Image Based Lighting in Augmented Reality Umgebungen

Peter Supan

DIPLOMARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Digitale Medien

in Hagenberg

im Juni 2006

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 23. Juni 2006

Peter Supan

Inhaltsverzeichnis

Kurzfassung

In dieser Arbeit wird eine Applikation vorgestellt, die in Echtzeit aktualisiertes Image Based Lighting mit Beschattung in Augmented Reality Anwendungen verwendet. Es wird entweder das Bild einer Spiegelkugel oder einer Fischaugenkamera verwendet, um Information über die reale Umgebung zu bekommen. Aus diesen Bildern werden in Echtzeit Environment Maps für diffuse und glänzende Objekte generiert. Dadurch ist es z. B. möglich, reflektierende virtuelle Objekte zu rendern und das Spiegelbild des realen Betrachters in der Reflexion zu sehen. Außerdem beeinflussen Beleuchtungsveränderungen der realen Umgebung unmittelbar die Beleuchtung der virtuellen Objekte. Realtätstreue Schatten werden gerendert, indem eine Kuppel aus schattenwerfenden Lichtquellen generiert wird, deren Intensität durch die Helligkeit der realen Umgebung in Richtung der Lichtquelle gesteuert wird.

Abstract

This work presents an application which uses realtime captured Image Based Lighting with appropriate shadowing in Augmented Reality Applications. The system uses a mirrored sphere or a camera with a fisheye lens to capture the real environment. From these images environment maps for diffuse and specular lighting are created up to 60 times per second. Thus it is possible to render reflective objects and see the reflection of the real viewer in the virtual object. Moreover changes in the lighting of the real sourrounding immidiately affect the lighting of all objects. Realistic shadows are calculated by creating a dome of shadow casting lights and sampling the environment map to achieve correct shadow intensities.

Kapitel 1

Einleitung

1.1 Motivation und Zielsetzung

In dieser Arbeit wird die Verwendung von Image Based Lighting in Augmented Reality Anwendungen behandelt. Augmented Reality ist eine Möglichkeit, die reale Welt mit virtuellen Grafiken zu erweitern. Diese Grafiken können einfache Information über die reale Umgebung [Paper] oder aber photorealistische Renderings von virtuellen Objekten wie Prototypen von Autokarosserien oder Architekturmodellen sein (siehe [Azu04] für eine Definition von Augmented Reality). In vielen AR-Applikationen ist gewünscht, dass virtuelle Objekte nahtlos in die reale Umgebung einfügt werden. Eine Voraussetzung dafür ist, dass virtuelle und reale Objekte konsistent beleuchtet werden. Wir versuchen die Beleuchtung von virtuellen Objekten mit Image Based Lighting (IBL) zu verbessern. In IBL wird ein Bild oder ein Video der Umgebung verwendet, um Beleuchtung zu berechnen. IBL findet Verwendung in vielen Spielfilmen und eindrucksvollen Kurzfilmen. Eine detaillierte Beschreibung der Technik wird von Paul Debevec in [Deb98] und [Deb02] gegeben. In [DTG+04] ist ein Bericht über die Produktion eines Kurzfilmes, der IBL als Technik verwendet, zu finden. In dieser Produktion wird die Beleuchtung der Umgebung als Vorbereitungsschritt erledigt, was relativ viel Zeit in Anspruch nimmt. Im Gegensatz dazu nimmt unsere Applikation die Beleuchtung in Echtzeit auf, und erhält mehrere Male pro Sekunde ein neues Bild. Dies erlaubt der realen Umgebung, die Beleuchtung der virtuellen Objekte unmittelbar zu beeinflussen. So ist es z. B. möglich, dass eine Person ihr eigenes Spiegelbild in einem der virtuellen Objekte sieht.

1.2 Verwandte Arbeiten

Verwandte Arbeiten gibt es in den einzelnen Bereichen Image Based Lighting und Schattendarstellung sowie in diesen beiden Bereichen kombiniert mit Augmented Reality.

Sehr viel Forschung im Bereich IBL wurde von Paul Debevec[1] betrieben. Neben den in der Einleitung erwähnten Arbeiten zählen auch einige Arbeiten über High Dynamic Range Bilder und Extraktion von Lichtquellen aus Environment Maps [Deb05] zu seinen Publikationen.

Die verwendete Schattentechnik mit einer großen Anzahl an Lichtern, die ihre Intensität abhängig von der Environment Map verändern, wurde in einer ähnlichen Form von Gibson et al. [GHH03] verwendet.

Auch Kakuta et al. [KOI05] verwenden eine Kuppel aus Lichtern, um realistische Beleuchtung von Architektur zu ermöglichen.

Von Drab [Dra03] wurde ein Framework zum Rendering von Schatten in Augmented Reality entwickelt. Bemerkenswert dabei ist der Schattenwurf von realen auf virtuelle Objekte und umgekehrt.

Karlsson et al [KS05] entwickelten eine Applikation, in der Image Based Lighting mit einer Kamera möglich ist. Sie verwenden Schattentexturen zur Schattendarstellung.

1.3 Aufbau der Arbeit

Zu Beginn der Arbeit wird im Konzeptteil eine kurze Einführung in Image Based Lighting gegeben. Danach wird auf unterschiedliche Möglichkeiten von Setups, um IBL durchzuführen, eingegangen. Die Pipeline von notwendigen Arbeitsschritten wird erklärt und einzelne Algorithmen beschrieben.

Im folgenden Implementierungsteil wird die konkrete Umsetzung der zuvor im Konzeptteil beschriebenen Algorithmen erläutert. Die Struktur einer Beispielapplikation in C++, mit *OpenGL* als Grafik API und Cg als Shader Sprache, wird erläutert.

Die Ergebnisse und Erkenntnisse dieser Implementierung werden im folgenden Kapitel 4 beschrieben, mögliche Erweiterungen und Verbesserungen werden aufgelistet.

[1] www.debevec.org

Kapitel 2

Konzept

2.1 Image Based Lighting

Motivation Einer der wichtigsten Aspekte, um virtuelle Objekte real wirken zu lassen, ist die Beleuchtung. In der traditionellen Computergrafik werden zur Beleuchtung Punktlichtquellen verwendet. Diese haben eine bestimmte Position und keine Ausdehnung. Daher erhält ein Objekt Licht nur von einigen wenigen, festgelegten Punkten im Raum. Es ist schwierig, nicht punktförmige Lichtquellen (wie z. B. Neonröhren oder bedeckten Himmel) zu simulieren, da eine große Anzahl an Punktlichtquellen dafür notwendig wäre (siehe Abb. 2.1) dafür. In der realen Welt erhält ein Objekt nicht nur Photonen, die direkt von Lichtquellen ausgesandt werden, sondern auch Photonen, die von anderen Objekten reflektiert wurden.

1976 wurde von Blinn und Newell [BN76] eine Technik entwickelt, die ein Bild der Umgebung zur Beleuchtung spiegelnder Objekte verwendet. Diese Technik wurde 1984 von Gene und Hoffman [MH84] auf die Darstellung matter Oberflächen erweitert. Die Technik, Bilder zur Beleuchtung zu

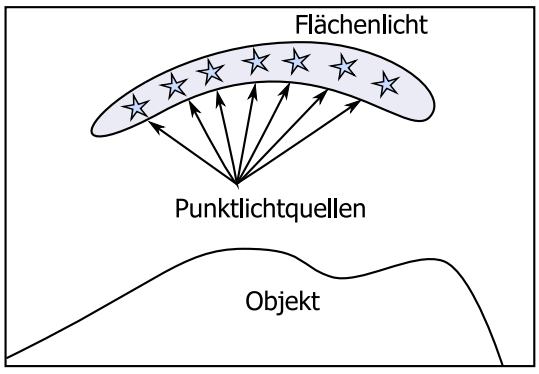


Abbildung 2.1: Ein Flächenlicht kann nur durch eine große Anzahl an Punktlichtquellen approximiert werden.

verwenden, wird Image Based Lighting (IBL) genannt.

Image Based Lighting (IBL) kann diesen Effekt simulieren, indem ein Bild der Umgebung zur Beleuchtungsberechnung verwendet wird. Daher ist es auch möglich beliebige unregelmäßig geformte Lichtquellen zu simulieren. In IBL wird normalerweise davon ausgegangen, dass die Licht-emittierende Umgebung unendlich weit von den beleuchteten Objekten entfernt ist (siehe [Fer04, Kap. 19] für eine Ausnahme). Das ist zwar theoretisch selten der Fall, die Unterschiede sind jedoch kaum sichtbar und diese Annahme ermöglicht sehr effizientes Rendering. Die Technik, Bilder zur Beleuchtung glänzender Objekte zu verwenden, wurde zum ersten Mal von Blinn und Newell [BN76] vorgestellt. Gene und Hoffman [MH84] erweiterten diese Technik auf matte Oberflächen.

2.1.1 Diffuse Beleuchtung in IBL

Ein Punkt auf einer Lambert'schen Oberfläche reflektiert Licht gleichmäßig in alle Richtungen. Er empfängt jedoch auch Licht aus mehr als einer Richtung. Es wird angenommen, dass Licht aus einer Halbkugel mit einem Raumwinkel von ungefähr 180° empfangen wird (siehe Abb. 2.2). Die Achse dieser Halbkugel ist die Oberflächennormalen $\vec{N}$ des gerenderten Punktes $\vec{P}$. Der Grund dafür ist, dass Punkte, die entlang der Oberflächennormalen liegen, den stärksten Einfluss auf die Beleuchtung haben. Der Einfluss nimmt graduell in Richtung der Tangentenebene ab.

In IBL werden alle Punkte in der Umgebung durch ein Bild approximiert. Anstatt die Beleuchtung mit einer oder mehreren Lichtquellen mit Lamberts Formeln zu berechnen, werden alle Pixel in der Umgebungstextur, die von der Oberfläche aus sichtbar sind, gesampled. Danach werden sie mit dem Kosinus des Winkels zwischen ihrer Richtung und der Oberflächennormale gewichtet. Dies hat zur Folge, dass jeder von ihnen wie eine direktionale Lichtquelle behandelt wird, die mit Lamberts Formel berechnet wird (siehe Abb. 2.2). Die Summe der Ergebnisse aller Pixel wird zur Beleuchtungsberechnung verwendet.

So viele Texturzugriffe können allerdings nicht in Echtzeit durchgeführt werden. Anstatt alle diese Texturzugriffe für jeden gerenderten Pixel durchzuführen, wird die Textur in einem Vorbereitungsschritt mit einer Faltung bearbeitet. Für jeden Texel werden alle Einträge in der Environment Map gesampled. Da jeder Texel in der Environment Map einem Vektor in die Umgebung entspricht, wird mit dem Kosinus des Winkels zwischen dem Vektor des Samples und des Vektors des aktuellen Texels gewichtet. Die Ergebnisse werden summiert, die Summe entspricht der Menge an Licht, die eine in Richtung des aktuellen Texels gerichtete Fläche erhält. Man erhält nun mit einem Texturzugriff an der Stelle des Normalvektors den korrekten Wert für diffuse Beleuchtung.

Eine andere Möglichkeit zur Vorbearbeitung nähert die Ergebnisse der

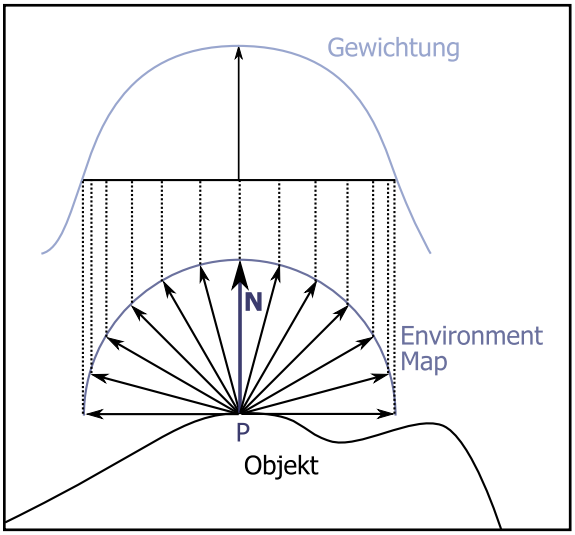


Abbildung 2.2: Jeder Texel in der dem Objekt zugewandten Seite der Environment Map muss ausgelesen werden. Die ausgelesenen Werte werden abhängig von ihrer Lage zur Oberflächennormale gewichtet.

oben genannten Technik an, indem die Environment Map mit einem Filter weichgezeichnet wird. Das liefert zwar nicht exakt die gleichen Resultate, ist aber für diffuse Beleuchtung ähnlich genug. Der große Vorteil dieser Methode ist, dass sie schneller ausgeführt werden kann als die exakte Lösung. Durch die Verwendung eines Gauss'schen Unschärfefilters erhält jedes Texel in der Environment Map Information von den es umgebenden Texeln. Für diffuse Beleuchtung muss der Kernel des Filters dazu jenen Bereich der Textur umfassen, der die Hälfte der Umgebung repräsentiert, um den Raumwinkel von 180° abzudecken.

In Abb. 2.3 wird der Filterungsvorgang für Diffus- und Glanzlichttexturen gezeigt. Dargestellt sind die Farbwerte der Environment Map als Graustufen und als Balken. Darunter sind die den Texeln entsprechenden Vektoren in die Umgebung und ihre Winkelabweichung gegenüber dem gerade bearbeiteten Texel dargestellt (der Kosinus des Winkels ist als Balkendiagramm dargestellt). Der Farbwert und der Kosinus des Winkels werden multiplitziert und die Ergebnisse summiert, um den endgültigen Helligkeitswert zu ermitteln. Das Ergebnis der Faltung wird Irradiance Map genannt (siehe [Fer04, Kap. 3] für mehr Details).

2.1.2 Glanzlichter in IBL

Glanzpunkte auf Objekten sind Reflexionen von Lichtquellen oder hellen Objekten. Daher ist es theoretisch möglich, klassisches Environment Mapping für das Rendern von Glanzlichtern zu verwenden. Das ist jedoch nur

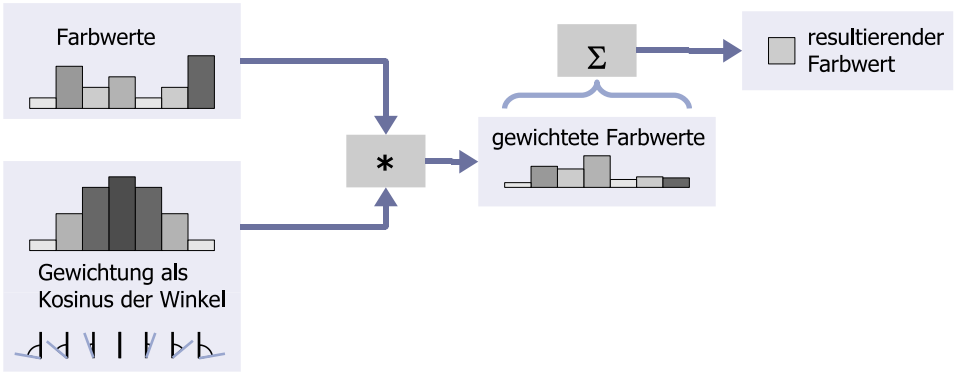


Abbildung 2.3: Filterungsvorgang für Diffus- und Glanzlichttexturen. Die Farbwerte im Filterkernel werden mit dem Kosinus ihrer Winkelabweichung gewichtet und danach summiert.

für perfekt spiegelnde Oberflächen (wie z. B. Chrom) korrekt. Die meisten Materialien absorbieren und streuen auch einen Teil des Lichts. Dadurch werden nur sehr helle Objekte wie Lichtquellen in der Reflexion sichtbar (siehe Abb. 2.4). Auch diese Objekte sind nicht komplett scharf, sondern zu einem gewissen Grad verwaschen. Der Grund dafür ist, dass nicht nur die Reflexion aus einer Richtung sichtbar ist (wie bei einem Spiegel), sondern aus allen Richtungen, mit unterschiedlicher Gewichtung. Wie in Abb. 2.5 zu sehen, erhalten die Texel in Richtung von $\vec{R}$, der die Reflexion des Augvektors $\vec{E}$ an der Oberflächennormale $\vec{N}$ darstellt, die größte Gewichtung. Alle anderen Texel werden mit dem Kosinus zwischen dem Vektor in ihre Richtung und $\vec{R}$, potenziert mit der Glanzstärke (engl. „shininess“) des Materials, gewichtet (siehe Abb. 2.5). Sie werden also so behandelt, als ob jedes von ihnen eine direktionale Lichtquelle, berechnet mit Phongs Beleuchtungsformel [Pho75] wäre. Die Summe aller Ergebnisse ist die Menge an Glanzlicht, die in Richtung der Kamera reflektiert wird.

Auch hier können nicht so viele Texturzugriffe in Echtzeit durchgeführt werden und eine Vorbearbeitung der Textur ist nötig (siehe Abschnitt 2.1.1). Analog zu Diffustexturen können alle Texel der Environment Map gesampled und gewichtet werden. Die Gewichtung ergibt sich diesmal durch den Kosinus des Winkels zwischen dem Sample und dem aktuellen Texel, potenziert mit der Shininess des Materials. Das Ergebnis ist die Menge an Licht, das in die Richtung der Kamera strahlt, wenn der reflektierte Augvektor $\vec{E}$ in die Richtung des aktuellen Texels zeigt. Auch hier bietet sich ausserdem die Alternative, das Ergebnis mit einem Unschärfefilter anznähern.

Das Problem mit Glanzlichtern ist, dass die Größe des Filter Kernels von der Shininess abhängt, die von Material zu Material variiert (siehe Abb. 2.6) für die Auswirkung verschiedener Filter auf das Erscheinungsbild eines Materials). Theoretisch muss eine eigene Textur für jedes Material erzeugt

Abbildung 2.4: Auf den meisten Oberflächen sind Glanzpunkte verwaschen. Durch Absorption von Licht sind nur helle Lichtquellen in der Reflexion sichtbar.

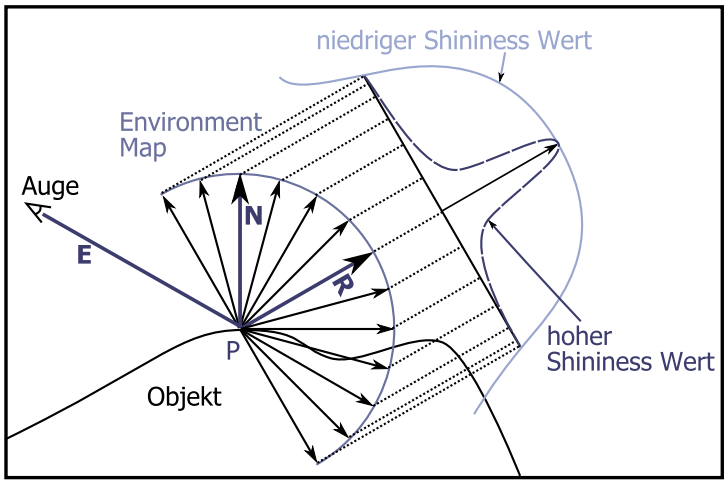


Abbildung 2.5: Gewichtung von Werten in der Environment Map für Glanzlichtexturen. Das größte Gewicht erhält der Texel in Richtung des reflektierten Augvektors. Das Gewicht der verbleibenden Texel hängt von ihrer Position und dem Shininess Wert des Materials ab.

(d) (e) (f)

Abbildung 2.6: Auswirkung unterschiedlicher Filterung auf das Erscheinungsbild eines Materials. Die Environment Map in (d) ist nur leicht gefiltert, das damit gerenderte Objekt sieht glänzend aus. Die Environment Map in (e) ist sehr stark gefiltert. Dementsprechend sieht das Material matter aus. In (f) ist eine stark abgedunkelte Environment Map zu sehen. Nur helle Lichtquellen bleiben sichtbar.

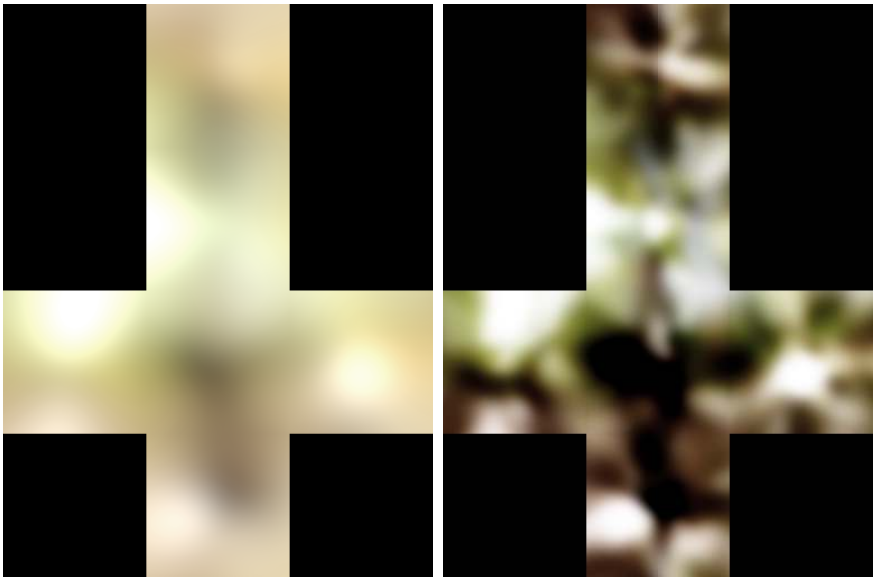
werden. Alternativ dazu können Mip-Maps erzeugt und für mattere Materialien ein niedrigeres Mip-Map Level verwendet werden. Eine weitere Möglichkeit ist, Summed Area Tables (siehe [HSC+05]) zu verwenden.

