsmodelle, die von der Graphikkarte direkt nicht unterstützt werden. Zudem sind alle Verfahren den entsprechenden Bedürfnissen nach erweiterbar: So ist eine Lightattenuation mit wenig Aufwand realisierbar. Auch die im Rahmen des Phong Shading vorgestellten Erweiterungen wie per-Pixel Specular Map (glossy map) oder Bumpmapping können auf die gleiche Weise den anderen Beleuchtungsmodellen hinzugefügt werden.

Auf diese Art und Weise werden zahlreiche Gestaltungsmöglichkeiten geschaffen, die nicht unmittelbar von den vorgegebenen Fähigkeiten der Graphikkarte abhängig sind. Diese Möglichkeiten werden mit zunehmender Standardisierung und Leistungsfähigkeit der Hardwareshader immer vielfältiger werden.

4.5 Implementierung: Schlagschatten

In diesem Abschnitt werden die entwickelten Klassen `rts_shadow` und `rts_shadowmap` vorgestellt und die während der Implementierung aufgetretenen Probleme diskutiert. In `rts_shadow` wurde dabei das vorgestellte Verfahren der Projektiven Schlagschatten und in `rts_shadowmap` das Prinzip des Shadowmappings realisiert.

Die beiden hier vorgestellten Schattierungsverfahren basieren nicht auf der Verwendung eines Vertex- oder Fragmentshaders. Beispiele zu den verschiedenen Verfahren sind in den Abbildungen 4.12 bis 4.14 dargestellt.

4.5.1 Projektive Schatten

Die Projektionsebene, auf die die Schlagschatten projiziert werden, wird mit Hilfe von drei Punkten, die auf der Ebene liegen, übergeben. Mit Hilfe dieser Punkte wird die Ebenengleichung in `findPlane()` und anschliessend die zugehörige Projektionsmatrix in `shadowMatrix()` bestimmt (vgl. 3.4.1).

Da es sich bei dem Verfahren der Projektiven Schlagschatten um ein Zwei-Pass-Renderingverfahren handelt, muss die Szene zweimal gerendert werden. Für den Schattenprojektionsrenderpass werden die entsprechenden Projektionsparameter mit `Bind()` gesetzt und mit `Release()` wieder freigegeben. Wie bereits in Abschnitt 3.4.1 besprochen, kann die Verwendung des Stencilbuffers über die Ebene hinausragende Schatten beseitigen. Dazu wird zunächst die Schattenebene gezeichnet, um die entsprechenden Stencil-Bits zu setzen. Anschliessend werden die projizierten Schatten nur bei gesetztem Stencil-Bit gerendert.

Die Verwendung von

```
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(-2.0,1.0);
```

beseitigt das bereits besprochene Problem der gleichen z-Werte von projizierten Objekten (Schlagschatten) und Schattenebene.

4.5.2 Shadow Mapping

Die Realisierung des Shadowmapping erfolgt den in Abschnitt 3.4.2 vorgestellten Konzepten entsprechend. Da es sich auch bei diesem Verfahren

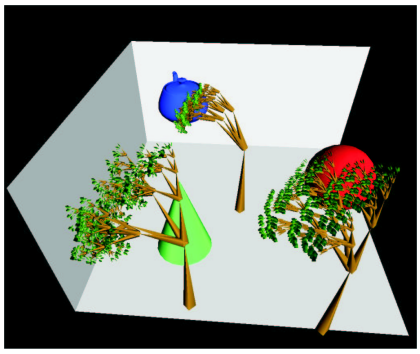

Abb. 4.12: Schlagschatten: Hier ist die Szene ohne Schatten dargestellt. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shadow`.

in der Regel (bei dynamischen Lichtquellen/Objekten) um ein Zwei-Pass-Renderingverfahren handelt, wird zunächst die Shadowmap erzeugt und anschliessend die schattierte Szene gerendert. Das Erzeugen der Shadowmap geschieht folgendermassen:

```
createShadowmapBind (lightPos);
    draw_Scene();
createShadowmapRelease ();
```