Um die Absorption von Licht im Material zu simulieren, könnte es notwendig sein, die Glanzlichttextur abzudunkeln, so dass nur helle Lichtquellen als Glanzpunkte sichtbar sind (siehe Abb. 2.7(b) und Abb. 2.6(c) für ein Beispiel einer abgedunkelten Glanzlichttextur).

2.2 Überblick über die Pipeline

In Abb. 2.8 ist ein grober Überblick über die Pipeline zu sehen. Als Eingabe dienen einer oder mehrere Videostreams. Aus diesen Videostreams werden sowohl das Bild einer in der Szene platzierten Spiegelkugel als auch die Positionierung eines Markers in der Szene extrahiert.

Das Bild der Spiegelkugel wird nachbearbeitet, um für Image Based Lighting von diffusen und glänzenden (engl. *glossy*) Oberflächen verwendet werden zu können. Das Ergebnis sind mehrere unterschiedliche Environment Maps (siehe Abb. 2.6). Mit Hilfe dieser Maps wird die Beleuchtung der vir-



(a) (b)

Abbildung 2.7: Diffus (2.7(a)) und Glanzlichttextur(2.7(b)). Es ist sichtbar, dass dunkle Bereiche in der Glanzlichttextur scharz erscheinen und nur helle Bereiche als Glanzlichter sichtbar werden. Beide Texturen sind aus der „Dawn“-Demo der nvidia Geforce FX Grafikkartenserie entnommen.

tuellen Objekte berechnet. Die Formeln von Lambert und Phong werden dabei durch einfache Texturzugriffe ersetzt. Außerdem wird Schattenwurf berechnet, indem die Environment Map zur Bestimmung von Position und Stärke von schattenwerfenden Lichtquellen verwendet wird. Die Ergebnisse der Beleuchtungs- und Schattenberechungen werden mit der Farbe des Objektes kombiniert um die endgültige Ausgangsfarbe zu bestimmen.

2.3 Setups

Drei Möglichkeiten für Setups wurden getestet. Das erste Setup verwendet eine Kamera, das zweite zwei Kameras, beide verwenden eine Spiegelkugel, um ein Bild der Umgebung zu bekommen. Das dritte Setup verwendet zwei Kameras, anstelle einer Spiegelkugel wird das Bild der Umgebung aber durch ein Fischaugenobjektiv aufgenommen.

2.3.1 Ein-Kamera Setup

Das Setup mit nur einer Kamera sieht so aus wie in Abb. 2.9 sichtbar. Eine Kamera nimmt die Szene auf, in der sich einer oder mehrere Marker sowie eine Spiegelkugel befinden. Die Spiegelkugel befindet sich in fixer

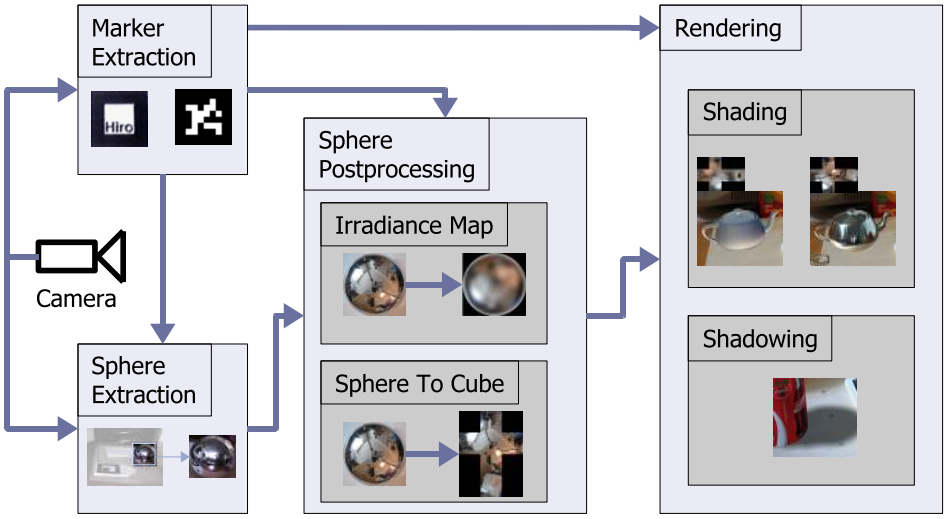
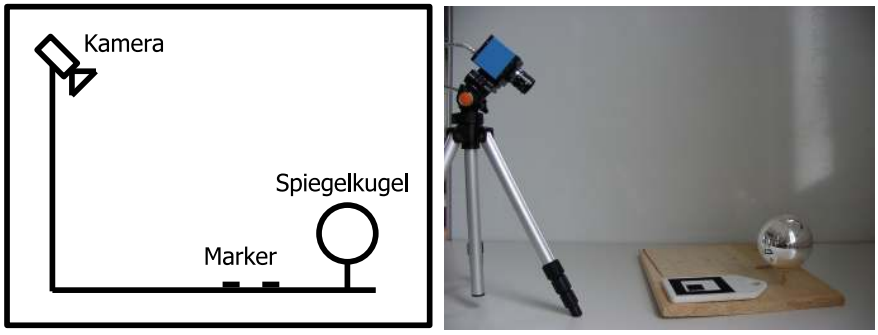


Abbildung 2.8: Überblick über die Pipeline.



(a) (b)

Abbildung 2.9: Das Setup mit einer Kamera.

Entfernung zu einem der Marker. Dadurch ist es möglich, das Bounding Rectangle der Spiegelkugel im Videobild zu bestimmen.

Vorteile des Ein-Kamera Setups

- Da dieselbe Kamera für die Aufnahme der Kugel und die Aufnahme der Szene verwendet wird, haben beide Bilder dieselben Einstellungen für Belichtung und Weißabgleich.

Nachteile des Ein-Kamera Setups

- Es ist nicht möglich, für die Aufnahme der Spiegelkugel eine andere Belichtungseinstellung zu wählen.

- Wenn die Spiegelkugel von der Kamera aus nicht sichtbar ist, kann die Environment Map nicht erneuert werden.
- Wenn die Spiegelkugel nur sehr klein sichtbar ist, ist die Auflösung der Environment Map sehr niedrig.

Bestimmen des Bounding Rectangles der Spiegelkugel im Videobild

Wie in Abb. 2.10 zu sehen, werden drei verschiedene Koordinatensysteme verwendet.

1. Das Markerkoordinatensystem, in dem alle Koordinaten relativ zum Ursprung eines Trackingsystems (in weiterer Folge Marker genannt) angegeben werden.
2. Das Kamerakoordinatensystem, dessen Ursprung die Kamera ist. Eine Matrix, die von Markerkoordinaten in Kamerakoordinaten transformiert (in weiterer Folge **M** genannt), kann durch Trackingsysteme erlangt werden.
3. Das dritte Koordinatensystem bilden Bildschirmkoordinaten. Dieses Koordinatensystem wird erreicht, indem das Kamerakoordinatensystem mit der Projektionsmatrix der Kamera (in weiterer Folge **P** genannt) multipliziert wird.

Der Mittelpunkt der Kugel in Markerkoordinaten (M_{Marker}) wird ermittelt, indem die reale Position der Kugel relativ zum Marker gemessen wird. Der Radius der Kugel r wird ebenfalls gemessen. Um den Mittelpunkt der Kugel in Kamerakoordinaten (entspricht Augkoordinaten) M_{Eye} zu erhalten, muss M_{Marker} mit der Matrix **M** multipliziert werden. Ausgehend von M_{Eye} soll nun ein Bounding Rectangle der Kugel in Bildschirmkoordinaten berechnet werden. Dafür werden zwei Punkte $(\vec{X}, \vec{Y})_{Eye}$ im Kamerakoordinatensystem ermittelt. Der Punkt

$$\vec{X}_{Eye} = M_{Eye} + \begin{pmatrix} r \\ r \\ 0 \end{pmatrix} \tag{2.1}$$

liegt in der rechten oberen Ecke des Bounding Rectangles der Spiegelkugel,

$$\vec{Y}_{Eye} = M_{Eye} - \begin{pmatrix} r \\ r \\ 0 \end{pmatrix} \tag{2.2}$$

in der linken unteren Ecke Durch eine Multiplikation mit **P** werden beide Punkte in Bildschirmkoordinaten gebracht ($\vec{X}_{Screen}$ und $\vec{Y}_{Screen}$). Beide Vektoren sind in Abb. 2.10 sichtbar.

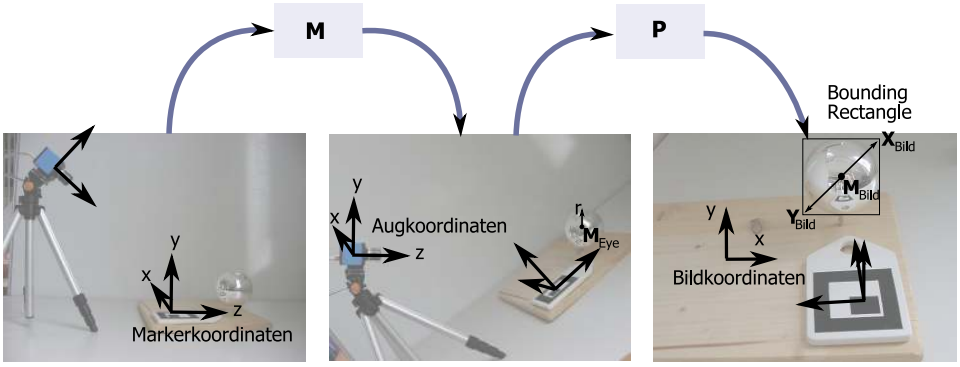


Abbildung 2.10: Die unterschiedlichen Koordinatensysteme beim Finden des Bounding Rectangles und die Transformationen dazwischen.

(a) (b)

Abbildung 2.11: Das Setup mit zwei Kameras.

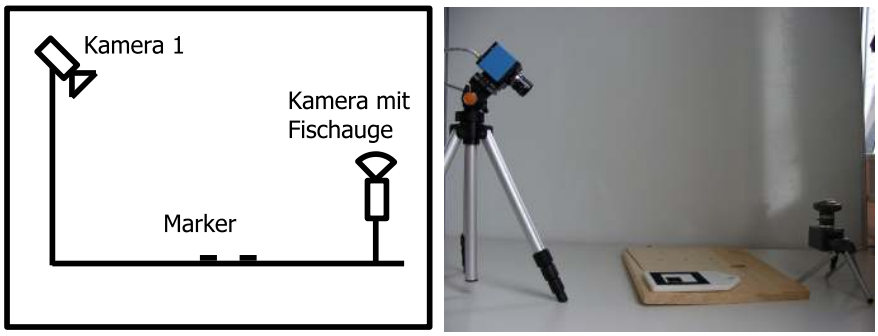
Nachdem das Bounding Rectangle der Spiegelkugel bekannt ist, wird es aus dem Kamerabild ausgeschnitten und in eine Textur kopiert. Das Ergebnis ist eine Textur, in der nur die Spiegelkugel sichtbar ist (siehe Abb. 2.6(a)). Diese Textur kann als Environment Map verwendet werden.

2.3.2 Zwei-Kamera Setup

Das Setup mit zwei Kameras ist in Abb. 2.11 sichtbar. Eine Kamera nimmt die Szene als Ganzes sowie den Marker auf, die zweite Kamera ist alleine für die Spiegelkugel zuständig und liefert die Environment Map.

Vorteile des Zwei-Kamera Setups

- Da sich die Kamera für die Spiegelkugel nicht bewegt, muss das Bounding Rectangle der Kugel nicht neu berechnet werden.



(a) (b)

Abbildung 2.12: Das Setup mit zwei Kameras und Fischaugenobjektiv.

- Es gibt keine Situationen, in denen sich die Spiegelkugel nicht mehr im Bild befindet.
- Die Auflösung der Spiegelkugel im Videobild und damit die Auflösung der Environment Map ist immer gleich hoch, unabhängig davon wo sich die Szenenkamera befindet.

Nachteile des Zwei-Kamera Setups

- Die zwei Kameras für Szene und Spiegelkugel können unterschiedliche Einstellungen für Weißabgleich und Belichtung haben. Das kann dazu führen, dass die Beleuchtung der virtuellen Objekte nicht mit der realen Umgebung übereinstimmt.

2.3.3 Zwei-Kamera Setup mit Fischauge

Das Setup mit zwei Kameras und einem Fischaugenobjektiv anstatt der Spiegelkugel ist in Abb. 2.12 zu sehen. Die erste Kamera ist wie im 2-Kamera-Setup für die Szene zuständig, die zweite nimmt die Umgebung mit dem Fischaugenobjektiv auf und liefert dadurch wieder eine Environment Map.

Vorteile des Setups mit Fischaugenobjektiv

- Die Anzahl der einzelnen Objekte im Setup verringert sich, dadurch wird es transportabler.
- Es kann nicht passieren, dass die Kamera verrutscht und dadurch keine Environment Map mehr erzeugt werden kann.
- Es gibt keinen Verlust von Bildqualität durch Abfilmen einer Spiegelkugel (die u.U. nicht perfekt sauber und reflektierend ist).

Nachteile des Setups mit Fischaugenobjektiv

- Die zwei Kameras für Szene und Environment Map können unterschiedliche Einstellungen für Weißabgleich und Belichtung haben. Das kann dazu führen, dass die Beleuchtung der virtuellen Objekte nicht mit der realen Umgebung übereinstimmt.
- Ein Fischaugenobjektiv sieht einen kleineren Bereich der Umgebung ein als eine Spiegelkugel.

2.4 Nachbearbeiten der Environment Maps

2.4.1 Erstellen von Irradiance Maps und matten Reflexionen

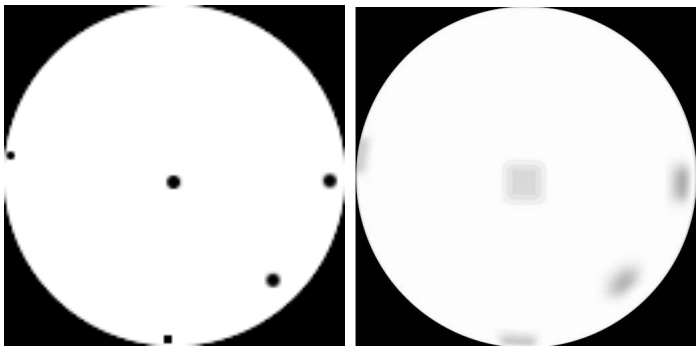
Wie bereits in Kapitel 2.1 erwähnt, kann eine Environment Map zum Rendern von perfekt spiegelnden Oberflächen verwendet werden. Durch Weichzeichnen der Environment Map können auch rauer wirkende Reflexionen (mit unscharfen Glanzlichttexturen) und diffuse Oberflächen (mit Irradiance Maps) erreicht werden. Theoretisch sollten unterschiedliche Teile der Textur unterschiedlich weichgezeichnet werden. Glücklicherweise lässt sich das menschliche Auge leicht täuschen, und der generelle Eindruck einer Reflexion ist wichtiger als die korrekte Reflexion selbst.

Abbildung 2.13: Das schwarze Rechteck verändert seine Form je nach Position in der Sphere Map.

Grundsätzlich kann sowohl für den Filtervorgang für unscharfe Reflexionen und Irradiance Maps eine verkleinerte Version der originalen Textur verwendet werden, um Rechenzeit zu sparen. Gerade bei Irradiance Maps muss ein Filter über die Hälfte der Map durchgeführt werden, was sehr aufwändig ist und durch eine kleinere Textur stark beschleunigt werden

kann. Die dabei entstehenden Fehler sind durch die starke Weichzeichnung normalerweise nicht sichtbar.

Die einfachste Art, unklare Reflexionen zu erreichen, ist, die Environment Map einheitlich mit einem 2D-Blurfilter weichzuzeichnen. Das ist jedoch nicht ausreichend, da ein Texel am Rand der Map Information über einen anders geformten Teil der Umgebung enthält als ein Texel in der Mitte der Map. Am Beispiel einer Sphere Map (Abb. 2.13) kann man erkennen, dass das schwarze Rechteck A, das in der Mitte der Map sichtbar ist, anders geformt ist als das schwarze Rechteck B am Rand der Map. Rechteck A ist größtenteils proportional, Rechteck B ist entlang der Kontur der Kugel gestreckt und entlang des Radius der Kugel gestaucht. Beide Rechtecke haben in der realen Welt die gleiche Entfernung zur Kugel und sehen die Kugel direkt an, nehmen also von der Kugel aus gesehen einen gleich großen und gleich geformten Bereich der Umgebung ein.



(a) (b)

Abbildung 2.14: Auswirkung des Filterkernels an unterschiedlichen Stellen der Map. In (a) ist die originale Sphere Map, in (b) die gefilterte Version zu sehen.

Daher sollte für einen korrekter Weichzeichner der Filterkernel für jeden Texel denselben Bereich der Umgebung umfassen. Je nach Position in der Environment Map wird die zweidimensionale Form des Kernels also anders aussehen, wie in Abb. 2.14 zu sehen. In [Lan05] wird eine Methode vorgeschlagen, um eine Cubemap weichzuzeichnen, die auf diese Probleme Rücksicht nimmt. Dabei wird ein zylindrischer Filter in 3 Achsen durchgeführt. In Tabelle 2.15 wird die Vorgangsweise an Hand einer Sphere Map dargestellt. Der zylindrische Filter wird erreicht, indem eine Kugel gerendert wird, deren Durchmesser genau der Bildbreite und -höhe entspricht. Die vollkommen reflektive Sphere Map wir darauf aufgebracht. Die Texturkoordinaten werden so gewählt, dass das Bild der Kugel genau auf dem Kugelobjekt zu sehen ist. Die Kugel wird Schritt für Schritt um die x-Achse gedreht. Jeder Schritt wird gerendert und alle resultierenden Bilder gemit-

telt. Das resultierende gefilterte Bild (siehe Tabelle 2.15(a)) wird jetzt auf das Kugelobjekt aufgebracht. Danach wird die Kugel um die y-Achse gedreht und die gerenderten Bilder gemittelt. Das Ergebnisbild (jetzt sowohl in x- als auch in y-Achse gefiltert) wird wieder auf das Kugelobjekt aufgebracht und die Kugel wird um die z-Achse gedreht. In Tabelle 2.15(d) ist das endgültige Ergebnisbild zu sehen.

(a) (b) (c) (d)

Abbildung 2.15: Korrektes Weichzeichnen einer Sphere Map. Bild (a) zeigt die Auswirkung eines radialen Weichzeichners in die x-Achse, Bild (b) zeigt den Effekt desselben Filters in die z-Achse, Bild (c) in die y-Achse und Bild (d) zeigt das resultierende weichgezeichnete Bild.

2.4.2 Erstellen von Cube Maps

Das aus dem Videobild ausgeschnittene Bild der Kamera eignet sich hervorragend als Sphere Map. Sphere Maps haben jedoch einige Nachteile (siehe [AMH02, Kap. 5.7.4]). Der gravierenste Nachteil ist, dass sie betrachterabhängig sind. Das bedeutet, dass sie nur korrekte Ergebnisse liefern, wenn die Kamera, die die Sphere Map aufgenommen hat, und die Kamera, die mit der Sphere Map rendert, in diesselbe Richtung blicken. Diese Tatsache macht Sphere Maps unpraktisch für das Zwei-Kamera Setup, da die Kamera für die Spiegelkugel (oder das Fischauge) fix, die Kamera für die Szene aber beweglich ist. Aus diesem Grund werden für das Rendering im Zwei-Kamera Setup Cubemaps verwendet. Da das ursprüngliche Bild der Umgebung (das Video der Spiegelkugel) aber als Sphere Map vorliegt, ist eine Umwandlung von Sphere Maps in Cube Maps notwendig.

Wie in Abb. 2.16 sichtbar, gibt es für die sechs Seiten in der Cube Map entsprechende Bereiche in der Sphere Map (in der Grafik durch gleiche Farben dargestellt). Jeder Texel in einer Environment Map entspricht einem dreidimensionalen Vektor in die Umgebung (in Abb. 2.17 sind die Vektoren farbkodiert dargestellt). Um eine Cube Map aus einer Sphere Map zu generieren, kann für jeden Texel der Cube Map der entsprechende Vektor berechnet werden. Danach wird die Stelle berechnet, die in der Sphere Map demselben Vektor entspricht. Der dort eingetragene Farbwert wird in die Cube Map eingetragen (siehe Abb. 2.18).

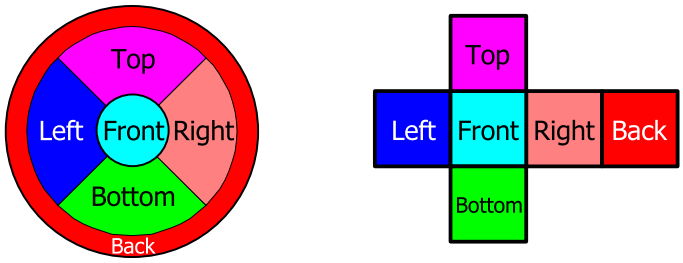


Abbildung 2.16: Die 6 Seiten der Cube Map können in der Sphere map ermittelt werden.

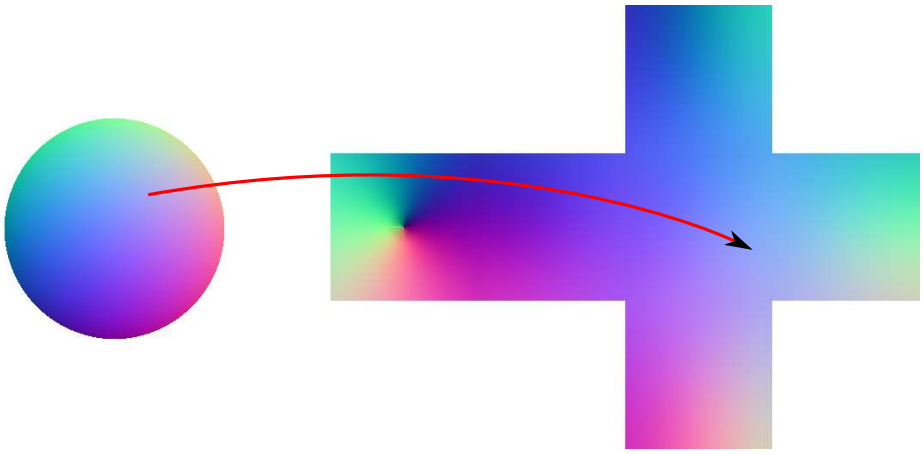


Abbildung 2.17: In Cube Maps und Sphere Maps entspricht jeder Punkt einem Vektor in die Umgebung. Jeder Vektor, der in der Sphere Map zu finden ist, kann auch für die Cube Map ermittelt werden.

Für genauere Details zur Berechnung von Vektoren aus Texeln einer Cube Map siehe [AMH02, Kap. 5.7.4], oder [Len04, Kap. 6.4.3]. Zur Berechnung von Sphere Map Koordinaten aus Richtungsvektoren wird zuerst der Augvektor $\vec{E}$ bestimmt. Da das Auge der Ursprung des Augkoordinatensystems ist und die Sphere Map theorethisch in unendlicher Entfernung in Richtung der negativen z-Achse liegt, entspricht $\vec{E}$ dem Vektor (0/0/1). Danach wird der Halbweg-Vektor $\vec{H}(x_H, y_H, z_H)$ zwischen dem Augvektor $\vec{A}(0.0/0.0/1.0)$ und dem Richtungsvektor $\vec{R}$ mit der Formel

$$\vec{H}(x_H, y_H, z_H) = \frac{\vec{A} + \vec{R}}{\|\vec{A} + \vec{R}\|} \tag{2.3}$$

berechnet. Der 2D-Vektor $\vec{H}(x_H, y_H)$ wird als Sphere Map Texturkoordinaten verwendet. Genauere Information zur Berechnung von Sphere Map Koordinaten aus Vektoren siehe Kapitel 2.5.1 dieser Arbeit oder [AMH02,

Abbildung 2.18: Farbwerte werden von der Sphere Map in die korrespondierende Stelle in der Cube Map geschrieben.

Kap. 5.7.4].

Sphere Maps aus unterschiedlichen Blickwinkeln Die Sphere Map kann aus jeder beliebigen Richtung aufgenommen werden, sie befindet sich also nicht zwingend im Weltkoordinatensystem (Das wäre der Fall, wenn die Mitte der Sphere Map genau der positiven y-Achse der Welt entspricht, also „oben“ darstellt). Daher wird eine Transformationsmatrix **M** übergeben, die die Transformation vom Weltkoordinatensystem (dem Koordiantensystem der zukünftigen Cube Map) ins Koordinatensystem der Sphere Map beschreibt (siehe Abb. 2.19). Mit dieser Matrix wird der Vektor $\vec{V}$, der für das Auslesen aus der Sphere Map verwendet werden, multipliziert:

$$\vec{V}_{Sphere} = \vec{V}_{Cube} \cdot \mathbf{M}. \tag{2.4}$$

Der resultierende Vektor in Sphere Map Koordinaten wird danach wie in Gleichung 2.3 zur Berechnung der Texturkoordinaten für den Zugriff in die Sphere Map verwendet.

Sphere Maps mit unterschiedlichem Sichtfeld Eine perfekte Sphere Map hat ein Sichtfeld von 360°. Das bedeutet, dass jeder Pixel am Rand der Map dem Vektor $(0.0/0.0/-1.0)$ entspricht und daher diesselbe Farbe hat. In der Realität hat eine Sphere Map, die von einer Spiegelkugel augenommen wurde, ein Sichtfeld von ca. 300°, eine Map, die mit einem Fischaugenobjektiv aufgenommen wurde, nur maximal 180° (siehe Abb. 2.20).

Um diesem Umstand gerecht zu werden, kann die Berechnung der den Cube Map Texeln entsprechenden Vektoren für die Sphere Map angepasst werden. Das hat den Effekt, dass nicht eine ganze Sphere Map auf eine

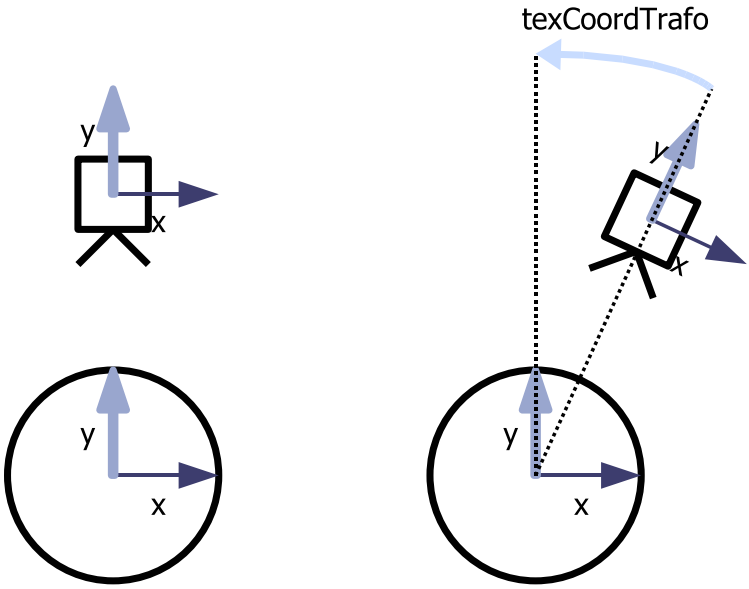


Abbildung 2.19: Transformation vom Raum der Sphere Map in Weltkoordinaten.

ganze Cube Map abgebildet wird, sondern dass ein gewisser Bereich im hinteren Teil der Cube Map nicht gefüllt wird. Die Texel in diesem Bereich können mit interpolierten Werten aus den nähesten bekannten Bereichen gefüllt werden. Um das zu erreichen wird die Berechnung der Sphere Map Texturkoordinaten aus dem Richtungsvektor verändert. Der Augvektor $\vec{A}$, der normalerweise als (0.0/0.0/1.0) angenommen wird, wird verkürzt. Wie in Abb. 2.21(a) zu sehen ist, verstärkt sich dadurch der Einfluss des Reflexionsvektors $\vec{R}$ auf den Halbwegvektor $\vec{H}$. Dies hat zur Folge, dass auf jeden Texel der Cubemap Texel der Spheremap gezeichnet werden, die einem Vektor entsprechen der weiter nach „hinten“ zeigt, also näher am Rand der Sphere Map liegt. Für Vektoren, deren Texel in der Sphere Map ausserhalb des Randes (der Sphere Map) liegen würden, wir der näheste Texel innerhalb der Map verwendet.

2.5 Rendering

Das eigentliche Rendering teilt sich auf in Beleuchtung und Beschattung. Die Beleuchtung mit IBL ist relativ einfach, weil die Formeln von Lambert und Blinn/Phong durch einfache Texturzugriffe ersetzt werden. Die Beschattung ist komplizierter, da keine eindeutigen Positionen für Lichtquellen bekannt ist und die einzige Information über Lichtquellen die Environment Map ist. Im Kapitel 2.5.2 werden zwei Ansätze vorgestellt, um mit dieser Situation umzugehen.

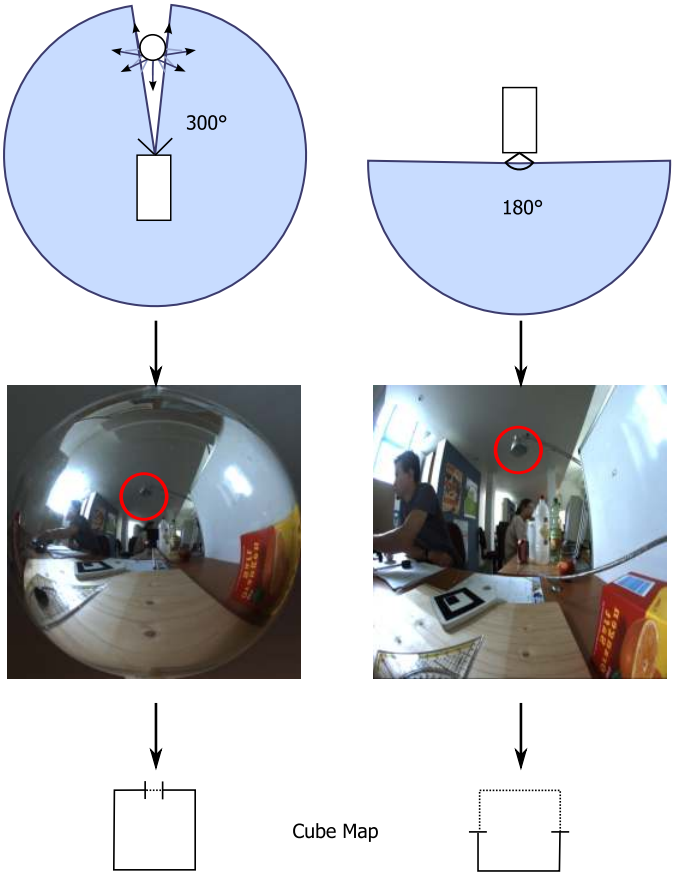
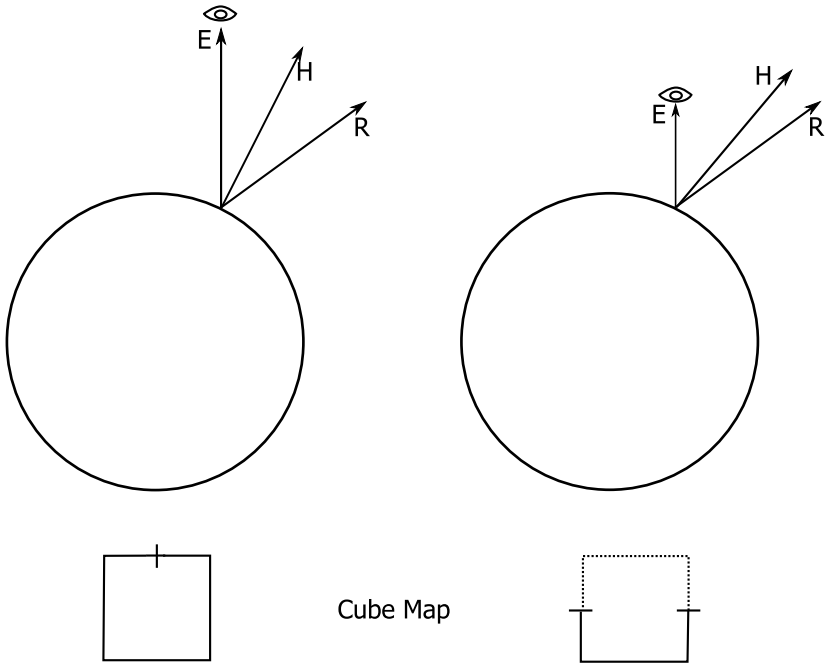


Abbildung 2.20: Sichtwinkel von Spiegelkugel und Fischaugenobjektiv. Im Bild der Spiegelkugel links ist ein wesentlich größerer Teil der Getränkepackung (rechts in der Sphere Map) zu sehen als im Bild des Fischaugenobjektivs rechts. Man beachte auch die Position der Schreibtischlampe.

2.5.1 Beleuchtung

Für die Beleuchtung mit IBL werden die Formeln von Lambert und Blinn durch Texturzugriffe in die Irradiance Map und Glanzlichttextur ersetzt. Für diffuse Beleuchtung wird aus der Environment Map an der Stelle der Oberflächennormale $\vec{N}$ ausgelesen, für Glanzlichtbeleuchtung an der Stelle der reflektierten Augvektors $\vec{R}$. Der Augvektor $\vec{E}$ ist der Vektor vom aktuell gerenderten Punkt zum Auge. Um mit Environment Maps beleuchten zu können, muss die Oberflächennormale $\vec{N}$ bzw. der reflektierte Augvektor $\vec{R}$ in Texturkoordinaten auf der Map umgewandelt werden.



(a)

Abbildung 2.21: Berechnung für Environment Maps mit unterschiedlichem Sichtfeld. Je kürzer der Augvektor E, desot kleiner ist der von der Sphere Map abgebildete Bereich.

Beleuchtung mit Sphere Maps

Da eine Sphere Map üblicherweise in Augkoordinaten vorliegt, müssen die Vektoren $\vec{N}$ und $\vec{R}$ ebenfalls in Augkoordinaten vorliegen. Um einen Normalvektor aus dem Objektkoordinatensystem ins Auggkoordinatensystem zu bringen, wird er mit der invers-transponierten Modelview Matrix $(\mathbf{MV})^{-\mathbf{T}}$ multipliziert (siehe [Len04, Kap. 3.5] für eine genaue Beschreibung des Grundes dafür).

$$\vec{N}_{Eye} = \vec{N}_{Object} \cdot (\mathbf{MV})^{-\mathbf{T}} \tag{2.5}$$

Um die Position eines Verticis $\vec{P}$ von Objektkoordinaten in Augkoordinaten zu bringen, muss er mit der Modelview Matrix $\mathbf{MV}$ multipliziert werden.

$$\vec{P}_{Eye} = \vec{P}_{Object} \cdot \mathbf{MV} \tag{2.6}$$

Glanzlichter Die Vorgangsweise, um Texturkoordinaten für die Glanzlichtfarbe zu finden, wird von Möller und Haines in [AMH02, Kap. 5.3.4] beschrieben. Hier wird zuerst der Augvektor $\vec{A}_{Eye}$ von der Position des Verticis $\vec{P}_{Eye}$ zur Position der Kamera $\vec{C}_{Aug}$ berechnet, indem $\vec{P}_{Eye}$ von $\vec{C}_{Aug}$

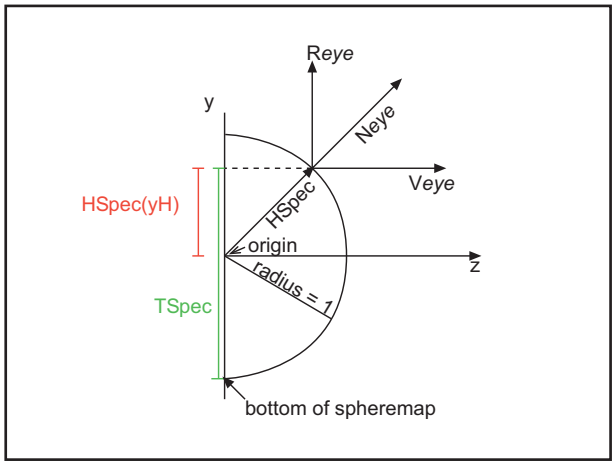


Abbildung 2.22: Die Berechnung der Texturkoordinaten $\vec{T}G$ für Glanzlichter. Es ist zu sehen, dass der Vektor $\vec{H}G$ ident ist mit dem Vektor $\vec{N}_{Eye}$. Der Vektor $\vec{T}G$ entspricht $\vec{H}G$ nach der Umwandlung in den Texturwertebereich zwischen 0.0 und 1.0.

subtrahiert wird.

$$\vec{A}_{Eye} = \vec{C}_{Eye} - \vec{P}_{Eye} \tag{2.7}$$

Dann wird der Reflexionsvektor $\vec{R}_{Eye}$ des Augvektors $\vec{V}_{Eye}$ an der Normale $\vec{N}_{Eye}$ berechnet. Aus $\vec{R}_{Eye}$ können die Texturkoodinaten auf der Sphere Map bestimmt werden. Dafür wird der Halbweg-Vektor $\vec{H}G(x, y, z)$ zwischen dem Augvektor $\vec{V}_{Eye}$ und dem Reflexionsvektor $\vec{R}_{Aug}$ mit der Formel

$$\vec{H}\mathrm{G}(x, y, z) = \frac{\vec{A}_{Eye} + \vec{R}_{Eye}}{\|\vec{A}_{Eye} + \vec{R}_{Eye}\|} \tag{2.8}$$

berechnet. Der 2D-Vektor $\vec{H}(x, y)$ wird als Sphere Map Texturkoordinaten verwendet. Wie in Abb. 2.22 zu sehen ist, ist $\vec{H}\mathrm{G}$ aber ident mit $\vec{N}_{Eye}$. Daher ist die Berechnung des Reflexionsvektors und anschließend des Halbweg-Vektors redundant.

Diffuse Beleuchtung Die Berechnung der diffusen Texturkoordinaten erfolgt analog zu den Glanzlichtkoordinaten. Der einzige Unterschied ist, dass die Basis für die Texturkoordinatenberechnung nicht die Reflexion des Augvektors $\vec{R}_{Eye}$, sondern die Normale $\vec{N}_{Eye}$ ist. Daher wird der normalisierte Halbweg-Vektor zwischen $\vec{N}_{Eye}$ und $\vec{V}_{Eye}$,

$$\vec{H}D(x, y, z) = \frac{\vec{N}_{Eye} + \vec{V}_{Eye}}{\|\vec{N}_{Eye} + \vec{V}_{Eye}\|} \tag{2.9}$$

berechnet und für den Texturzugriff verwendet (siehe Abb. 2.23).

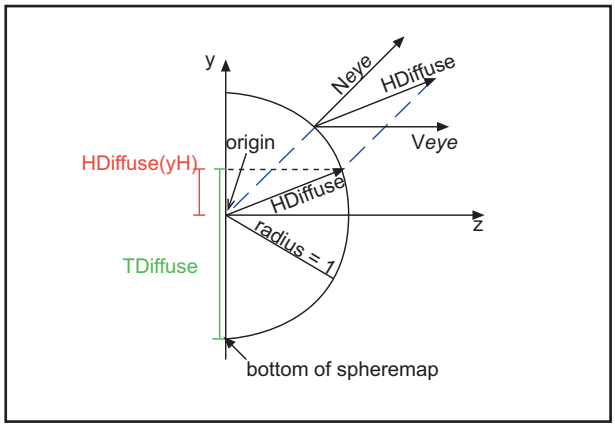


Abbildung 2.23: Die Berechnung der Texturkoordinaten $\vec{T}_{\text{Diff}}$ für diffuse Beleuchtung. Der Vektor $\vec{T}_{\text{Diff}}$ entspricht $\vec{H}_{\text{Diff}}$ nach der Umwandlung in den Texturwertebereich zwischen 0.0 und 1.0. Der Vektor $\vec{H}$Diff ist der Halbweg-Vektor zwischen dem NormalVektor $\vec{N}_{Eye}$ und dem Augvektor $\vec{V}_{Eye}$.