Anhand der übergebenen Lichtquellenkoordinaten wird die Szene aus Sicht der Lichtquelle gerendert. Paramter wie field-of-view etc. können dabei mit den Methoden

```
void setLightPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)
void setLightLookAt(GLdouble centerX, GLdouble centerY, GLdouble centerZ,
                                          GLdouble upX, GLdouble upY, GLdouble upZ)
```

entsprechend den OpenGL-Befehlen `gluPerspective` und `gluLookAt` gesetzt werden. Um den späteren Vergleich der Tiefenwerte korrekt durchführen zu können und nicht wegen etwaigen Fliesskommaungenauigkeiten Artefarkte zu erhalten, kann zusätzlich ein Paramter mit `setzFarMappingFactor(float f)` gesetzt werden, der mit dem Wert `zFar` während des eigentlichen Renderns multipliziert wird (also etwas über 1 liegen sollte) und damit einen korrekten Vergleich der Tiefenwerte ermöglicht.

Während des Renderns werden die Projektionstexturkoordinaten mit Hilfe des OpenGL-Paramters `GL_EYE_LINEAR` erzeugt. Der Vergleich der Tiefenwerte während des Mappings erfolg mit

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE_ARB, GL_COMPARE_R_TO_TEXTURE_ARB);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC_ARB, GL_LEQUAL);
```

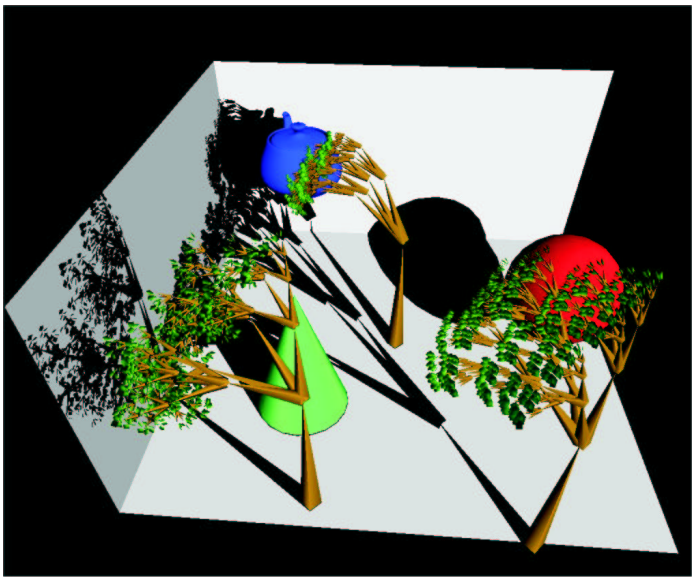

Abb. 4.13: Projektive Schlagschatten: Die Objekte werfen keine Schatten aufeinander oder auf sich selber. Lediglich auf die Ebenen werden Schlagschatten geworfen. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shadow`.

Falls der Vergleich negativ ausfällt, wird an dem betreffenden Pixel nichts gezeichnet. Hier würde im Rahmen einer Erweiterung der diffuse Lichtanteil gerendert werden.

Die vorliegende Implementierung wurde wegen der Unabhängigkeit von einem Vertex- oder Fragmentshader neben der ATI Radeon Graphikkarte auch auf einer nVidia GeForce 4 Ti4800 getestet. Wie sich gezeigt hat, ist die Hardwarerealisierung des OpenGL-Befehls `glReadPixels()` auf der nVidia-Graphikkarte wesentlich schneller. Mit dieser werden bei komplexeren Szenen Frameraten um die $20Hz$ erreicht, auf der ATI Radeon hingegen lediglich einstellige Frameraten. In der Praxis allerdings würde dieser Unterschied keine Relevanz besitzen, da statt der Verwendung von `glReadPixels()` einer Verwendung des pBuffers Vorrang gewährt wird. Dieser führt die entsprechenden Kopier-Befehle direkt auf dem Graphikartenspeicher aus. Der im Rahmen dieser Arbeit gewählte Weg hingegen führt über den Hauptspeicher und anschliessend wieder zurück auf den Graphikkartenspeicher und ist damit deutlich langsamer. Auf eine Verwendung des pBuffers im Rahmen dieser Studienarbeit wurde jedoch verzichtet, um den Implementierungsaufwand nicht zu gross werden zu lassen.

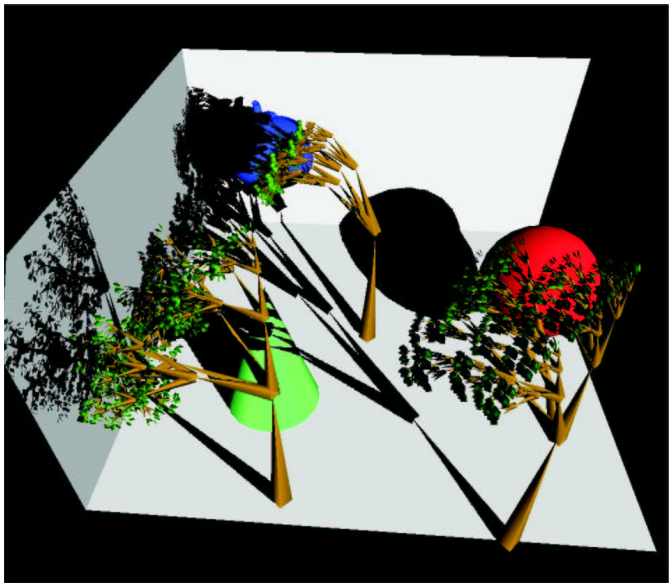

Abb. 4.14: Shadowmapping: Die Schlagschatten werden korrekt gerendert. Die Objekte werfen auch Schlagschatten aufeinander und auf sich selber. Der Unterschied ist deutlich im Vergleich zu Abbildung 4.13 zu erkennen. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `Real Time Shadow`.