Umrechnung in Texturkoordinaten Die oben beschriebenen Berechnungen ergeben Koordinaten im Bereich -1.0 bis 1.0. Texturkoordinaten werden üblicherweise aber im Bereich 0.0 bis 1.0 dargestellt. Daher ist es notwendig, die Werte von $\vec{H}S$ und $\vec{H}D$ in den korrekten Bereich zu bringen. Das geschieht durch eine Multiplikation mit 0.5 und anschliessende Addition um 0.5. Diese Operation erzeugt zwei neue Vektoren, genannt $\vec{T}S$ und $\vec{T}D$ (siehe Abbildungen 2.22 und 2.23 sowie Gleichungen 2.10 und 2.11).

$$\vec{T}S(x,y) = \vec{H}S(x,y) \cdot 0.5 + 0.5 \tag{2.10}$$

$$\vec{T}D(x,y) = \vec{H}D(x,y) \cdot 0.5 + 0.5 \tag{2.11}$$

Beleuchtung mit Cube Maps

Im Gegensatz zu Sphere Maps liegen Cube Maps üblicherweise im Weltkoordinatensystem vor. Die Oberflächennormale $\vec{N}$ bzw. der reflektierte Augvektor $\vec{R}$ müssen daher auch im Weltkoordinatensystem vorhanden werden. Um einen Normalvektor vom Objektkoordinatensystem ins Weltkoordinatensystem zu bringen, muss er mit der invers transponierten Model Matrix $(\mathbf{M})^{-\mathbf{T}}$ multipliziert werden (siehe [Len04, Kap. 3.5] für eine genaue Beschreibung des Grundes dafür).

Die Berechnung von Texturkoordinaten auf der Cube Map aus einem 3-D Vektor in die Umgebung wird wie folgt durchgeführt. Die Koordinate mit dem größten Wert bestimmt, auf welchem Face der Cube Map sich das gesuchte Texel befindet. Die Texturkoordinaten auf diesem Cube Map Face selbst werden berechnet, indem die kleineren zwei Koordinaten durch den

Absolutwert der größten Koordinate dividiert werden. Sie befinden sich danach im Bereich von -1.0 bis 1.0 und müssen auf den Bereich von 0.0 bis 1.0 abgebildet werden, um als Texturkoordinaten verwendet werden zu können. Wenn also z. B. die Texturkoordinaten für den Vektor $\vec{V}(-3.2, 5.1, -8.4))$ gesucht sind, wird zuerst mit $\max(x, y, z)$ bestimmt, dass das gesuchte Texel auf dem Cube Map Face $-Z$ (*hinten*) liegt.

$$\text{face} = \max(\vec{V}_x, \vec{V}_y, \vec{V}_z) \tag{2.12}$$

Die beiden übrigen Koordinaten ($\vec{V}_x$ und $\vec{V}_y$ werden durch $\vec{V}_z$ dividiert und auf auf den Texturkoordiatenbereich abgebildet um die Texturkoordinaten $\vec{T}_{(u,v)}$ zu erhalten:

$$\vec{T}_u = \frac{\vec{V}_x}{\vec{V}_z} \cdot 2.0 + 1.0 \tag{2.13}$$

$$\vec{T}_v = \frac{\vec{V}_y}{\vec{V}_z} \cdot 2.0 + 1.0 \tag{2.14}$$

Die Beschreibung der Vorgangsweise sowie das Beispiel sind aus Möller und Haines [AMH02, Kap. 5.7.4] entnommen. Eine detailliertere Beschreibung kann u.a. in Lengyel [Len04, Kap. 6.4.3] gefunden werden.

Glanzlichter Zum Auslesen der Glanzlichtfarbe ist der reflektierte Augvektor in Weltkoordinaten $\vec{R}_{World}$ notwendig (siehe Abb. 2.22). Da die Normale $\vec{N}$ und die Position des aktuellen Verticis $\vec{P}$ zu Beginn nur in Objektkoordinaten bekannt ist, müssen sie in Weltkoordinaten gebracht werden. Dazu wird $\vec{N}$ mit der invers transponierten Modelmatrix $(\mathbf{M})^{-\mathbf{T}}$, $\vec{P}$ mit der Modelmatrix $\mathbf{M}$ multipliziert,

$$\vec{N}_{World} = \vec{N}_{Object} \cdot (\mathbf{M})^{-\mathbf{T}} \tag{2.15}$$

$$\vec{P}_{World} = \vec{P}_{Object} \cdot \mathbf{M} \tag{2.16}$$

Zur Berechnung des Augvektors $\vec{A}$ wird neben $\vec{N}$ und $\vec{P}$ die Position der Kamera $\vec{C}$ benötigt. $\vec{C}$ ist der Ursprung des Kamerakoordinatensystems. Um die Position der Kamera in Weltkoordinaten $\vec{C}_{World}$ zu erhalten, muss $\vec{C}_{Eye}$ mit der inversen Modelmatrix $(\mathbf{M})^{-\mathbf{1}}$ transformiert werden:

$$\vec{C}_{World} = \vec{C}_{Eye}(0, 0, 0) \cdot (\mathbf{M})^{-\mathbf{1}} \tag{2.17}$$

Danach kann der Augvektor $\vec{E}_{World}$ berechnet werden, indem $\vec{P}_{World}$ von $\vec{C}_{World}$ subtrahiert wird:

$$\vec{E}_{World} = \vec{C}_{World} - \vec{P}_{World} \tag{2.18}$$

Um den reflektierten Augvektor $\vec{R}$ zu bekommen, wird $\vec{E}$ an $\vec{N}$ mit der Formel

$$\vec{R} = 2 \cdot (\vec{N} \bullet \vec{A}) \cdot \vec{N} - \vec{A} \tag{2.19}$$

reflektiert, wie von Möller und Haines in [AMH02, Kap. 4.3.2] und Lengyel in [Len04, Kap. 5.4.1] beschrieben.

Diffuslichter Zum Auslesen der Diffusfarbe ist die Oberflächennormale in Weltkoordinaten $\vec{N}_{World}$ notwendig. Normalerweise ist die Normale in Objektkoordinaten $\vec{N}_{Objext}$ gegeben. Durch eine Multiplikation mit der invers transponierten Model Matrix $(\mathbf{M})^{-\mathbf{T}}$ kann sie in Weltkoordinaten gebracht werden.

$$\vec{N}_{World} = \vec{N}_{Object} \cdot (\mathbf{M})^{-\mathbf{T}} \tag{2.20}$$

2.5.2 Beschattung

Wenn mit Image Based Lighting beleuchtet wird, stellt die Beschattung ein Problem dar, da die einzige Information über die Umgebung die Environment Map ist. Sämtliche Techniken zur Berechnung von Schatten benötigen jedoch eine eindeutige Position einer virtuellen Lichtquelle. Eine Möglichkeit, mit diesem Problem umzugehen, ist, die Positionen realer Lichtquellen aus der Environment Map durch Bildanalyse zu extrahieren. Eine andere Möglichkeit ist, die Environment Map ohne Wissen über tatsächliche Positionen der realen Lichtquellen zu samplen und mit einer großen Anzahl an fixen Lichtquellen zu beschatten.

Keine Lichtquellenextrahierung

Der einfachste Version, Schatten darzustellen, verwendet eine einzelne Lichtquelle, die Schatten mittels Shadow Maps wirft. Die Position dieser Lichtquelle muss manuell eingestellt werden. Der harte Rand des Schatten kann mit Techniken wie *Percentage Closer Filtering* [RSC87] weichgezeichnet werden.

Diese Technik ist zu überlegen, wenn die Beleuchtungssituation im Voraus bekannt ist und sich während der Laufzeit nicht ändert. Ein Vorteil ist, dass keine Artefakte wie Flackern der Lichtposition entstehen können und Attribute wie Härte und Tiefe des Schattens sehr genau eingestellt werden können, ohne sich auf automatische (und möglicherweise nicht robuste) Verfahren verlassen zu müssen. Ausserdem fällt kein zusätzlicher Rechenaufwand für Lichtquellenextrahierung an.

Lichtquellenextrahierung durch Bildanalyse

Die erste Möglichkeit zur Lösung des Schattenwurfproblems ist das Extrahieren von möglichen Lichtquellen aus der Environment Map. Dazu müssen helle Bereiche in der Environment Map gefunden werden, die möglicherweise Lichtquellen darstellen. Die einfachste Methode dazu ist, alle Texel

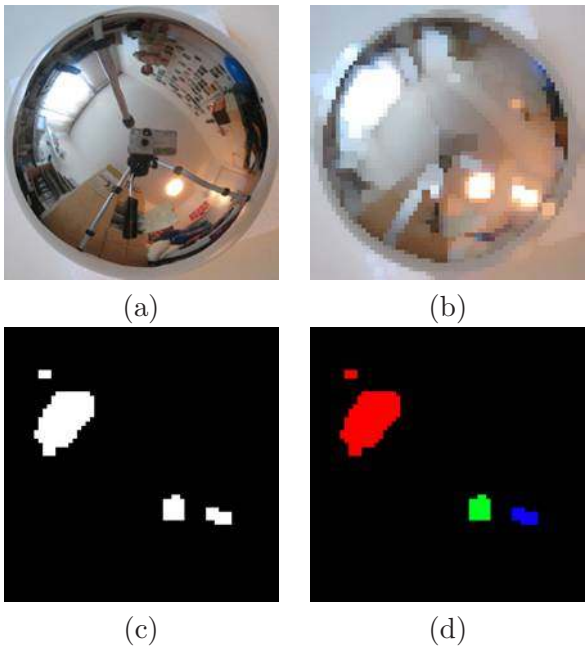


Tabelle 2.1: Finden von sinnvollen Lichtpositionen. In Bild (a) ist die originale Sphere Map zu sehen. Diese wird mit einem Maximum Filter verkleinert (b). Bild (c) zeigt alle potentiellen Lichtquellen, und Bild (d) zeigt diese zu 3 Gruppen zusammengefasst.

der Environment Map, die über einem bestimmten Schwellwert liegen, als potentielle Lichtquellen anzunehmen.

Da u.U. die Anzahl an potentiellen Lichtquellen viel zu groß ist bzw. viele von ihnen direkt nebeneinander liegen, bietet es sich an, mehrere Lichtquellen zu einer zusammenzufassen. Entweder werden zusammenhängende Regionen heller Pixel mittels Region Labelling (siehe [BB05]), oder Texel in geringer Entferung zueinander mit Clustering Algorithmen zusammengefasst.

Ein Problem der Lichtquellenextrahierung mit Bildanalyse ist die Tatsache, dass CCD-Chips selbst guter Kameras nur einen geringen Kontrastumfang abbilden können[1]. Das bedeutet, dass, wenn ein Punkt im Bild als reines Schwarz dargestellt wird, ein Pixel, der z. B. 16 mal so hell ist, bereits als reines Weiss dargestellt wird. Alles, was heller oder dunkler ist, geht verloren. Die Helligkeitsunterschiede in realen Szenen sind aber beträchtlich höher. Das führt dazu, dass eine große Menge an Pixeln denselben Farbwert

[1]Die Behauptungen, welcher Kontrastumfang von Digitalkameras dargestellt werden kann, unterscheiden sich sehr stark. In einem Eintrag in Wikipedia (http://de.wikipedia.org/wiki/Kontrast) wird z. B. von einem maximalen Kontrastumfang von 1:40 gesprochen. Ein Test mit der in dieser Arbeit verwendeten Firewire-Kamera ergab einen Kontrastumfang von 1:16 (Vier Blendenstufen zwischen einem rein weissen und einem rein schwarzen Pixel)

(reines Weiß) im Videobild haben, obwohl sich die reale Lichtemission der Objekte stark unterscheidet. Man stelle sich eine Szene vor, in der die Sonne sichtbar ist und eine weisse Wand beleuchtet. Höchstwahrscheinlich werden sowohl die Sonne als auch die Wand im Videobild weiss erscheinen, obwohl die Sonne hunderfach heller ist als die Wand und einen viel deutlicheren Schatten verursacht.

Große Anzahl an fixen Lichtquellen

Die zweite Möglichkeit der Schattenbewältigung ist die Verwendung vieler fixer Lichtquellen unterschiedlicher Intensität. Dafür wird in einer Halbkugel über der Szene eine große Menge an Lichtquellen (eigentlich Schattenquellen, weil die Beleuchtung nach wie vor über IBL funktioniert) platziert. Jede dieser Schattenquellen berechnet die von ihrer Position aus sichtbaren (und damit von ihr beleuchteten) Objekte mit Shadow Maps.

Shadow Maps verwenden eine Tiefentextur, um die Entfernung der von der Lichtquelle aus sichtbaren Objekte zu kodieren. Beim Rendering wird diese Textur auf alle Objekte projeziert und mit der tatsächlichen Entfernung zur Lichtquelle verglichen. Ist die tatsächliche Entfernung zur Lichtquelle größer als der in der Textur gespeicherte Wert, liegt eine anderes Objekt zwischen dem gerade gerenderten Punkt und der Lichtquelle und das Objekt befindet sich dadurch im Schatten.

Da aus der Sicht jeder Lichtquelle eine Tiefentextur erzeugt werden muss und auch die mathematischen Operationen beim Projezieren von Texturen auf Objekte praktisch ident sind mit dem Projezieren von Objekten auf den Bildschirm, ähnelt eine Lichtquelle für Shadow Mapping einer Kamera. Jede Lichtquelle benötigt daher eine Transformations- und eine Projektionsmatrix. Diese Matritzen müssen daher z. B. beim Programmstart berechnet werden. Weiters benötigt jede Lichtquelle eine Tiefentextur, die während der Laufzeit erneuert werden muss, damit Bewegungen der Objekte auch in den Schatten sichtbar werden.

Da das Erstellen von aktualisierten Tiefentexturen für jede Lichtquelle in jedem Frame sehr aufwändig ist (es muss jedes Mal die gesamte Szene gerendert werden), kann optimiert werden, indem pro Frame nur die Textur einer einzigen Lichtquelle erneuert wird. Die Auswirkung davon ist, dass in dynamischen Szenen die Schattentexturen je nach Framerate und Anzahl der Lichtquellen bis zu einer Sekunde benötigen, bis alle Schatten wieder von der aktuellen Szene geworfen werden. Dadurch entsteht der Eindruck, dass die Schatten hinter den beweglichen Objekten „herlaufen“.

Beim Rendering wird für jeden Punkt der Schatten von jeder Lichtquelle überprüft. Die Stärke der Schatten wird akkumuliert. Das bedeutet, dass die Helligkeit jedes Punktes abhängig ist von der Menge an Lichtquellen, von denen aus er sichtbar ist. Ein Punkt, der aus viele Richtungen Licht erhält, ist wesentlich heller beleuchtet, als ein Punkt der nur aus wenigen Rich-

tungen Licht erhält. Dadurch entsteht ein Effekt, der Ambient Occlusion (siehe [Hay02]) ähnlich sieht.

Unterschiedliche Intensitäten Die Farbe und Intensität jedes Schattens soll jedoch nicht gleich sein. Vielmehr soll sie abhängig sein von der Farbe des Bereichs der Environment Map, der in derselben Richtung liegt wie die Lichtquelle. Dadurch werfen Lichtquellen, die vor einem hellen Bereich stehen, einen deutlich sichtbaren Schatten, Lichtquellen, die vor einem dunklen Bereich stehen jedoch einen kaum sichtbaren Schatten. Die Lichtbedingungen der Szene wirken sich so unmittelbar auf die Schatten aus, unabhängig davon, ob sie von einer starken Lichtquelle, einer einfärbigen Wand oder von mehreren kleinen Lichtquellen herrühren.

Um diese Intensitäten berechnen zu können, muss die Environment Map in Bereiche aufgeteilt werden, die jeweils die Farbe und Intensität einer Lichtquelle bestimmen. Idealerweise sollte die Environment Map so aufgeteilt werden, dass jeder Pixel auf jene Lichtquelle am meisten Einfluss hat, der er am nächsten liegt. Eine einfachere Möglichkeit ist, eine verkleinerte Version der Environment Map (z. B. ein niedrigeres Mip-Map Level) zu nehmen und jedem Texel der Map eine Lichtquelle zuzuordnen. Die Helligkeit

$$H = T \cdot \vec{N} \bullet \vec{V}, \tag{2.21}$$

die ein Punkt P auf der Oberfläche eines Objektes von einer Lichtquelle empfängt, ist der ausgelesene Wert T aus der Environment Map an der Stelle der Lichtquelle, multipliziert mit dem Kosinus des Winkels α zwischen der Oberflächennormale $\vec{N}$ von P und dem Vektor $\vec{V}$ in Richtung der Lichtquelle. Danach kann für P die diffuse Beleuchtung

$$D = \sum H_{vis} \tag{2.22}$$

berechnet werden als die Summe der Helligkeit aller Lichter, die von P aus sichtbar sind ($\sum H_{vis}$). Alternativ dazu kann D berechnet werden als

$$D = D_{Irrad} - \sum H_{occ}, \tag{2.23}$$

wobei $\sum H_{occ}$ die Helligkeit der Lichter, die verdeckt sind, und D_{Irrad} die mit Irradiance Maps berechnete Diffusfarbe ist.

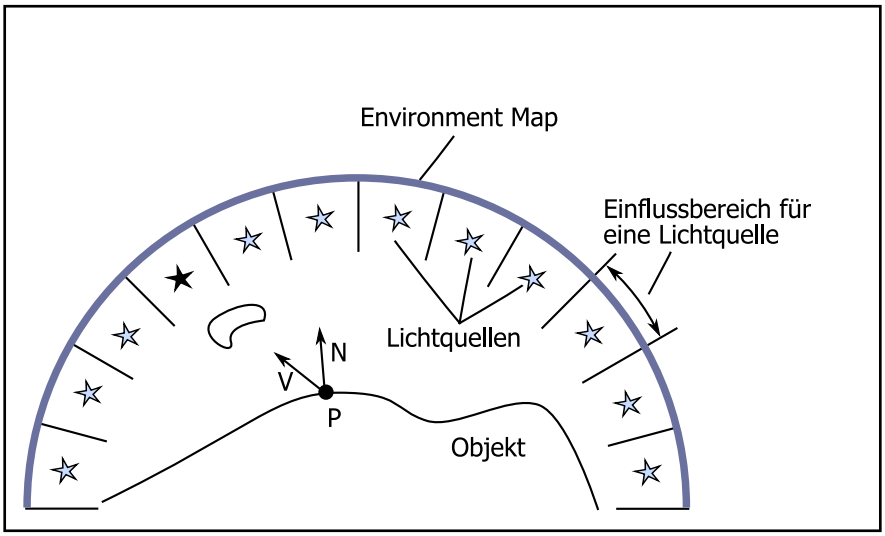


Abbildung 2.24: Eine große Anzahl an schattenwerfenden Lichtern ist gleichmäßig über die Umgebung verteilt. Die Lichtintensität jedes Punktes P auf der Oberfläche eines Objektes ist die Summe aller von P aus sichtbaren Lichtquellen.

Kapitel 3

Implementierung

3.1 Überblick über die Programmstruktur

Die im folgenden Kapitel vorgestellte Applikation wurde in C++ entwickelt. Als Grafik-API wurde *OpenGL* verwendet, Shader wurden in *Cg* implementiert.

In Tabelle 3.1 ist eine Auflistung der verwendeten Klassen mit kurzer Beschreibung ihrer Aufgabe zu sehen. Die konkrete Aufgabe der wichtigsten Klassen wird danach genauer beschrieben.

Klasse	**Aufgabe**
Datenhaltung und Steuerung	
`Scene`	Beinhaltet alle globalen Daten der Szene.
`SceneLoader`	Lädt ein `Scene` Objekt aus einer Textdatei.
`LightsManager`	Lädt eine Anzahl an Lichtern aus einer Textdatei und speichert sie in eine `Scene`.
`ARWrapper`	Abstrahiert die Funktionalität von Tracking-Libraries wie ARToolkit.
`Light`	Repräsentation einer Lichtquelle für Shadow Mapping, d.h. mit einer Transformations- und einer Projektionsmatrix sowie einer Tiefentextur.
`Camera`	Repräsentation einer Kamera.
`Renderer`	Rendert die Objekte eines Renderdurchgangs mit ihren Effekten.
`RendererLoader`	Lädt ein `Renderer` Objekt aus einer Textdatei.
Fortsetzung auf nächster Seite...	

Klasse	Aufgabe
`RenderParameters`	Sammlung aller Statusdaten wie z. B. aktuelle Transformationsmatrizen, Texturen, usw. Wird zur Übergabe von Daten vom `Renderer` an ein `RenderEffectInterface` Objekt verwendet.
`RenderEffectInterface`	Schreibt eine Schnittstelle für Effekte vor, die vom `Renderer` verwendet wird.
Hilfsklasse	
`Helpers`	Sammlung von praktischen Hilfsmethoden wie das Zeichnen eines bildschirmfüllenden Rechtecks, Testausgabe von Cube Maps usw.
Beleuchtungseffekte	
`CubeMapLightingEffect`	Beleuchtet Objekte mit Environment Cube Maps (Diffus- und Glanzlichter).
`CubeMapLighting EffectShadow`	Beleuchtet Objekte mit Environment Cube Maps (Diffus- und Glanzlichter) und beschattet sie mit einer vorberechneten Schattentextur.
`ManyShadowsEffect`	Zeichnet Objekte von acht verschiedenen Lichtquellen beschattet.
`DepthOnlyEffect`	Zeichnet nur in den Tiefenbuffer, keine Farbwerte werden gezeichnet. Wird für das Rendering in die Tiefentexturen für Shadow Mapping verwendet.
`ShadowOnlyEffect`	Zeichnet Objekte nur mit Schatten aus einer vorberechneten Schattentextur, der Rest bleibt transparent. Wird für Phantomobjekte verwendet, d. h. für Objekte, die die reale Umgebung repräsentieren und nur Schatten empfangen sollen.
Textureffekte	
`SphereToCubeMap`	Konvertiert eine Sphere Map in eine Cube Map.
`BlurEffect`	Führt einen Unschärfefilter auf eine Textur aus.
`MarkerSphere`	Liefert die Bounding Box der Spiegelkugel im Kamerabild im ein-Kamera Setup.
`FixedShere`	Liefert die Bounding Box der Spiegelkugel im Kamerabild im zwei-Kamera Setup.

Tabelle 3.1: Aufgaben der implementierten Klassen.

3.1.1 Scene

Die Klasse `Scene` enthält Daten, die in der Szene vorkommen. Das beinhaltet die Geometriedaten von 3D-Modellen, die Daten von Lichtquellen und Effekten sowie die Environment Maps. `Scene` ist für das Löschen all dieser Daten verantwortlich, alle anderen Klassen verwenden diese Daten nur. Welche Daten in einer `Scene` beinhaltet sind, wird in einer einfachen Textdatei angegeben. Das Format dieser Datei wird in folgendem Abschnitt beschrieben.

Das Szenenformat

Das Szenenformat ist sehr einfach aufgebaut. Jede Zeile beginnt mit einem Schlüsselwort, das angibt, welches Objekt aus den nachfolgenden Parametern erstellt werden soll. Es gibt z.Zt. zwei mögliche Schlüsselworte, nämlich `MODEL` und `EFFECT`.

Listing 3.1: Beispiel eines Scene Textfiles

```
MODEL 0 ../data/models/teapot.3ds       ../data/images/
MODEL 1 ../data/models/plane.3ds        ../data/images/
MODEL 2 ../data/models/watch.3ds        ../data/images/

EFFECT 0 CubeLighting
EFFECT 1 ManyShadows
EFFECT 2 ShadowOnly
```

Die Parameter des `MODEL` Objektes sind wie folgt:

- **ID** Eine eindeutige Nummer für das Modell.
- **Dateiname** Relativer Pfad zum Ordner und Dateiname des Modells.
- **Texturpfad** Relativer Pfad zum Ordner in dem Texturen zum Modell liegen.

Die Parameter des `EFFECT` Objektes:

- **ID** Eine eindeutigen Nummer für den Effekt.
- **Name** Eindeutiger Name des Effekts.

In Abb. 3.1 ist die Klasse `Scene` mit allen Klassen, die direkt mit ihr zusammenarbeiten, zu sehen. Sie können unterschieden werden in Klassen, die Eingabe liefern und Klassen, die Daten in der `Scene` verwenden. Die Eingabe kann weiter unterschieden werden in Eingabe, die nur einmal in der Initialisierungsphase geschieht und Eingabe, die zur Laufzeit ständig aktualisiert wird.

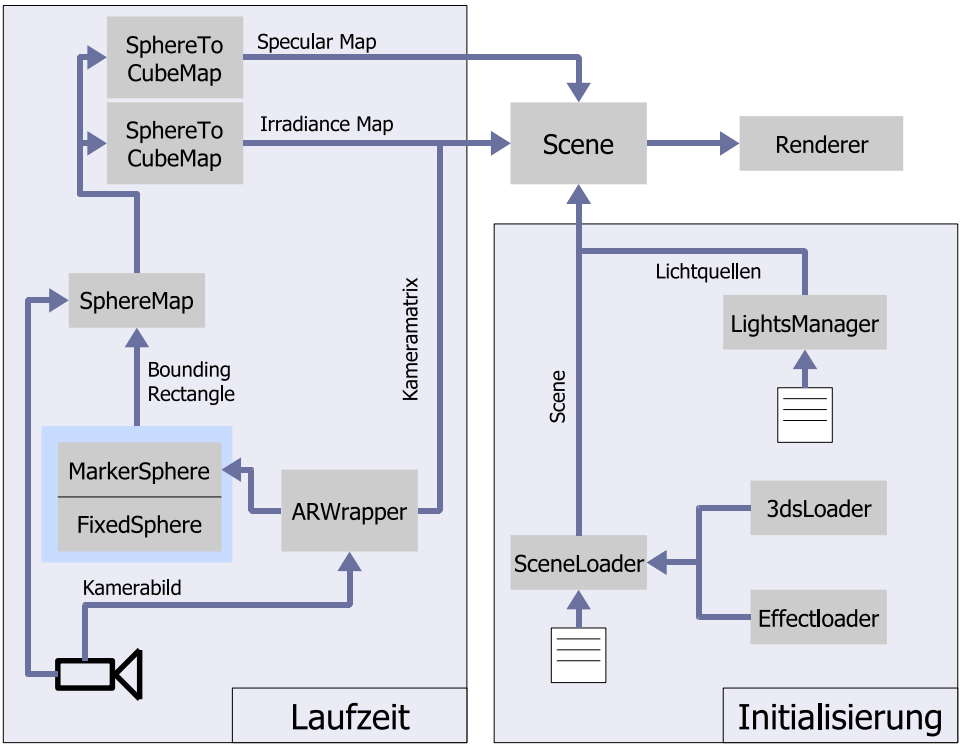


Abbildung 3.1: Die Klasse `Scene` mit allen Klassen, die mit ihr zusammenarbeiten.

Eingabe zur Initialisierung Um eine `Scene` zu erhalten, wird die Klasse `SceneLoader` verwendet. `SceneLoader` besitzt nur eine statischen Methode namens `loadSceneFromFile(string filename)`, die den Namen eines `Scene`-Textfiles bekommt und ein fertig geladenes Objekt zurückliefert. Er lädt alle 3ds-Modelle und Effekte. Modelle werden im Format `ARSModel` gespeichert, welches für das Rendering mit Vertex Buffer Objects optimiert ist. Effekte implementieren das Interface `RenderEffectInterface`, welches die Methoden zum Aktivieren und Deaktivieren des Effekts sowie einen Datentyp für Parameterübergabe vorschreibt.

Lichtquellen werden von der Klasse `LightManager` geliefert, der die Positionen von Lichtquellen aus einer Textdatei ausliest. Das Dateiformat ist ähnlich dem Szenenformat, jede Zeile beginnt mit einem Schlüsselwort. Gültige Schlüsselwörter sind für direktionale Lichter `DIRLIGHT` und für Spotlights das Schlüsselwort `PERSPLIGHT`. Im Dateiformat für Spotlights werden folgende Parameter definiert:

- **ID** Eindeutige Nummer der Lichtquelle.
- **At** Drei durch Leerzeichen getrennte Fließkomma Werte, die den Punkt angeben, auf den die Lichtquelle blickt.

- **Pos** Drei durch Leerzeichen getrennte Fließkomma Werte, die die Position der Lichtquelle angeben.
- **fov** Das Sichtfeld der Lichtquelle in Grad (gültige Werte sind 0.0 bis 180.0).
- **aspect** Das Seitenverhältnis des Spots.
- **near** Die Entfernung der near-plane zur Lichtquelle.
- **far** Die Entfernung der far-plane zur Lichtquelle. Schattenwerfende Objekte müssen sich zwischen near- und far-plane befinden.

Eingabe zur Laufzeit Einige Parameter ändern sich zur Laufzeit ständig. Zu diesen zählen die Matrix der virtuellen Kamera, die mittels einer Tracking Library aus dem realen Kamerabild extrahiert wird sowie die Environment Maps, die aus dem Bild der Spiegelkugel bzw. des Fischaugenobjektivs extrahiert werden.

Die Matrix der Kamera wird von der Klasse `ARWrapper` geliefert. Durch diese Klasse wird die Funktionalität von Tracking Libraries wie ARToolkit oder ARTag abstrahiert. Sie bekommt ein Bild als Eingabe und liefert eine Transformationsmatrix als Ausgabe.

Um die aktualisierten Environment Maps zu bekommen, wird zuerst das Bounding Rectangle der Spiegelkugel berechnet. Dafür sind die Klassen `MarkerSphere` (für das Ein-Kamera-Setup) bzw. `FixedSphere` (für das Zwei-Kamera-Setup) zuständig. MarkerSphere benötigt die Transformationsmatrix der realen Kamera für diese Aufgabe, da sich die Spiegelkugel in einer fixen Entfernung relativ zum Marker befindet.

Wenn das Bounding Rectangle der Spiegelkugel bekannt ist, kann die Kugel aus dem Kamerabild ausgeschnitten werden. Dafür ist die Klasse `SphereMap` zuständig. Sie erzeugt danach aus der reflektiven Sphere Map eine Irradiance Sphere Map. Die beiden Sphere Maps werden mit der Klasse `SphereToCubeMap` in Cube Maps konvertiert und an die `Scene` weitergegeben.

3.1.2 Renderer

Die Klasse `Renderer` speichert eine Liste von Verweisen auf Modelle, die in einem Renderdurchgang gerendert werden. Zusätzlich werden die lokalen Transformationen der Modelle sowie Verweise auf die Shadingeffekte der Modelle gespeichert. Die Modelle und Shadingeffekte selbst werden in der `Scene` gespeichert und vom `Renderer` verwendet. Dadurch ist es möglich, Modelle und Effekte in mehreren verschiedenen Renderern wiederzuverwenden. In Abb. 3.2 ist die Klasse `Renderer` mit allen Klassen, die mit ihr zusammenarbeiten, zu sehen.

Wie die `Scene` wird auch jeder `Renderer` aus einer Textdatei wie der folgenden generiert:

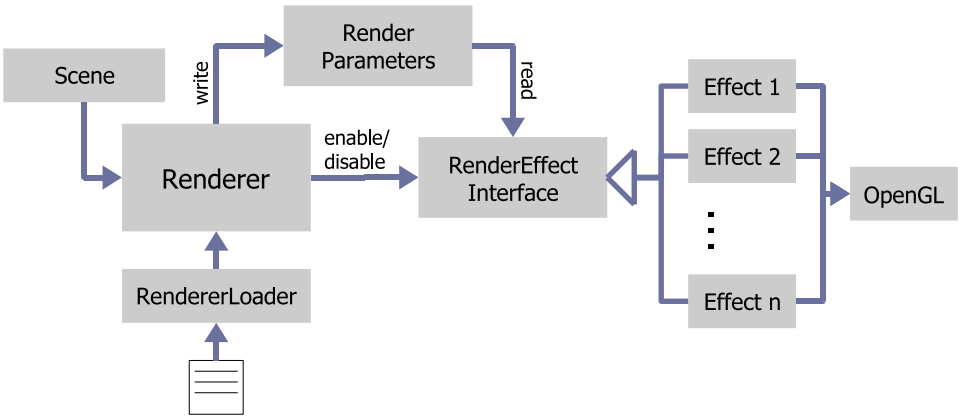


Abbildung 3.2: Die Klasse `Renderer` mit allen Klassen, die mit ihm zusammenarbeiten.

Listing 3.2: Beispiel einer Renderer Textdatei

```
IBLMODEL 0 1   60.0   100.0   0.0   0.0 0.0 0.0   1.0 1.0 1.0
IBLMODEL 6 0    0.0     0.0   0.0   0.0 0.0 0.0   1.0 1.0 1.0
```

In dieser Textdatei werden die Modelle definiert, die von dem jeweiligen `Renderer` gezeichnet werden sollen. Folgende Parameter müssen definiert werden:

- **ModelID** Nummer des Modells, das gezeichnet werden soll. Diese Nummer verweist auf ein Modell in der `Scene`.
- **EffectID** Nummer des Effekts, mit dem das Objekt gezeichnet werden soll. Diese Nummer verweist auf einen Effekt in der `Scene`.
- **Position** Drei durch Leerzeichen getrennte Fließkomma Werte, die die Position des Modells angeben.
- **Rotation** Drei durch Leerzeichen getrennte Fließkomma Werte, die die Rotation des Modells angeben.
- **Skalierung** Drei durch Leerzeichen getrennte Fließkomma Werte, die die Skalierung des Modells angeben.

Alle Effekte implementieren das Interface `RenderEffektInterface`. Dieses Interface schreibt folgende Methoden vor:

```
void setParameters(EnvParameters* env);
void enableEffect();
void disableEffect();
```

Der Renderer kann über dieses Interface Effekte aktivieren und deaktivieren. Zusätzliche Daten, die ein Effekt benötigt, werden über ein `EnvParameters`

Objekt übergeben. In diesem Objekt sind Statusinformationen wie aktuelle Matrizen, Lichtpositionen, benötigte Texturobjekte etc. gespeichert. `EnvParameters` erfüllt dadurch eine ähnliche Funktion wie die *OpenGL* Renderstates und erweitert diese insofern, als z. B. beliebig viele Lichtquellen möglich sind und Beleuchtungseinstellungen für Image Based Lighting gesetzt werden können. Für jedes Element, das gerendert wird füllt der Renderer ein Objekt des Typs `EnvParameters` mit allen vorhandenen Daten und übergibt es an den aktuell aktivierten Effekt. Jeder Effekt extrahiert die von ihm benötigten Daten und setzt die benötigten *OpenGL* bzw. Shader States.

3.2 Überblick über die Pipeline

In Abb. 3.3 sind die im vorigen Abschnitt beschriebenen Klassen eine erweiterte Version der Pipeline aus Kapitel 2.2 eingetragen. Jede Klasse ist in der Stufe der Pipeline zu sehen, für den sie verantwortlich ist. Während Abb. 3.3 sich auf die Stufen der Pipeline konzentriert, ist in Abb. 3.1 auf Seite 33 eine genauere Sicht auf die Ein- und Ausgabe jeder Klasse der Vorbereitungsschritte zu finden. In Abb. 3.2 auf Seite 35 sind die Ein- und Ausgaben jeder Klasse der Renderingschritte abgebildet.

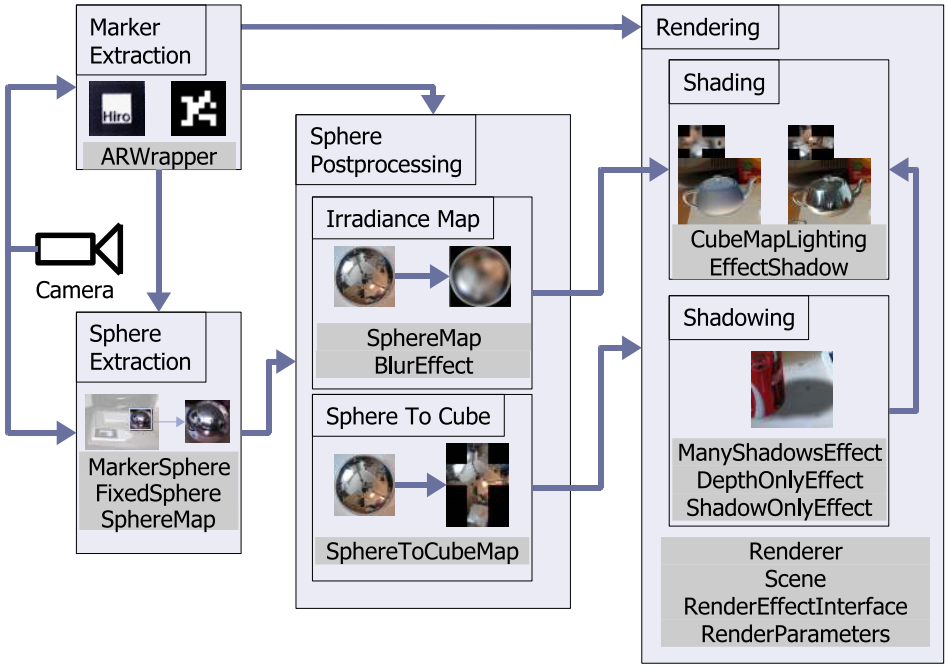


Abbildung 3.3: Die Pipeline mit den verantwortlichen Klassen.

Marker und Kugelextraxtion Das Bild der Kamera wird in `ARWrapper` geschickt. Dort wird die Transformationsmatrix der Kamera relativ zu einem Marker des Trackingsystems *ARToolkit* [KB99] berechnet. Die Transformationsmatrix kommt gemeinsam mit dem Kamerabild in den Bereich der Kugelextraktion, in dem von der Klasse `MarkerSphere` (im 1-Kamera Setup) oder der Klasse `FixedSphere` (im 2-Kamera Setup) das Bounding Rectangle der Spiegelkugel berechnet wird. In der Klasse `SphereMap` wird das Bild der Spiegelkugel aus dem Kamerabild ausgeschnitten. Damit ist eine reflektive Sphere Map vorhanden.

Sphere Map Nachbearbeitung Aus dieser reflektiven Sphere Map wird, ebenfalls von `SphereMap`, mit Hilfe der Textureffekts `BlurEffekt` eine Irradiance Map erzeugt. Die reflektive Sphere Map sowie die Irradiance Map werden von der Klasse `SphereToCubeMap` in Cube Maps konvertiert.

Rendering Die Cube Maps werden von der Klasse `CubeMapLightingEffect` zur Beleuchtung verwendet. Mittels `DepthOnlyEffect` können Tiefentexturen erzeugt werden, die von `ManyShadowsEffect` verwendet werden, um ein Schattenbild der Szene zu generieren. Dieses Schattenbild wird in `CubeMapLighting EffectShadow` und `ShadowOnlyEffect` zum Beschatten der Szene verwendet.

In den folgenden Abschnitten wird auf einige Stufen der Pipeline genauer eingegangen.

3.3 Extraktion der Sphere Map

Um eine Sphere Map aus einem Kamerabild zu bekommen, in dem sich irgendwo eine Spiegelkugel befindet, muss zuerst die Position und Größe (also das Bounding Rectangle) des Bildes der Kugel im Kamerabild ermittelt werden. Danach muss genau der Bereich des Kamerabildes, in dem die Kugel sichtbar ist, ausgeschnitten und in eine Textur kopiert werden. Die notwendigen Schritte werden im Folgenden genauer erklärt.

3.3.1 Bestimmen des Bounding Rectangles der Spiegelkugel

Um das Bild der Spiegelkugel aus dem Kamerabild ausschneiden zu können, muss das Bounding Rectangle der Spiegelkugel im Bild berechnet werden. Während sich im 2-Kamera Setup die Position der Spiegelkugel im Kamerabild nicht ändert und einmal zu Beginn des Programms festgelegt wird, muss sie im 1-Kamerasetup jedes Frame neu berechnet werden. Dafür ist die Klasse `MarkerSphere` zuständig.

In Abb. 3.4 ist eine schematische Darstellung der Vorgangsweise beim Berechnen des Bounding Rectangles zu sehen. Das gewünschte Ergebnis sind die linke untere und die rechte obere Ecke des Bounding Rectangles.

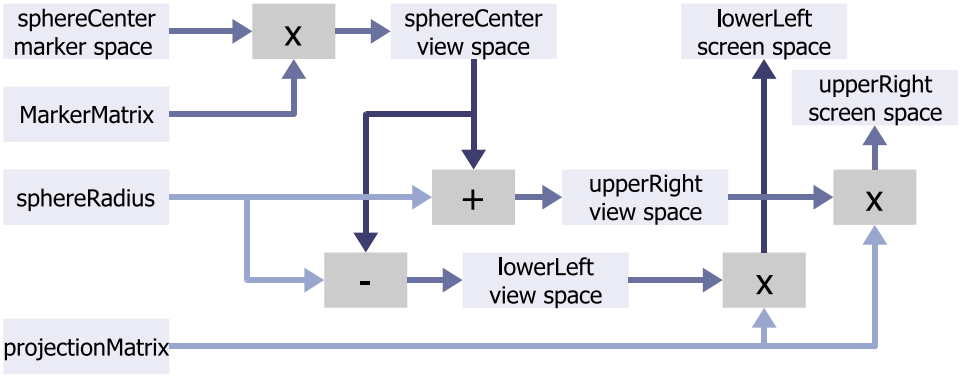


Abbildung 3.4: Bestimmen des Bounding Rectangles

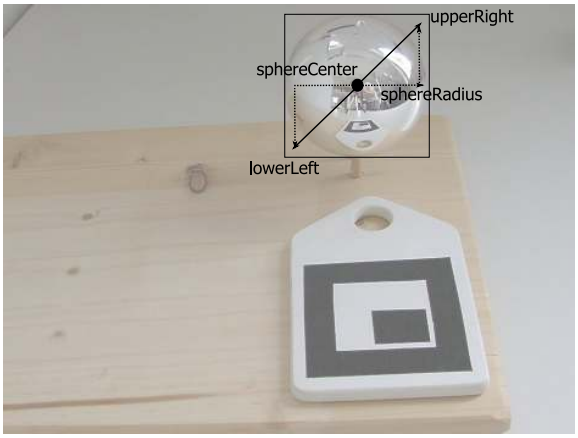


Abbildung 3.5: Durch Addition und Subtraktion des Radius `sphereRadius` der Spiegelkugel vom Mittelpunkt `sphereCenter` der Kugel lässt sich die linke unter Ecke `lowerLeft` und die rechte obere Ecke `upperRight` des Bounding Rectangles ermitteln.