4.6 Performance

In diesem Abschnitt werden einige Laufzeitmessung der im Rahmen dieser Arbeit entwickelten Programme vorgestellt. Alle Messungen wurden auf einem Pentium IV/1700MHz mit einer auf dem Chip Radeon 9800 Pro basierenden Graphikkarte bei einer Auflösung von 900×900 durchgeführt.

Als Beispielobjekte wurden die Folgenden gewählt:

- Q1: Quadrat, bestehend aus zwei Dreiecken
- Q2: Quadrat, bestehend aus 882 Dreiecken mit Variationen der Tangenten und Binormalen
- Q3: Quadrat, bestehend aus zwei Dreiecken, lediglich ein Fünftel der dargestellten Fläche einnehmend
- Quader, bestehend aus 12 Dreiecken
- Head (siehe z.B. Abbildung 4.1), Anzahl der Dreiecke leider unbekannt, soll aber einen Eindruck der Laufzeiten für komplexere Objekte vermitteln

Die ersten beiden Quadrate wurden sowohl mit als auch ohne Texturemapping gemessen (Q1, Q1mT, Q2, Q2mT). Die Objekte wurden bildfüllend dargestellt. Lediglich Q3 und der Quader nahmen etwa nur ein Fünftel der dargestellten Fläche ein.

Bei den Schlagschattenverfahren wurde einmal die in Abbildung 4.14 gezeigte Szene gemessen und einmal die gleiche Szene, allerdings nur aus der Kugel und den drei Ebenen bestehend.

	Q1	Q1mT	Q2	Q2mT	Q3	Quader	head
Phong	580	540	610	580	890	780	350
Cook Torrance	390	410	430	450	820	630	210
Banks	540	500	580	550	860	760	300
Ashikhmin	300	290	320	320	730	530	130
Reflektion	730	610	730	640	910	870	230
Brechung	750	630	750	670	920	890	270
Refl. u. Br.	750	630	750	670	920	890	270

Tabelle 4.4: Laufzeiten der Shader: Angaben in FPS

In der folgenden Tabelle sind die Laufzeiten im Verhältnis zum Phong-Shading dargestellt.

	Q1	Q1mT	Q2	Q2mT	Q3	Quader	head
Phong	1	1	1	1	1	1	1
Cook Torrance	0,67	0,75	0,70	0,78	0,92	0,80	0,60
Banks	0,93	0,92	0,95	0,94	0,97	0,97	0,85
Ashikhmin	0,52	0,54	0,52	0,55	0,82	0,68	0,37
Reflektion	1,26	1,13	1,20	1,10	1,02	1,12	0,65
Brechung	1,29	1,17	1,23	1,16	1,03	1,14	0,77
Refl. u. Br.	1,29	1,17	1,23	1,16	1,03	1,14	0,77

Tabelle 4.5: Laufzeiten der Shader: Verhältnis zu Phong

Diskussion Die Grösse der jeweiligen Darstellung und die damit verbundene Interpolation ist maßgeblich entscheidend für die Geschwindigkeit (siehe Q3 und Quader). Das Ashikhmin-Shading-Verfahren ist, bedingt durch seinen Zwei-Pass-Ansatz, das mit Abstand langsamste Shadingverfahren. Das Cook-Torrance-Verfahren ist mit seinen drei

	Q2mT	head
5 Phong Lights	93	110
Cook-Torrance RGB	90	190

Tabelle 4.6: Laufzeiten der erweiterten Shadingverfahren

Texturzugriffen das zweitlangsamste Verfahren. Banks- und Phong-Shading unterscheiden sich nur wenig - ihr prinzipieller Aufwand ist auch nahezu gleich. Die Reflektion und Brechung ist sehr schnell, wenn nur wenige Dreiecke gezeichnet werden: Hier macht sich die Berechnung innerhalb des Vertexshaders (und nicht innerhalb des Fragmentshaders wie bei den anderen Verfahren) bemerkbar. Mit stark zunehmender Anzahl der Dreiecke wird dieser Vorteil jedoch zu einem kleinen Nachteil (siehe head). Die erweiterten Shadingverfahren sind Multipass-Verfahren und damit relativ langsam.

Die Messwerte weisen im Vergleich der verschiedenen Objekte auch einige Inkonsistenzen auf: So wird die Annahme, dass die jeweiligen Verhältnisse der Renderingzeiten (hier im Verhältnis zum Phong-Shading) einigermassen konstant seien, nicht bestätigt. Die Abweichungen von Q3 und dem Quader in diesem Punkt könnte durch die massgebliche Einflussnahme der Grösse der dargestellten Fläche zur Renderingzeit erklärt werden. Die Abweichung des head könnte durch die gestiegene Komplexität des Objektes im Vergleich zu den Quadraten (alle Dreiecke in einer Ebene) erklärt werden, so dass eventuell verschiedene Optimierungsschritte der Graphikkarte nicht mehr durchgeführt werden können.

Zusätzlich ist bemerkenswert, dass die Berechnung lediglich der Reflektion langsamer ist, als die Berechnung von Reflektion und Brechung zusammen, obwohl die beiden Algorithmen vollständig unabhängig voneinander sind und der Aufwand wesentlich grösser ist.

Diese Inkonsistenzen können an dieser Stelle jedoch nicht geklärt werden - die Ursachen sind wohl auf den Optimierungseinheiten der Graphikkarte zu suchen. Die repräsentativsten Messergebnisse sind demnach diejenigen, die mit dem head gemacht wurden, da dieses Objekt durch seine Komplexität den praktischen Anwendungen am nächsten kommt.

Das Shadowmapping ist auf der Radeon-Graphikkarte aus den erwähnten Gründen sehr langsam. Auf der nVidia-Graphikkarte ist das Shadomapping

	Szene komplett	nur Kugel
ohne Schatten	25	370
projektive Schatten	8	250
Shadowmapping	1,4	1,5

Tabelle 4.7: Laufzeiten der Schlagschattenverfahren (gemessen auf: Radeon 9800 Pro): Angaben in FPS

	Szene komplett	nur Kugel
ohne Schatten	31	50
projektive Schatten	13	43
Shadowmapping	15	47

Tabelle 4.8: Laufzeiten der Schlagschattenverfahren (gemessen auf: nVidia GeForce 4 Ti4800): Angaben in FPS

ein wenig schneller als die projektiven Schlagschatten: Da drei Schattenebenen verwendet werden, ist ein Vier-Pass-Rendering vonnöten, beim Shadowmapping jedoch nur ein Zwei-Pass-Verfahren. Interessant sind auch die Unterschiede zwischen den beiden Graphikkarten, die sicherlich durch unterschiedliche Hardwarerealisierungen der benutzten Befehle entstehen - eine Diskussion dieser würde aber an dieser Stelle zu weit führen.

4.7 Die entstandenen Beispielprogramme

Neben den in den vorherigen Abschnitten beschriebenen Klassen sind im Rahmen dieser Studienarbeit folgende Programme entwickelt worden, die die Funktionalität der entwickelten Klassen beispielhaft aufzeigen und Interaktivitäten wie Rotation und Variation der Parameter ermöglichen:

- `Real Time Bumpmapping`
- `Real Time Shader`
- `Real Time Shadowing`

Im Folgenden werden die im Rahmen dieser Arbeit entwickelten und implementierten Programme kurz vorgestellt. In den nächsten drei Abschnitten werden dabei die Demonstrationsprogramme vorgestellt, die die in dieser Arbeit beschriebenen Verfahren anwenden und interaktiv darstellen. Der