Die Daten, die dazu zur Verfügung stehen, sind der Mittelpunkt der Kugel relativ zum Marker `sphereCenterMS`, die Transformationsmatrix vom optischen Marker zur Kamera `markerMatrix` aus `ARWrapper`, der Radius der Kugel `sphereRadius` und die Projektionsmatrix der Kamera `projectionMatrix`. Um den Mittelpunkt der Kugel relativ zur Kamera `sphereCenterVS` zu bekommen, wird `sphereCenterMS` mit `markerMatrix` multipliziert.

```
// multiply center of sphere with Marker Matrix
arsVec4Mat4x4Mult(sphereCenterMS, markerMatrix,
                  sphereCenterVS);
```

Danach werden die rechte obere Ecke des Bounding Rectangles in Kamerakoordinaten `upperRightVS` berechnet, indem zur x- und y- Koordinate des Mittelpunkts der Kugel der Radius `sphereRadius` addiert wird. Die linke

untere Ecke `lowerLeftVS` wird analog dazu durch Subtraktion des Radius' erreicht.

Um die Eckpunkte des Bounding Rectangles in Bildschirmkoordinaten `upperRightSS` und `lowerLeftSS` zu erlangen, werden die Punkte `lowerLeftVS` und `upperRightVS` mit der Projektionsmatrix der Kamera `projectionMatrix` multipliziert.

Listing 3.3: Berechnen des Bounding Rectangles in Augkoordinaten

```
// create the two corner points of the sphere
// rectangle in view space
lowerLeftVS[0] = sphereCenterVS[0] - sphereRadius;
lowerLeftVS[1] = sphereCenterVS[1] - sphereRadius;
lowerLeftVS[2] = sphereCenterVS[2];
lowerLeftVS[3] = sphereCenterVS[3];

upperRightVS[0] = sphereCenterVS[0] + sphereRadius;
upperRightVS[1] = sphereCenterVS[1] + sphereRadius;
upperRightVS[2] = sphereCenterVS[2];
upperRightVS[3] = sphereCenterVS[3];

// transform everything to screen space
arsVec4Mat4x4Mult(lowerLeftVS, projectionMatrix,
                  lowerLeftSS);
arsVec4Mat4x4Mult(upperRightVS, projectionMatrix,
                  upperRightSS);
```

3.3.2 Ausschneiden der Spiegelkugel aus dem Kamerabild

Nachdem das Bounding Rectangle der Spiegelkugel ermittelt wurde, kann das Bild der Spiegelkugel aus dem Kamerabild ausgeschnitten werden.

In Abb. 3.6 und Listing 3.4 ist die Vorgangsweise beim Ausschneiden der Spiegelkugel aus dem Kamerabild dargestellt. Als Eingabe ist eine Textur vorhanden, in der das Kamerabild gespeichert ist. Ausserdem ist die linke untere und rechte obere Ecke des Bounding Rectangles der Spiegelkugel im Kamerabild bekannt (das wie in Abschnitt 3.3.1 beschrieben bestimmt wurde).

Listing 3.4: Ausschneiden der Spiegelkugel aus dem Kamerabild

```
// set viewport to desired size of Sphere Map
glViewport(0, 0, sphereMapTexRes, sphereMapTexRes);
// draw quad with camera image texture
glBindTexture(GL_TEXTURE_2D, sourceTexId);
drawFullScreenQuad(lowerLeft[0] * 0.5 + 0.5,
                   lowerLeft[1] * 0.5 + 0.5,
                   upperRight[0] * 0.5 + 0.5,
                   upperRight[1] * 0.5 + 0.5);
```

```
// copy rendered image to texture
glBindTexture(GL_TEXTURE_2D, sphereMapTexId);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0,
                    sphereMapTexRes, sphereMapTexRes);
```

Es wird zuerst der *OpenGL* Viewport auf die gewünschte Größe der Sphere Map gestellt. Danach wird ein Rechteck gerendert, das den ganzen Viewport einnimmt. Darauf wird die Textur, in der das Kamerabild gespeichert ist, gebunden. Die Texturkoordinaten werden so gewählt, dass auf dem Rechteck nur der Bereich sichtbar ist, auf dem sich die Spiegelkugel befindet. Dieser Bereich ist durch die linke untere und rechte obere Ecke des Bounding Rectangles gegeben. Da das Bounding Rectangle in Bildschirmkoordinaten gegeben ist, müssen die Koordinaten noch in den Bereich von 0.0 bis 1.0 gebracht werden. Im gerenderten Rechteck ist nur mehr das Bild der Spiegelkugel zu sehen. Dieses Bild wird in die Sphere Map kopiert.

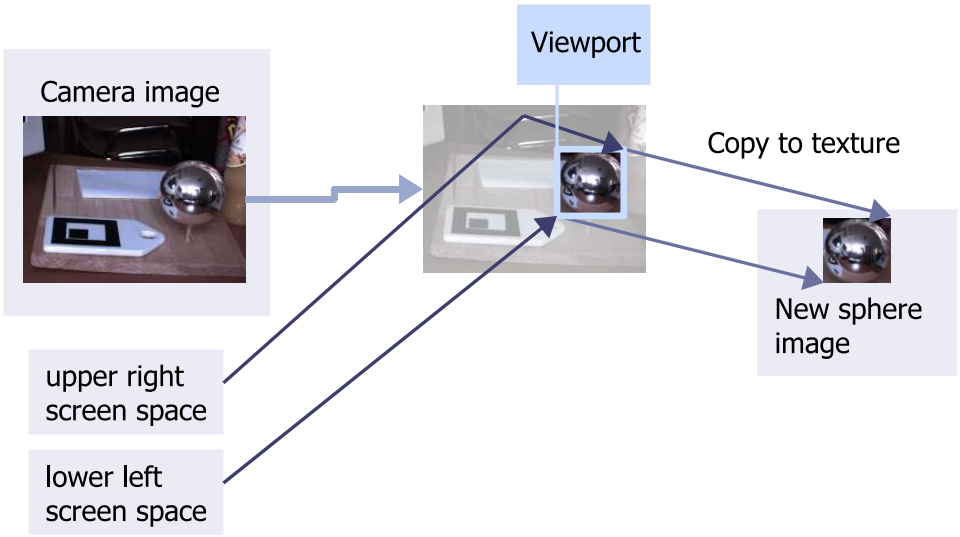


Abbildung 3.6: Ausschneiden der Spiegelkugel aus dem Kamerabild.

3.4 Nachbearbeiten der Sphere Maps

Nach dem Ausschneiden der Spiegelkugel aus dem Kamerabild ist eine Sphere Map für reflektierende Objekte vorhanden. Für andere Materialien muss diese Map gefiltert werden. Da das eigentliche Rendering mit Cube Maps funktioniert, müssen alle erstellten Sphere Maps danach in Cube Maps konvertiert werden. In Abb. 3.7 sind diese Arbeitsschritte der Nachbearbeitung dargestellt.

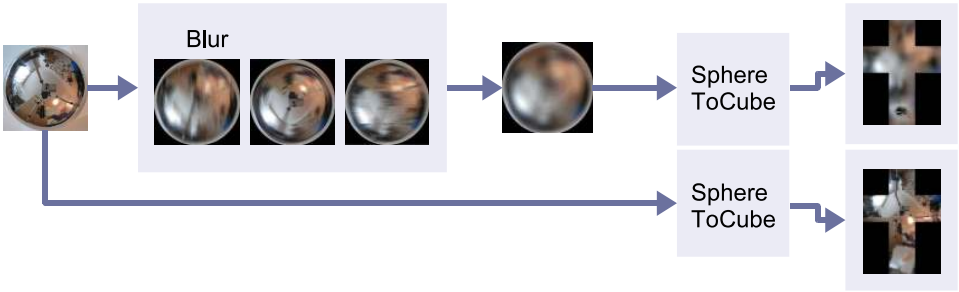


Abbildung 3.7: Arbeitsschritte beim Vorbearbeiten der Environment Maps. Es wird ein Unschärfefilter und eine Umwandlung in Cube Maps durchgeführt.

3.4.1 Erstellen von Irradiance Maps und matten Reflexionen

Das ausgeschnittene Bild der Spiegelkugel ist eine Sphere Map für perfekt reflektierende Objekte. Um diffuse oder matt reflektierende Oberflächen darzustellen, muss diese Sphere Map bearbeitet werden (wie in Abschnitt 2.1 beschrieben). Dazu kann entweder ein 2D-Filter oder ein zylindrischer Filter in 3 Achsen verwendet werden. Diese beiden Filter werden im Folgenden beschrieben.

Zweidimensionaler Unschärfefilter

Der zweidimensionale Unschärfefilter ist ein seperierbarer Box-Filter. Es wird also zuerst in eine Achse gefiltert und das Ergebnis danach in die zweite Achse bearbeitet. In Abb. 3.8 ist der Vorgang dargestellt.

Zur Realisierung des zweidimensionalen Unschärfefilters wird zuerst der Viewport auf die gewünschte Größe der weichgezeichneten Textur gesetzt. Danach wird ein Rechteck gerendert, das den gesamten Viewport einnimmt.

Es wird ein Shader verwendet, der wie ein linearer (eindimensionaler) Unschärfefilter wirkt (siehe Abb. 3.8 links). Um einen zweidimensionalen Filter auf eine Textur durchzuführen, wird der Shader zuerst in horizontaler Richtung auf das scharfe Bild durchgeführt. Auf das danach horizontal weich gezeichnete Bild wird derselbe Filter noch einmal in vertikaler Richtung angewandt, wie in Abb. 3.8 sichtbar. Der Datenfluss für die Durchführung eines 2D-Unschärfefilters ist ist Abb. 3.9 zu sehen.

Vertex Shader Der Vertex Shader bekommt als Eingabe folgende Parameter von der Applikation:

Abbildung 3.8: Weichzeichnen in zwei Dimensionen.

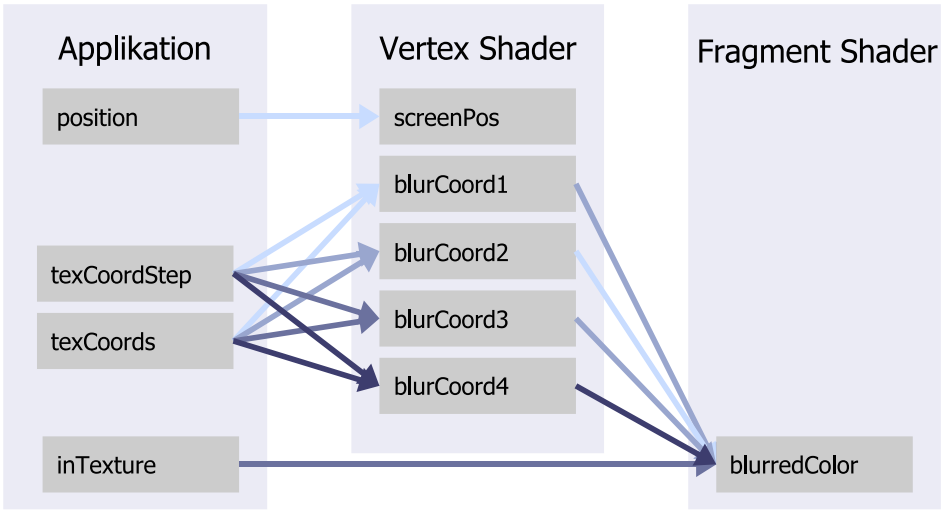


Abbildung 3.9: Datenfluss beim Unschärfefilter.

Listing 3.5: Übergabeparameter an den Blur-Vertexshader

```
// position of vertex in object space
float4 position : POSITION,
// original texture coordinates of vertex
float2 texCoord : TEXCOORD0,
// distance between 2 texel in map in texture coord
uniform float2 texCoordStep
```

Als Ausgabe soll folgende Struktur erzeugt werden. Im Vertex Shader existiert eine Struktur dieses Typs mit dem Namen `OUT`:

Listing 3.6: Ausgabe des Blur-Vertexshaders - Eingabe des Fragmentshaders

```
struct vert2frag {
  float4 position   : POSITION;
  float2 blurCoord1 : TEXCOORD0;
  float2 blurCoord2 : TEXCOORD1;
  float2 blurCoord3 : TEXCOORD2;
  float2 blurCoord4 : TEXCOORD3;
};
```

Die originalen Texturkoordinaten des Objekts (`texCoord`) werden mehrmals in diesselbe Richtung um z. B. ein Texel verschoben, damit ein Filtereffekt entsteht. Ob der Filter in horizontaler, in vertikaler, oder in eine beliebige andere Richtung durchgeführt wird, hängt von den übergebenen Werten für `texCoordStep` ab. Hat `texCoordStep` nur einen Eintrag in der x-Komponente, wird der Filter in horizontaler Richtung durchgeführt. Dasselbe gilt für einen Eintrag in der y-Komponente und vertikale Filter. Existieren sowohl Einträge in x und y, arbeitet der Filter diagonal.

```
OUT.blurCoord1 = texCoord +     texCoordStep;
OUT.blurCoord2 = texCoord + 2 * texCoordStep;
OUT.blurCoord3 = texCoord + 3 * texCoordStep;
OUT.blurCoord4 = texCoord + 4 * texCoordStep;
```

Fragment Shader Der Fragment Shader bekommt außer den vier Texturkoordinatensets aus dem Vertex Shader noch ein Texturobjekt mit dem Namen `inTexture` aus der Applikation. Es werden vier Texturzugriffe an den im Vertex Shader berechneten Stellen durchgeführt. Die Ergebnisse werden gemittelt:

```
float4 color0 = tex2D(inTexture, blurCoord1);
float4 color1 = tex2D(inTexture, blurCoord2);
float4 color2 = tex2D(inTexture, blurCoord3);
float4 color3 = tex2D(inTexture, blurCoord4);

outColor = (color0 + color1 + color2 + color3) * 0.25;
```

Parameter wie die Größe und Richtung des Filters werden von der Applikation über den an den Vertex Shader übergebenen Parameter `texCoordStep` gesteuert. Für den ersten Durchgang des Unschärfefilters (horizontaler Filter) wird dieser Parameter auf $(\Delta x, 0.0)$ gesetzt, für den zweiten Durchgang auf $(0.0, \Delta y)$. Die Variable Δ gibt die Veränderung in Texturkoordinaten (im Bereich 0.0 bis 1.0) an, die genau der Breite eines Texels entspricht und ist daher abhängig von der Seitenlänge s der Textur (siehe Abb. 3.10):

$$\Delta x = 1.0/s_x \tag{3.1}$$

$$\Delta y = 1.0/s_y \tag{3.2}$$

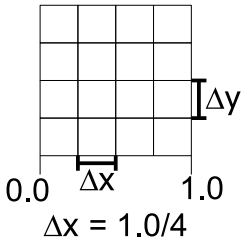


Abbildung 3.10: Berechnung der Verschiebung der Texturkoordinaten.

Das Rechteck, auf das die Textur gezeichnet wird, wird mit der Funktion `drawScreenQuad` gerendert, die als Parameter die Texturkoordinaten des linken unteren und des rechten oberen Verticis des Rechtecks bekommt. Damit kann das Texturbild beliebig verschoben und skaliert werden. Da im Vertex Shader `texCoordStep` immer zu den originalen Texturkoordinaten addiert wird, wirkt das unscharfe Bild leicht nach rechts oben verschoben. Um das zu verhindern, können die originalen Texturkoordinaten nach links unten verschoben werden, und zwar um 1.5 Texel (`texCoordStep * 1.5` - siehe Abb. 3.11). Das hat den weiteren Vorteil, dass die Texturzugriffe immer genau zwischen 2 Texeln zu liegen kommen. Der ausgelesene Farbwert ist durch bilineares Filtering daher eine Mischung aus den Farbwerten beider Texel. Der Blur-Effekt wird dadurch verstärkt.

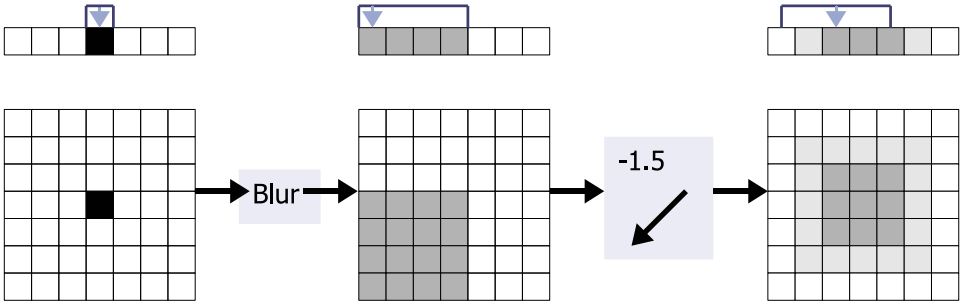


Abbildung 3.11: Auswirkung der Verschiebung der originalen Texturkoordinaten.

Mit folgendem Code-Stück wird der Filter in horizontaler Richtung durchgeführt. Danach wird das gerenderte, horizontal verwischte Bild auf eine temporäre Textur kopiert:

Listing 3.7: Filter in horizontaler Richtung

```
// bind shaders
...
// first pass
// set the unblurred texture as texture for the shader
```

```
cgGLSetTextureParameter(mCgTexture, mSharpTextureId);
// set texCoordStep as (deltax, 0)
cgGLSetParameter2f(mCgTexCoordStep, mDeltaX, 0.0f);
drawScreenQuad(-mTexCoordStep * 1.5, 0.0,
               1.0f - mTexCoordStep * 1.5, 1.0f);

// copy the horizontally blurred image to a texture
// bind the one dimensioal blurred texture
glBindTexture(GL_TEXTURE_2D, m1dBlurTextureId);
// copy the rendered image to the bound texture
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0,
                    mWidth, mHeight);
```

Für den zweiten Renderdurchgang wird das horizontal verwischte Bild an den Shader übergeben. Der Parameter `mCgTexCoordStep` wird so eingestellt, dass in vertikaler Richtung verwischt wird. Das danach in beide Dimensionen verwischte Bild wird auf die Ausgabetextur kopiert:

Listing 3.8: Filter in vertikaler Richtung

```
// second pass
// set the 1d blurred texture as texture for the shader
cgGLSetTextureParameter(mCgTexture, m1dBlurTextureId);
cgGLSetParameter2f(mCgTexCoordStep, 0.0f, mDeltaY);
drawScreenQuad(0.0f, -mTexCoordStep * 1.5,
               1.0f, 1.0f - mTexCoordStep * 1.5);

glBindTexture(GL_TEXTURE_2D, mBlurredId);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 0, 0,
                    mWidth, mHeight);
```

Zylindrischer Unschärfefilter

Wie in Abschnitt 2.4.1 erwähnt, ist ein zweidimensionaler Unschärfefilter eigentlich nicht geeignet um eine Environment Map zu filtern. Um auf jeden Texel der Environment Map einen gleich großen Bereich der Umgebung einwirken zu lassen, wird von [Lan05, Seite 100] ein zylindrischer Filter in 3 Achsen vorgeschlagen.

Der zylindrische Unschärfefilter wird nach demselben Prinzip durchgeführt wie der zweidimensionale Filter. Es wird also pro Achse, um die verwischt wird, ein Renderdurchgang durchgeführt und das Ergebnis jedes Renderdurchganges dient als Eingabe für den darauf folgenden Renderdurchgang.

Im Gegensatz zum zweidimensionalen Filter wird beim zylindrischen Filter in mehreren Achsen das Bild nicht auf ein Rechteck gezeichnet, sondern auf eine Kugel. Diese Kugel wird in jedem Renderdurchgang um eine Achse gedreht. Das entstehende Bild jedes Drehungsschrittes wird gespeichert und die einzelnen Bilder zusammengezählt.