darauf folgende Abschnitt stellt einige Programme vor, die über die einfache Darstellung eines einzigen Verfahrens hinausgehen. Die jeweilige `.exe` Datei ist im zugehörigen `binary`-Verzeichnis zu finden. Die Beispielprogramme ermöglichen es, jegliche Parameter der beschriebenen Verfahren frei einzugeben und verschiedene Objekte oder Szenen mit den unterschiedlichen Algorithmen darzustellen.

4.7.1 `Real Time Bumpmapping`

Dieses Programm realisiert per-Fragment Phongshading mit Texturen, einer per-Fragment-Specularmap und Bumpmapping. Dabei kann interaktiv zwischen verschiedenen Texturen, Specularmaps und Bumpmaps gewählt werden. Die Objekte können mit der Maus rotiert werden.

4.7.2 `Real Time Shader`

Das Programm `Real Time Shader` demonstriert alle in dieser Arbeit entwickelten Shading-Verfahren. Dabei kann interaktiv zwischen folgenden Shading-Verfahren gewählt werden:

- Phong Shading
- Banks Shading
- Cook-Torrance Shading
- Ashikhmin Shading
- Reflection Shading
- Refraction Shading

Dargestellt werden verschiedene mit 3D-Studio Max erstellte Objekte. Falls bei diesen Texturkoordinaten definiert sind, können verschiedene Texturen auf die Objekte gemappt werden. Jegliche Parameter der verschiedenen Shading-Verfahren können eingegeben werden. Zudem kann die Farbe der Lichtquelle gesetzt werden. Auto-Rotation der Objekte und Animation der Lichtquelle sind möglich. Die Objekte können interaktiv rotiert werden.

4.7.3 `Real Time Shadow`

Dieses Programm realisiert die beiden Schattierungsverfahren Projektive Schattendarstellung und Shadowmapping. Es wird eine Beispielszene mit verschiedenen darstellbaren Objekten gerendert. Es kann zwischen den beiden

Schattenwurfverfahren und keiner Schattendarstellung gewählt werden. Die Szene kann interaktiv rotiert werden. Die Animation der Objekte und der Lichtquelle kann aktiviert/deaktiviert werden.

4.7.4 Weiterführende Anwendungsmöglichkeiten

Neben den einfachen Beispielprogrammen, die im vorhergehenden Abschnitt kurz angesprochen wurden, wurden im Rahmen dieser Arbeit einige weitere Möglichkeiten der entwickelten Klassen beispielhaft getestet. Diese Programme besitzen lediglich exemplarischen Charakter und sollen nur Ansätze für die vielfältigen Möglichkeiten der entwickelten Klassen darstellen:

- `simpleMultipleLights` (siehe Abb. 4.15 und 4.16)
- `simpleScene` (siehe Abb. 4.17)
- `simpleSoftShadows` (siehe Abb. 4.18)
- `simpleCookTorranceRGB` (siehe Abb. 4.19)

Abb. 4.15: Phong Shading mit 5 verschiedenfarbigen Lichtquellen. Zugrunde liegt ein 5-Pass-Renderingverfahren. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `simpleMultipleLights`.

Die folgenden Programme sind beispielhafte Demonstrationen der erweiterten Möglichkeiten der im Rahmen dieser Arbeit entwickelten Klassen. Screenshots zu diesen Programmen sind in den Abbildungen 4.15 bis 4.19 zu finden.

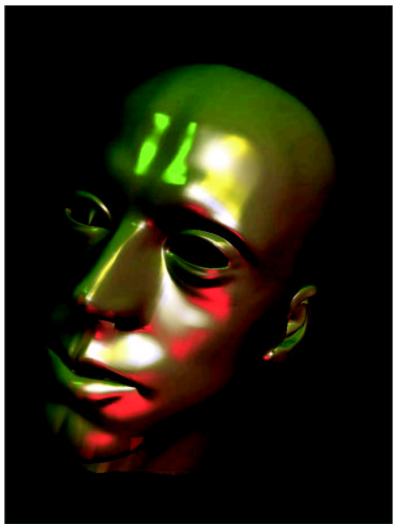

Abb. 4.16: Phong Shading mit 5 verschiedenfarbigen Lichtquellen. Zugrunde liegt ein 5-Pass-Renderingverfahren. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `simpleMultipleLights`.

4.7.4.1 `Real Time Shadow nVidia`

Dieses Programm entspricht im Wesentlichen dem Programm `Real Time Shadow`. Es wurde lediglich derart abgeändert, dass es auf nVidia Geforce Graphikkarten lauffähig ist. Getestet wurde es auf einer Graphikkarte, basierend auf dem Chip Geforce 4 TI4800.

4.7.4.2 `Simple Scene`

Dieses Programm stellt eine Szene dar, die mit verschiedenen Shadingverfahren und zusätzlich Schlagschatten gerendert wird. Interaktive Drehung ist möglich.

4.7.4.3 `Simple Multiple Lights`

Dieses Programm stellt die Möglichkeiten der entwickelten Klassen in Bezug auf Multipassrendering dar. Es werden verschiedene Objekte mit fünf verschiedenfarbigen, bewegten Lichtquellen gerendert.

4.7.4.4 `Simple Soft Shadow`

Dieses Programm demonstriert eine Erweiterung der Projektiven Schlagschattendarstellung. Unter Verwendung mehrerer Pässe und mehrerer nah beieinander liegender Lichtquellen werden weiche Schlagschatten erzeugt.

Abb. 4.17: Eine komplexere Szene unter Verwendung von Bump Mapping, Projektiven Schlagschatten, reflection/refraction Mapping, Banks-, Cook-Torrance und Phong Shading. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `simpleScene`.

4.7.4.5 `Simple Cook Torrance RGB`

Dieses Programm demonstriert das Cook Torrance Shading Verfahren mit verschiedenen Brechungsindizes für die drei Farben Rot, Grün und Blau. Zugrunde liegt ein Dreipass Renderingverfahren. Auf diese Art und Weise entfaltet das Cook-Torrance Verfahren seine Mächtigkeit in Bezug auf Dispersion.

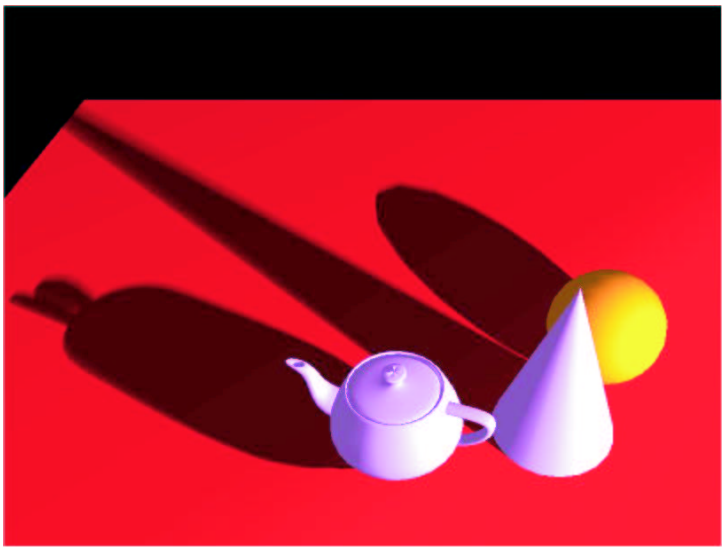

Abb. 4.18: Weiche Schatten werden durch viele nah beieinander liegende Lichtquellen erzeugt (In diesem Beispiel: 20 Lichtquellen). Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `simpleSoftShadow`.

Abb. 4.19: Cook-Torrance Beleuchtung mit verschiedenen Brechungsindizes für die 3 Farben RGB. Zugrunde liegt ein 3-Pass-Renderingverfahren. Die beiden Bilder sind mit unterschiedlichen Lichtquellenpositionen gerendert worden. Die durch den Fresnel-Effekt hervorgerufene Farbverschiebung von Rot nach Blau bei verschiedenen Einfallswinkeln des Lichtes ist zu erkennen. Gerendert mit dem im Rahmen dieser Arbeit entstandenen Programm `simpleCookTorranceRGB`.

Kapitel 5

Zusammenfassung und Ausblick

In dieser Arbeit wurden einige Möglichkeiten der Echtzeit-Computergraphik vorgestellt, die aktuelle Graphikkartengenerationen bieten. Dabei wurden verschiedene per-Pixel Shadingverfahren, basierend auf Beleuchtungsmodellen wie Phong, Banks, Cook-Torrance und Ashikhmin diskutiert und mit Hinblick auf eine, zugleich im Rahmen einer zugrunde liegenden, konkreten Implementierung, bewertet. Zusätzlich wurden physikalische Effekte und ihre Konzepte mit Bezug auf eine Implementierung wie Reflektion, Brechung und Schlagschattenwurf vorgestellt und im Rahmen dieser Arbeit auch realisiert. Alle in dieser Arbeit entwickelten Algorithmen wurden hinsichtlich ihrer Performance, ihrer Nutzbarkeit und ihrer Möglichkeiten diskutiert und bewertet.