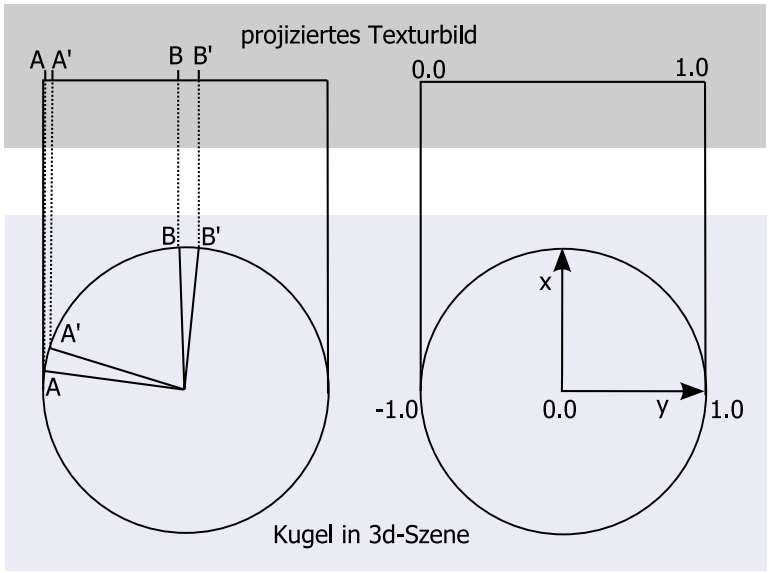


Abbildung 3.12: Zylindrischer Unschärfefilter

In Abb. 3.12 ist zu sehen, dass das ungefilterte Bild planar in der $x-y$-Ebene auf die Kugel projiziert wird. Die zum Rendern verwendete Kamera ist ebenfalls eine orthogonale Kamera, die in Richtung der negativen z-Achse auf die Kugel blickt. Wird die Kugel nun gedreht, verschiebt sich das auf die Kugel projizierte Bild. In der Kamera ist jedoch aufgrund der orthogonalen Projektion nur die Verschiebung in x- und y- Richtung zu sehen. Das bedeutet, dass Punkte, die sich am Rand der Kugel befinden (wie **A** in Abb. 3.12) und durch die Drehung auch in z-Richtung verschoben werden, im Kamerabild viel schwächer verschoben werden als Punkte die sich in der Mitte der Kugel befinden (wie **B** in Abb. 3.12) und sich in z-Richtung kaum bewegen (siehe Abb. 3.12).

Die planare Projektion der Textur auf die Kugel bedeutet, dass die Texturkoordinaten $\vec{T}_{(u,v)}$ jedes Verticis den x und y Koordinaten der Position $\vec{P}_{(x,y,z)}$ des Verticis entsprechen. Wenn die Kugel mit dem Radius r um den Ursprung erzeugt wird, müssen die Texturkoordinaten noch in den Bereich von 0.0 bis 1.0 gebracht werden.

$$\vec{T}_{(u,v)} = \vec{P}_{(x,y)} * \frac{r}{2} + \frac{r}{2} \tag{3.3}$$

In der Applikation wurde dies folgendermaßen umgesetzt, wobei r in diesem Fall den Wert 1.0 besitzt:

```
// u texture coordinate
sphereTexCoord[vertexIndex * 2    ] =
        spherePosition[vertexIndex * 3    ] * 0.5 + 0.5;
// v texture coordinate
```

```
sphereTexCoord[vertexIndex * 2 + 1] =
        spherePosition[vertexIndex * 3 + 1] * 0.5 + 0.5;
```

Die einfachste Möglichkeit, diesen Filter umzusetzen, ist, die Kugel mehrmals zu rendern und mit additivem Alpha Blending zu kombinieren. In Listing 3.9 ist der Code dafür zu sehen.

Listing 3.9: Zylindrischer Filter

```
// disable depth test for performance reasons
glDisable(GL_DEPTH_TEST);

// enable additive blending
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);

// bind the unblurred texture
glBindTexture(GL_TEXTURE_2D, heighttextureId);
// set alpha
glColor4f(1.0, 1.0, 1.0, 1.0/blurSteps);

// blur around x-axis
// render a lot of spheres and add the images
for(int i = -blurSteps/2; i < blurSteps/2; i++) {
  glPushMatrix();
    // rotate in x-axis
    glRotatef((float)i, 1.0, 0.0, 0.0);
    drawSphere(sphereCoord, sphereTexCoord, 50);
  glPopMatrix();
}
// copy the blurred image to the first texture
glBindTexture(GL_TEXTURE_2D, blurTexture1);
glCopyTexSubImage2D(GL_TEXTURE_2D,
                    0, 0, 0, 0, 0, 512, 512);

// repeat the above steps for y and z axis ...
```

Diese Methode, einen radialen Filter durchzuführen, ist zwar die einfachste, hat jedoch zwei gravierende Nachteile.

- Es werden viele Renderdurchgänge mit derselben Kugel gerendert, die relativ hoch tesseliert werden muss, um rund auszusehen. Daher müssen sehr viele Vertices verarbeitet werden, was u.U. viel Leistung benötigt.
- Es werden sehr viele Renderdurchgänge auf einen Buffer mit einer Genauigkeit von 8 Bit überblendet. Dadurch entstehen sehr leicht Banding-Artefakte.

Eine bessere Möglichkeit wäre, nur eine Rechteck zu rendern, und die verschobenen Texturkoordinaten für die Drehung der Kugel im Fragment

Shader auszurechnen. Damit wären (für den Preis eines längeren Fragment Shaders) beide oben genannten Probleme gelöst.

Daher wurde der zylindrische Filter in der vorliegenden Applikation letzten Endes nicht verwendet. Ein Vergleich der visuellen Unterschiede diffuser Beleuchtung mit 2D-Filter und zylindrischem Filter ist in Abschnitt 4.2 zu finden.

3.4.2 Erstellen von Cube Maps

Da das Rendering der virtuellen Objekte mit Cube Maps durchgeführt wird, ist es notwendig, die berechneten Sphere Maps in Cube Maps zu konvertieren. Der Prozess der Generierung von Cube Maps aus Sphere Maps beschränkt sich im Grunde auf das Berechnen von Positionen in der Sphere Map aus Positionen in der Cube Map sowie dem Auslesen der Pixel in der Sphere Map. Grafikhardware ist für alle diese Aufgaben sehr gut geeignet ist, daher werden die Berechnungen auf der GPU durchgeführt.

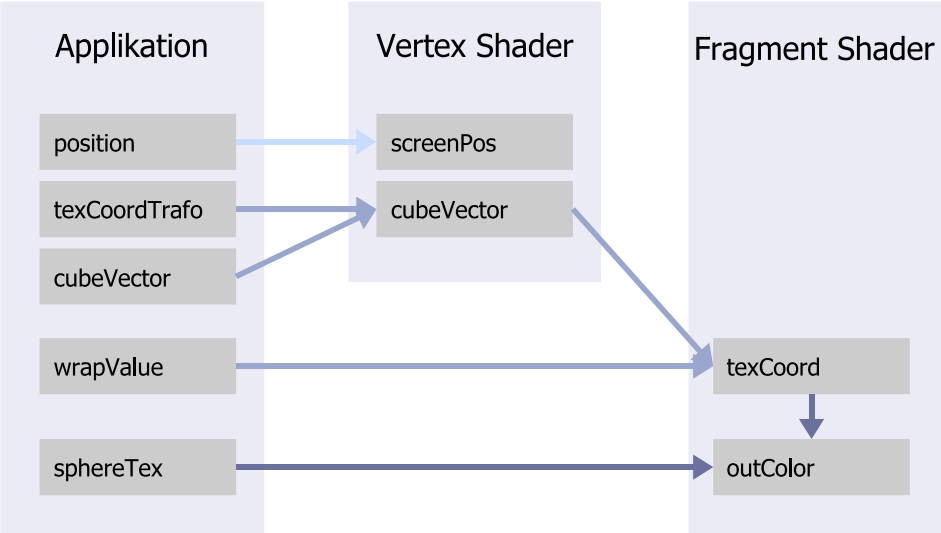


Abbildung 3.13: Datenfluss bei der Konvertierung von Sphere Maps in Cube Maps.

Die Cube Map kann man sich als um den Ursprung zentrierten Würfel vorstellen, wie in Abb. 3.14 dargestellt. Die Diagonale des Würfels hat die Länge 2.

Die Größe des Viewports wird auf die gewünschte Auflösung der Cube Map gesetzt. Für jede Seite der Cube Map wird ein Rechteck gerendert, das den gesamten Viewport füllt.

Vertex Shader Als Texturkoordinaten jedes Vertices wird der Vektor vom Ursprung des fiktiven Würfels zum Vertex mitgegeben. Die Textur-

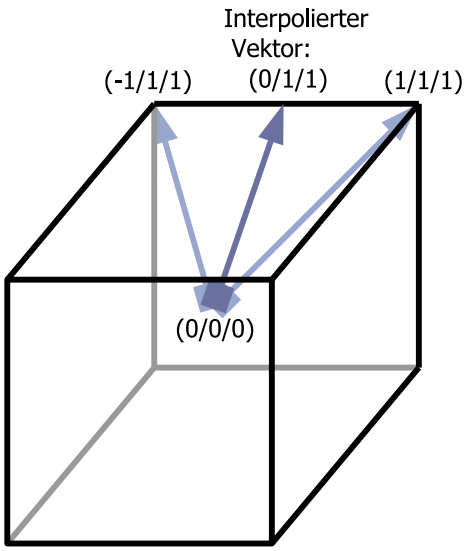


Abbildung 3.14: Die Cube Map kann man sich als um den Ursprung zentrierten Würfel vorstellen.

koordinaten werden über das Rechteck interpoliert und mit ihrer Hilfe kann im Fragment Shader für jeden Texel der Cube Map der zugehörige Texel in der Sphere Map berechnet werden.

In Abb. 3.13 ist der Datenfluss für die Konvertierung in Cube Maps dargestellt. Der Vertex Shader bekommt von der Applikation folgende Eingabe:

Listing 3.10: Eingabe des Vertex Shaders zum Konvertieren in Cube Maps

```
// vertex position in object space
float4 position : POSITION,
// vector from cube origin to vertex
float3 cubeVector : TEXCOORD0,
// transformation matrix from sphere map to world space
uniform float4x4 texCoordTrafo
```

Die Ausgabe zum Fragment Shader sieht folgendermaßen aus:

Listing 3.11: Ausgabe des Vertex Shaders - Eingabe des Fragment Shaders

```
struct vert2frag {
  float4 position   : POSITION;
  float4 cubeVector : TEXCOORD0;
};
```

Im Vertex Shader exisitiert eine Struktur des Typs `vert2frag` mit dem Namen `OUT`. Die Sphere Map kann aus jeder beliebigen Richtung aufgenommen werden. Daher wird die Transformationsmatrix `texCoordTrafo`, die die Transformation vom Weltkoordinatensystem (das Koordiantensystem der zukünftigen Cube Map) ins Koordinatensystem der Sphere Map beschreibt

(siehe Abb. 2.19), an den Vertex Shader übergeben. Durch Multiplikation mit `texCoordTrafo` wird `cubeVector` ins Sphere Map Koordinatensystem gebracht. Damit wird sichergestellt, dass die korrekten Texel aus der Sphere Map ausgelesen und in die Cube Map geschrieben werden.

```
OUT.cubeVector = mul(texCoordTrafo,
                     float4(cubeVector, 1.0f));
```

Fragment Shader Der Fragment Shader bekommt als Eingabe ein Objekt des Typs `vert2frag` vom Vertex Shader und ausserdem folgende Eingabe aus der Applikation:

Listing 3.12: Eingabe des Fragment Shaders aus der Applikation

```
// 2D texture with sphere map
uniform sampler2D sphereTex,
// determines how much of the environment is visible
uniform float wrapValue
```

Der Parameter `cubeVector` gibt an, welchem Vektor das aktuell berechnete Fragment entspricht. Durch die Interpolation von Werten zwischen mehreren Fragments hat dieser Vektor nicht mehr die Länge 1.0 und muss daher normalisiert werden:

```
float3 cubeVectorNorm = normalize(cubeVector);
```

Danach wird der normalisierte Halbwegvektor zwischen dem Augvektor und `cubeVector` berechnet. Der Augvektor `view` wird als $(0.0, 0.0, 1.0)$ angenommen:

```
float3 view = float3(0.0, 0.0, 1.0);
float3 newNormal = normalize(view + cubeVectorNorm);
```

Die `x` und `y`-Komponente von `newNormal` müssen nun vom Bereich -1.0 bis 1.0 in den Bereich 0.0 bis 1.0 gebracht werden, dann kann an der korrekten Stelle in der Sphere Map ausgelesen werden:

```
outColor = tex2D(sphereTex, newNormal.xy * 0.5 + 0.5);
```

Sphere Maps mit unterschiedlichem Sichtfeld Um Sphere Maps, in denen ein unterschiedlich großer Bereich der Umgebung sichtbar ist, verarbeiten zu können, gibt der Parameter `wrapValue` an, ein wie großer Teil der Umgebung in der Sphere Map sichtbar ist. Für eine perfekte Sphere Map, in der 360° der Umgebung sichtbar sind, ist dieser Wert 1.0. Da in einer Spiegelkugel jedoch normalerweise nur ca. 300°, in einem Fischaugenobjektiv gar nur 180° sichtbar sind, entsprechen Texel am Rand der Sphere Map anderen Vektoren als bei einer perfekten Sphere Map. Um diesem Umstand gerecht zu werden, wird `wrapvalue` mit dem Augvektor multipliziert.

```
float3 view = float3(0.0, 0.0, 1.0);
view *= wrapValue;
```

Wird der Wert von `wrapValue` nun verringert, verringert sich die Länge des Augvektors. Bei der Berechnung des Halbwegvektors wird daher `cubeVector` stärker gewichtet als `view` und Texel weiter am Rand der Sphere Map ausgelesen. Um Zugiffe auf ausserhalb der Sphere Map zu verhindert, wird nach der Berechnung von `newNormal` überprüft, ob `newNormal.z` kleiner als 0 ist. Wenn das der Fall ist, wird `newNormal.z` auf 0.0 gesetzt, der Vektor erneut normalisiert und damit am äußeren Rand der Sphere Map zugegriffen.

```
if(newNormal.z < 0.0) {
  newNormal = normalize(float3(newNormal.xy, 0.0));
}
```

3.5 Rendering

Nachdem wie in den vorigen Kapiteln beschrieben die erforderlichen Texturen generiert wurden, kann mit dem eigentlichen Rendering begonnen werden.

In Abb. 3.15 sind die Arbeitsschritte für das Rendering dargestellt. Zu Beginn werden die Schatten aller Lichtquellen berechnet. Dies geschieht in mehreren Renderdurchgängen zu je acht Lichtern. Die Ergebnisse der einzelnen Renderdurchgänge werden mit additivem Blending gemischt. Danach wird ein Unschärfefilter durchgeführt, um scharfe Kanten von Einzelschatten zu verwischen. Das Ergebnis ist ein Bild, in dem alle Schatten sichtbar sind. Dieses Bild wird in den Beleuchtungsprozess eingespeist.

Dort wird zuerst das Schattenbild mit der diffusen Beleuchtung multipliziert, um die Beschattung im Endergebnis sichtbar zu machen. Das Ergebnis dieser Multiplikation wird mit der Glanzlichtbeleuchtung addiert, die von Schatten unbeeinflusst bleibt. Schliesslich wird noch die Materialfarbe aufmultipliziert, um das endgültige Ergebnis zu erhalten.

In den folgenden Abschnitten werden die einzelnen Schritte im Detail beschrieben.

3.5.1 Beleuchtung

Beleuchtung mit SphereMaps

Im 1-Kamera-Setup wird die Beleuchtung ausschliesslich mit Sphere Maps durchgeführt. Im Folgenden wird der Datenfluss sowie die verwendeten Shader beschrieben.

Vertex Shader Im Vertex Shader erfolgt die Berechnung der Texturkoordinaten für die Zugriffe auf die Glanzlichttextur und Diffustextur. In

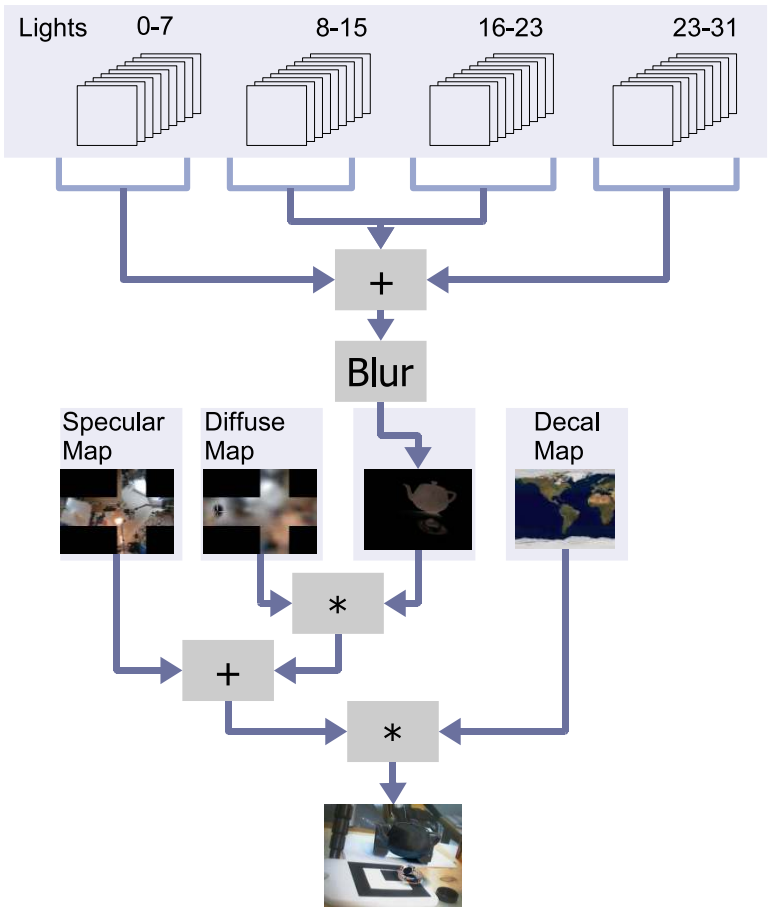


Abbildung 3.15: Übersicht über die Arbeitsschritte beim Rendering.

Abb. 3.16 ist der Datenfluss beim Beleuchten mit Sphere Maps dargestellt. Die Eingabe des Vertex Shaders von der Applikation ist in Listing 3.13 zu sehen. Die Ausgabe, die daraus berechnet werden soll, wird in Listing 3.14 dargestellt.

Listing 3.13: Eingabe des Vertexshaders für Sphere Map Lighting

```
float4 position : POSITION,  // position in obj space
float4 texCoord : TEXCOORD0, // texCoord of decal tex
float4 normal   : NORMAL,    // normal in obj space

uniform float4x4 modelViewPrj, // obj to screen space
uniform float4x4 modelView     // obj to view space
uniform float4x4 modelViewIT,  // inverse transposed mv
```

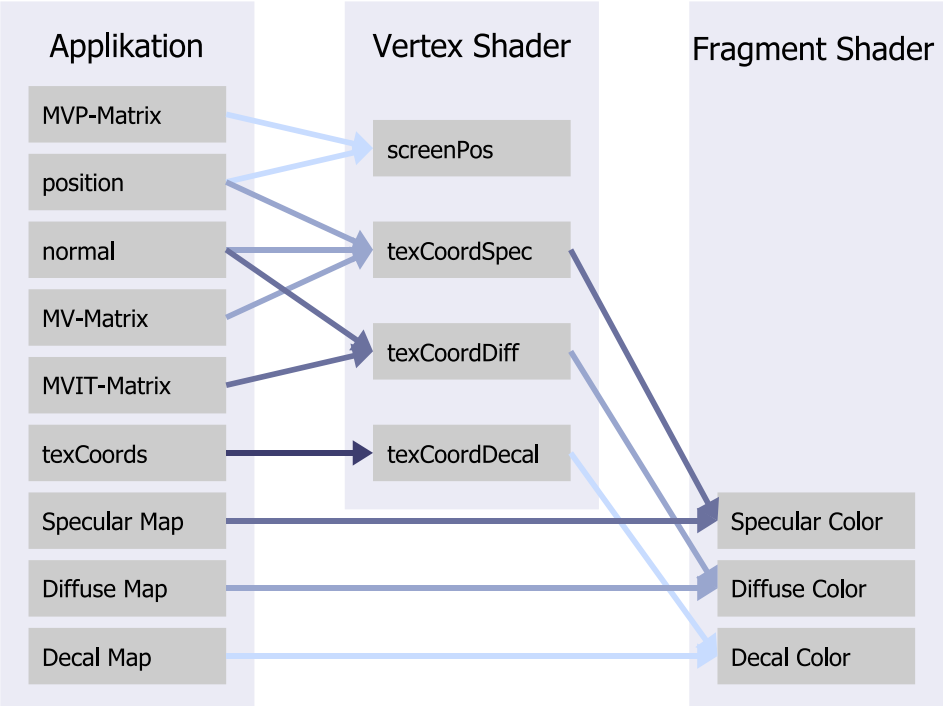


Abbildung 3.16: Datenfluss beim Beleuchten mit Sphere Maps.

Listing 3.14: Ausgabe des Vertexhaders - Eingabe des Fragmentshaders

```
struct vert2frag {
   // position in screen space
   float4 position   : POSITION;
   // texture Coordinates for diffuse texture lookup
   float2 texCoordDiff : TEXCOORD0;
   // texture Coordinates for specular texture lookup
   float2 texCoordSpec : TEXCOORD1;
   // texture Coordinates for decal texture lookup
   float2 texCoordDec  : TEXCOORD2;
};
```

Im Vertex Shader wird ein struct des Typs `vert2frag` mit dem Namen `OUT` verwendet.