Die hier vorgestellten Prinzipien sind als solche allgemeingültig und nicht an die hier durchgeführte Implementierung gebunden. Gerade die durch den verwendeten Fragmentshader `ATI_fragment_shader` gegebenen Einschränkungen hinsichtlich Befehlsumfang und Portierbarkeit werden sich mit dem kommenden Standard OpenGL 2.0 verringern. DirectX 9.0 z.B. stellt an dieser Stelle bereits standardisierte Vertex- und Fragmentshader zur Verfügung, die von der aktuellen Graphikkartenhardware bereits unterstützt werden. Der Befehlsumfang eines DirectX-Fragmentshaderprogrammes kann dabei bereits bis zu 1024 Befehle umfassen [5] - derart große Fragmentshaderprogramme sind allerdings zur Zeit noch nicht für die Echtzeitcomputergraphik nutzbar, da ihre Performance eher gering ist (einstellige Frameraten). Dennoch ist dieser Standard ein gutes Beispiel für das, was die zukünftigen Graphikkartengenerationen leisten werden.

Durch Verwendung von programmierbaren Shadern wird sich das Real-Time-Rendering sicher stark verändern. So ist die Berechnung von z.B. zahlreichen Lichtquellen und Beleuchtungsmodellen oder auch komplexen prozeduralen Texturen in nur einem einzigen Fragmentshader denkbar.

Literaturverzeichnis

[1] Akenine-Möller, T., Haines, E., *Real-Time Rendering*, second Edition, A K Peters, Massachusetts, 2002.

[2] ATI Inc., *Bump Mapping on ATI Rage 128*, Verfügbar im World Wide Web unter: `http://mirror.ati.com/developer/sdk/rage128sdk/Rage128BumpTutorial.html` (09.09.2003).

[3] Baker, P., *Simple Bump Mapping*, 2001. Verfügbar im World Wide Web unter: `http://www.paulsprojects.net/opengl/bumpmap/bumpmap.html` (23.09.2003).

[4] Bronstein, I.N., Semendjajew, K.A., Musiol, G., Mühlig, H., *Taschenbuch der Mathematik*, Harri Deutsch, 4. Auflage, 1999.

[5] Bertuch, M., Weiner, L., *3D-Spaß bezahlbar*, Zeitschrift c't 19/2003, Heise Verlag, 2003.

[6] Cook, R. L., Torrance, K.E., *A Reflectance Model for Computer Graphics*, ACM Transactions on Graphics (TOG), Volume 1 Issue 1, 1982.

[7] Everitt, C., Kilgard, M.J., *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*, NVIDIA Corp., Texas, 2002.

[8] Foley J.D., van Dam, A., Feiner, S.K., Hughes, J.F., *Computer Graphics - Principles and Practice*, Second Edition, Addison-Wesley, 2000.

[9] Gosseling, D., Hart, E., *`ATI_fragment_shader` Specification*, ATI Research, 2001. Verfügbar im World Wide Web unter: `http://oss.sgi.com/projects/ogl-sample/registry/ATI/fragment_shader.txt` (17.09.2003).

[10] Gosseling, D., Hart, E., ***EXT_vertex_shader*** *Specification*, ATI Research, 2001. Verfügbar im World Wide Web unter: `http://oss.sgi.com/projects/ogl-sample/registry/EXT/vertex_shader.txt` (17.09.2003).

[11] Hart, E., Mitchel, J.L., *Hardware Shading with* ***EXT_vertex_shader*** *and* ***ATI_fragment_shader***, 3D Application Research Group, ATI Research, 2001. Verfügbar im World Wide Web unter: `http://mirror.ati.com/developer/ATIHardwareShading.pdf` (17.09.2003).

[12] Hart, E., Mitchel, J.L., *Hardware Shading with* ***EXT_vertex_shader*** *and* ***ATI_fragment_shader***, ATI Research, 2001. Verfügbar im World Wide Web unter: `http://citeseer.nj.nec.com/context/2065681/0` (10.09.2003).

[13] Heidrich, W., Seidel, H.-P., *Realistic Hardware-Accelerated Shading and Lighting*, Siggraph '99. Verfügbar im World Wide Web unter: `http://www.cs.unc.edu/~andrewz/comp238/Hardware-Shading.ppt` (09.09.2003).

[14] Kilgard, M.J., *Shadow Mapping with Today's OpenGL Hardware*, Game Developer Conference, 2000. Verfügbar im World Wide Web unter: `http://www.gdc.org` (11.09.2003).

[15] Kilgard, M.J., *Robust Stencil Shadow Volumes*, CEDEC, Tokio, Japan, 2001

[16] Kolb, A., *Realtime Shading*, 2002. Verfügbar im World Wide Web unter: `http://www.fh-wedel.de/~ko/Data/rts-handout-3.pdf` (09.09.2003).

[17] Merzinger, G., With, T., *Repetitorium der Höheren Mathematik*, Binomi Verlag, Hannover, 1995.

[18] N.N., *Environment Mapping Techniques*, 2003. Verfügbar im World Wide Web unter: `www.developer.com/lang/other/article.php/2169281` (09.09.2003).

[19] Nvdia Corporation, Kilgard, M.J., *Texture Shader Specification*, 2002. Verfügbar im World Wide Web unter: `http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader.txt` (09.09.2003).

[20] Olando, M., Hart, C.J., Heidrich W., McCool, M., *Real-Time Shading*, Massachusetts, A K Peters, 2002.

[21] Press, W.H., et al., *Numerical Recipes in C, Second Edition*, Cambridge University Press, 1996. Verfügbar im World Wide Web unter: `http://www.library.cornell.edu/nr/bookcpdf.html` (05.06.2003).

[22] Rheingans, P., *Illumination*, 2001. Verfügbar im World Wide Web unter: `http://www.cs.umbc.edu/ rheingan/435/pages/res/gen-11.Illum-single-page-0.html` (11.09.2003).

[23] Schumann, H., *Realistische Bilddarstellung*, Skript zur Vorlesung, Universität Rostock, 2002.

[24] Scopigno, R., *Real Time Bump Mapping using OpenGL*, 2001. Verfügbar im World Wide Web.

[25] Seidel, H.-P., Heidrich, Wolfgang, *Hardware Shading: State-of-the-Art and Future Challenges*, 2000. Verfügbar im World Wide Web unter:
`http://www.ibiblio.org/hwws/previous/www_2000/programs.html`
(20.09.2003).

[26] Smith, B., *Geometrical shadowing of a random rough surface*, IEEE Trans. Ant. and Propagation, AP 15(5): 668-671, 1967.

[27] Stöcker, H., *Taschenbuch der Physik*, Harri Deutsch, 1998.

Anhang A

CD

Auf der beiliegenden CD (siehe letzte Seite) sind alle im Rahmen dieser Arbeit entwickelten Programme zu finden. Dies betrifft insbesondere auch die jeweiligen Sourcecodes, die verwendeten Texturen und die 3D-Studio Objekte. Zu allen Programmen existieren Microsoft Visual C 6.0 Project Files. Die `.exe`-Files sind in den jeweiligen `binary` bzw. `debug` Verzeichnissen zu finden. Zu jedem Programm existiert eine kurze `readme.txt` Datei, die die jeweilige Funktionalität und Bedienung beschreibt. Zusätzlich ist dieses Dokument als `studienarbeit.pdf` auf der CD-ROM abgelegt.