Die Position `position` gibt die Position des Verticis in Objektkoordinaten an und wird durch Multiplikation mit der Modelview-Projection Matrix `modelViewPrj` in Bildschirmkoordinaten gebracht:

```
OUT.position = mul(modelViewPrj, position);
```

Für die Berechnung des reflektierten Augvektors in Augkoordinaten, der für die Berechnung des Sphere-Map Texturzugriffs notwendig ist, wird `position` auch in Augkoordinaten benötigt:

```
float3 posVSpace;
posVSpace = normalize(mul(modelView, position).xyz);
```

Da das Auge der Ursprung des Augkoordinatensystems ist, ist der Augvektor (der Vektor vom Vertex zum Auge) die invertierte Position des Verticis:

```
float3 viewVSpace = -posVSpace;
```

Für den Texturzugriff in die Diffustextur wird die Normale des Verticis in Augkoordinaten berechnet. Die Normale wird mit der invers transponierten Modelview Matrix von Objekt- in Augkoordinaten gebracht:

```
float3 normalVSpace;
normalVSpace = normalize(mul(modelViewIT, normal).xyz);
```

Danach werden die Texturkoordinaten für die Diffustextur als Halbwegvektor zwischen der Normale und dem Augvektor (beide in Augkoordinaten) berechnet und vom Bereich -1.0 bis 1.0 in den Bereich 0.0 bis 1.0 gebracht:

```
float3 texCoordDiff;
texCoordDiff = normalize(normalVSpace + viewVSpace);
OUT.texCoordDiff = float4(texCoordDiff * 0.5 + 0.5, 1.0);
```

Die Texturkoordinaten für die Glanzlichttextur entsprechen der Normale des Verticis in Augkoordinaten, in den Bereich zwischen 0.0 und 1.0 gebracht:

```
OUT.texCoordSpec = float4(normalVSpace * 0.5 + 0.5, 1.0);
```

Fragment Shader Der Fragment Shader bekommt zusätzlich zu einem `vert2Frag` Objekt names `IN` noch folgende Parameter von der Applikation:

Listing 3.15: Eingabe des FragmentShaders aus der Applikation

```
// Irradiance Map as Sphere map
uniform sampler2D diffuseMap,
// Specular Map as Sphere map
uniform sampler2D specularMap,
// decal texture of object
uniform sampler2D decalMap
```

In jede dieser drei Texturen wird mit den entsprechenden Texturkoordinaten aus dem Vertex Shader zugegriffen:

```
float4 diffCol  = tex2D(diffuseMap,  IN.texCoordDiff.xy);
float4 specCol  = tex2D(specularMap, IN.texCoordSpec.xy);
float4 decalCol = tex2D(decalMap,    IN.texCoordDec.xy);
```

Im Alpha-Kanal der Decal Map wird gespeichert, wie stark ein Objekt an welcher Stelle reflektiert (die Decal Map dient also gleichzeitig als *Gloss Map* - siehe [AMH02, Kap. 5.7.3]). Die endgültig verwendete Beleuchtungsfarbe ist also eine Mischung aus der Diffusfarbe und Glanzfarbe, abhängig vom Alphawert der Materialfarbe. Zum Mischen wird die Funktion `lerp` verwendet, die eine lineare Interpolation zwischen 2 Vektorwerten, abhängig von einem Skalarwert zwischen 0.0 und 1.0 durchführt:

```
float4 lightingCol = lerp(specCol, diffCol, decalCol.a);
```

Abschliessend wird die Farbe der Beleuchtung mit der Materialfarbe multipliziert, um den Eindruck einer beleuchteten, texturierten Oberfläche zu erzeugen:

```
outColor.rgb = lightingCol.rgb * decalCol.rgb;
```

Beleuchtung mit CubeMaps

Die Beleuchtung mit Cube Maps wird im 2-Kamera Setup sowie im Setup mit Fischaugenobjektiv verwendet. In den Klassen `CubeMapLightingEffect` und `CubeMapLightingEffectShadow` wird diese Aufgabe übernommen. Der Datenfluss sowie die verwendeten Shader werden im Folgenden beschrieben.

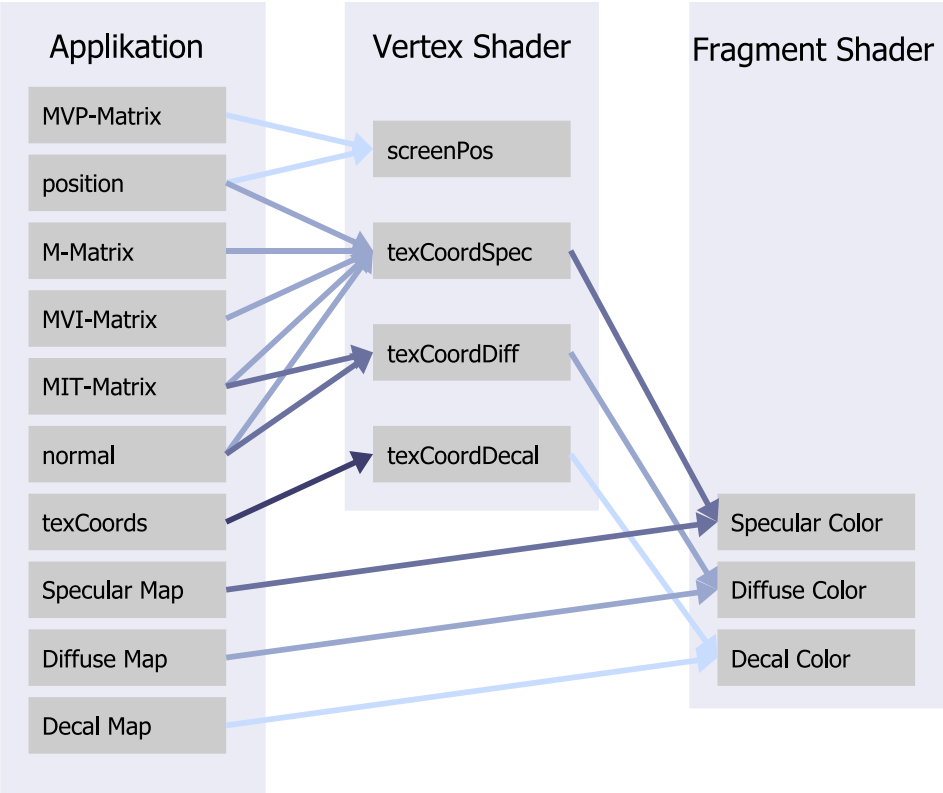


Abbildung 3.17: Der Datenfluss bei der Beleuchtung mit CubeMaps.

Vertex Shader Der Datenfluss für die Beleuchtung mit Cubemaps ist in Abb. 3.17 dargestellt. Der Vertex Shader sorgt für die Berechnung des Normalvektors und Reflexionsvektors in Weltkoordinaten. Dafür bekommt er folgende Eingabe von der Applikation:

Listing 3.16: Eingabe des Cubemap Vertex Shaders

```
float4 position : POSITION,  // position in object space
float4 texCoord : TEXCOORD0, // texCoord of decal texture
float4 normal   : NORMAL,    // normal in object space

uniform float4x4 modelViewProj, // object to screen space
uniform float4x4 modelViewInv   // view to object space
uniform float4x4 modelToWorldIT // obj to world inv trans
uniform float4x4 modelToWorld   // object to world
```

Daraus soll, genau wie bei der Beleuchtung mit Sphere Maps, folgende Ausgabe für den Fragment Shader berechnet werden. Auch hier wird im Vertex Shader ein struct des Typs `vert2frag` mit dem Namen `OUT` verwendet.

Listing 3.17: Ausgabe des Vertex Shaders - Eingabe des Fragment Shaders

```
struct vert2frag {
   // position in screen space
   float4 position     : POSITION;
   // texture Coordinates for diffuse texture lookup
   float3 texCoordDiff : TEXCOORD0;
   // texture Coordinates for specular texture lookup
   float3 texCoordSpec : TEXCOORD1;
   // texture Coordinates for decal texture lookup
   float2 texCoordDec  : TEXCOORD2;
};
```

Im Gegensatz zur Berechnung mit Sphere Maps sind die Texturkoordinaten für den Diffus- und Glanzlicht Texturzugriff Vektoren mit 3 Komponenten, da eine Cube Map anstelle einer 2D-Textur ausgelesen wird.

Für den diffusen Texturzugriff wird die Normale in Weltkoordinaten benötigt. Dafür wird sie mit der invers transponierten Model-Matrix namens `modelToWorldIT` multipliziert und danach wieder normalisiert:

```
float3 normalWSpace = mul(modelToWorldIT, normal).xyz;
normalize(normalWSpace);
OUT.texCoordDiff  = normalWSpace;
```

Die Berechnung des an der Normale reflektieren Augvektors wird in Weltkoordinaten durchgeführt. Zuerst wird die Position des Auges in Objektkoordinaten bestimmt. Da die Augposition nur in Augkoordinaten bekannt ist (dort ist sie der Ursprung, also (0/0/0)), wird sie durch Multiplikation mit der inversen Modelview-Matrix in Objektkoordinaten gebracht:

```
float4 camPosMSpace = mul(modelViewInv, float4(0,0,0,1));
```

Mit der Kameraposition und der Vertexposition in Objektkoordinaten kann nun der Augvektor (Vektor vom Vertex zum Auge) in Objektkoordinaten `viewMSpace` berechnet werden. Dieser wird durch eine Multiplikation mit der Model Matrix ins Weltkoordinatensystem gebracht:

```
float4 viewMSpace = camPosMSpace - position;
float4 viewWSpace = mul(modelToWorld, viewMSpace);
```

Die Texturkoordinaten für den Glanzlichttexturzugriff werden nun berechnet, indem der Augvektor `viewWSpace` an der Normalen `normalWSpace` reflektiert wird. Dafür wird die `reflect` Funktion verwendet:

```
OUT.texCoordSpec = reflect(viewWSpace.xyz, normalWSpace);
```

Fragment Shader Der Fragment Shader bekommt zusätzlich zu einem `vert2Frag` Objekt namens `IN` vom Vertex Shader noch folgende Parameter als Eingabe von der Applikation:

Listing 3.18: Eingabe des Fragment Shaders aus der Applikation

```
// decal texture of object
uniform sampler2D   decalMap,
// Irradiance Map as cube map
uniform samplerCUBE diffMap,
// specular map as cube map
uniform samplerCUBE specMap
```

Auf diese drei Texturen wird mit den aus dem Vertex Shader übergebenen Texturkoordinaten zugegriffen:

```
float4 diffColor  = texCUBE(diffMap, texCoordDiff.xyz);
float4 specColor  = texCUBE(specMap, texCoordSpec.xyz);
float4 decalColor = tex2D(decalMap,  texCoordShin.xy);
```

Die Berechnung der finalen Ausgabefarbe erfolgt analog zur Beleuchtung mit Sphere Maps. Auch hier ist das Verhältnis von Glanz- und Diffusfarbe im Alpha-Kanal der Materialfarbe gespeichert und mit der Funktion `lerp` gemischt. Die Materialfarbe selbst wird danach mit der Reflexionsfarbe multipliziert.

```
float4 reflectColor;
reflectColor = lerp(specColor, diffColor, decalColor.a);
outColor.rgb = reflectColor.rgb * decalColor.rgb;
```

Näheres dazu ist bei der Beleuchtung mit Sphere Maps in Abschnitt 3.5.1 beschrieben.

3.5.2 Beschattung

Große Anzahl an fixen Lichtquellen

In dieser Schattentechnik wird eine große Anzahl an Lichtquellen mittels Shadow Mapping berechnet. Dafür müssen Tiefenbilder der Szene aus der Sicht jeder Lichtquelle berechnet werden. Diese Tiefenbilder können beim Rendering mit Schatten verwendet werden, um zu ermitteln, ob ein gerenderter Punkt im Schatten liegt.

Positionierung von Lichtquellen Die Positionen der Lichtquellen werden der Einfachheit halber aus einer Textdatei eingelesen. Diese Textdatei ist in Abschnitt 3.1 beschrieben Die Menge an Lichtquellen ist nicht begrenzt, die Anzahl muss aber ein Vielfaches von 8 sein.

Rendering mit Schatten Die Vorgangsweise beim Rendering mit Schatten ist in Abb.3.15 dargestellt. Zuerst werden mehrere Renderdurchgänge mit den Schatten von jeweils 8 Lichtquellen gerendert. Die Ergebnisse der Durchgänge werden summiert, das Ergebnis wird gefiltert und ergibt ein Schattenbild, in dem die Schatten aller Lichtquellen sichtbar sind. Für virtuelle Objekte wird dieses Schattenbild wird in der Beleuchtungsberechnung zum Abdunkeln der Szene verwendet. Um auch auf reale Objekte Schatten werfen zu können, werden sogenannte „Phantomobjekte“ verwendet, die nur Schatten empfangen, aber selbst nicht sichtbar sind, wie von [Dra03] vorgeschlagen.

Um die Tiefentexturen auf die Geometrie der Szene zu projizieren, muss jeder Vertex mit der Model-View-Projection Matrix jedes Lichtes transformiert werden. Die transformierte Position dient als Texturkoordinate für den Texturzugriff im Fragment Shader und muss daher vom Vertex- an den Fragment Shader übergeben werden. Da üblicherweise nur 8 Texturkoordinatensets übergeben werden können [1], können nicht alle Schatten in einem Renderdurchgang berechnet werden. Aus diesem Grund werden die Schatten in mehreren Passes von jeweils 8 Schatten gerendert.

Diese Passes werden mit additivem Alpha Blending miteinander kombiniert. In der Ausgabe der Fragmentshader sind beschattete Bereiche hell und beleuchtete Bereiche dunkel. Daher entsteht ein invertiertes Schattenbild der Szene, in dem die Schatten aller Lichtquellen akkumuliert sind. Dieses Schattenbild wird in einer Textur gespeichert.

Um die Kanten einzelner Schatten zu verwischen, wird das Schattenbild anschliessend mit einem Unschärfe-Filter bearbeitet. Dieser Filter ist ein seperierbarer 2D Gauss'scher Unschärfefilter in 2 Renderdurchgängen, übernommen von der Klasse `BlurEffect`. Im ersten Renderdurchgang wird ein Rechteck auf einen Viewport in der Größe der Textur gerendert. In einem einfachen Fragmentshader werden 4 Texturzugriffe durchgeführt, die alle um horizontal um 1 Texel zueinander verschoben sind. Dadurch wird das Bild horizontal verwischt. Das Ergebnis wird in eine zweite Textur kopiert und als Eingabe für denselben Shader verwendet, die Texturkoordinaten sind nun aber um jeweils ein Pixel vertikal verschoben. Der Prozess wird in Abschnitt 3.4.1 dargestellt.

Danach werden die Objekte mit Beleuchtung gerendert, wie in Kapitel 3.5.1 beschrieben. Die Schattentextur wird aus der Sicht der Kamera auf

[1] Ab Shader Model 3.0 sind es 10.

die 3d-Szene projiziert und im Fragment Shader mit der diffusen Beleuchtung multipliziert. Spiegelungen bleiben unbeeinflusst von Schatten.

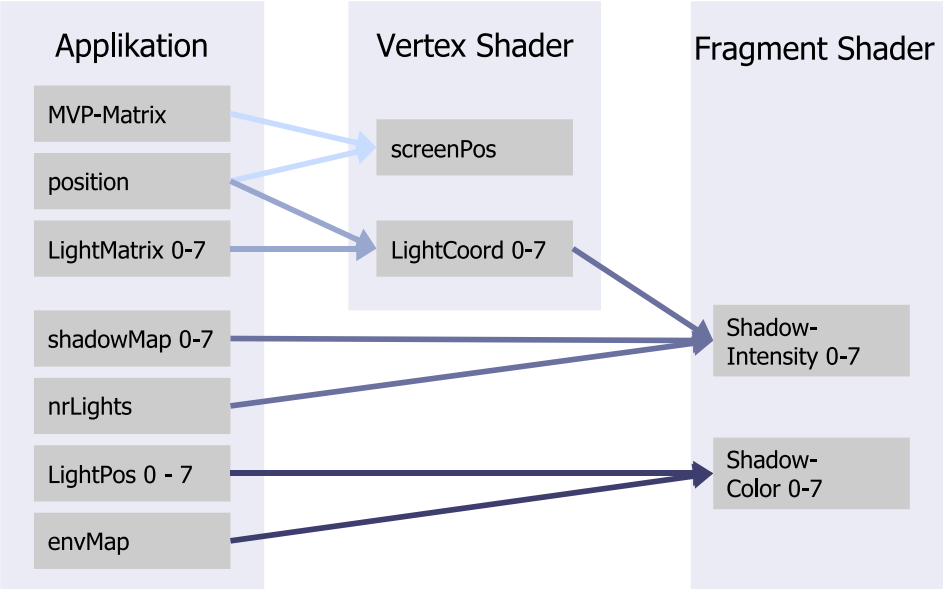


Abbildung 3.18: Datenfluss beim Rendering von Schatten

In Abb. 3.18 ist der Datenfluss beim Rendern der Schatten dargestellt. Die Applikation übergibt die Vertexdaten, die Modelview-Projection Matrix der Kamera sowie die Modelview-Projection Matrizen der 8 zu rendernden Lichter an den Vertex Shader.

Listing 3.19: Eingabe des Vertexshaders

```
// the position of the vertex in object space
float4 position : POSITION,
// the mvp matrix of the camera
uniform float4x4 modelViewProj,
// the mvp matrix of light 0 to 7
uniform float4x4 shadowMatrix0,
   ...
uniform float4x4 shadowMatrix7
```

Die Ausgabe soll die Position des Verticis in Bildschirmkoordinaten der Kamera sowie in Texturkoordinaten der acht Lichter sein. Diese Daten werden interpoliert und dienen als Eingabe für den Fragment Shader.

Listing 3.20: Ausgabe des Vertexshaders - Eingabe des Fragmentshaders

```
struct vert2frag {
  float4 position       : POSITION;
  float4 shadowCoord0   : TEXCOORD0;
       ...
```

```
  float4 shadowCoord7   : TEXCOORD7;
};
```

Der Vertex Shader transformiert jeden Vertex mit der Matrix der Kamera um seine Position in Bildschirmkoordinaten zu erlangen sowie mit den Matrizen der Lichter für die Texturkoordinaten für die Tiefentexturzugriffe. Die Ergebnisse werden in die Variable `OUT` vom Typ `struct vert2frag` gespeichert:

```
OUT.position = mul(modelViewProj, position);
OUT.shadowCoord0 = mul(shadowMatrix0, position);
     ...
OUT.shadowCoord7 = mul(shadowMatrix7, position);
```

Der Fragment Shader bekommt als Eingabe ausserdem von der Applikation die Texturobjekte für die Tiefentexturen und die Anzahl an Lichtern, die insgesamt berechnet werden müssen.

Listing 3.21: Eingabe des Fragmentshaders von der Applikation

```
uniform sampler2D shadowMap0,
     ...
uniform sampler2D shadowMap7,
uniform float nrLights
```

Im Fragment Shader erfolgt ein Texturzugriff auf jede der 8 Tiefentexturen an der Stelle der übergebenen Texturkoordinaten. Die Ergebnisse der Texturzugriffe werden invertiert und addiert.

```
outColor = float4(0.0f, 0.0f, 0.0f, 1.0f);

outColor += 1.0 - tex2Dproj(shadowMap0, shadowCoord0);
     ...
outColor += 1.0 - tex2Dproj(shadowMap7, shadowCoord7);
```

Durch Division durch 8.0 wird das Ergebnis auf den Bereich zwischen 0.0 und 1.0 gebracht. Danach wird noch durch die Anzahl an Passes dividiert. Dieser letzte Schritt ist notwendig um die Ergebnisse der einzelnen Passes mit Alpha Blending kombinieren zu können.

```
outColor /= 8.0;
// nrLights / 8 is the number of passes
outColor *= 8.0 / nrLights;
```

Unterschiedliche Intensitäten. Um die Intensität der Schatten von der Environment Map abhängig zu machen, benötigt der Fragment Shader als zusätzliche Information die Positionen der Lichtquellen sowie eine Repräsentation der Environment Map.

Listing 3.22: Zusätzliche Eingabe für den Fragmentshader für unterschiedliche Intensitäten

```
uniform float4 lightPos0,
     ...
uniform float4 lightPos7,
uniform samplerCUBE envMap,
```

Für jede Lichtquelle wird der Wert der Environment Map in Richtung der Lichtquelle ausgelesen. Der ausgelesene Wert wird mit dem invertierten Shadow-Map Wert der Lichtquelle multipliziert. Dadurch nehmen Regionen, die stark im Schatten sind, die Farbe der Lichtquelle stark an, Regionen, die beleuchtet sind, bleiben jedoch nahezu unbeeinflusst.

```
outColor += (1.0 - tex2Dproj(shadowMap0, shadowCoord0)) *
             texCUBE(envMap, lightPos0.xyz);
     ...
outColor += (1.0 - tex2Dproj(shadowMap7, shadowCoord7)) *
             texCUBE(envMap, lightPos7.xyz);
```

Beleuchtung mit Schatten. Der neue Fragment Shader für die Beleuchtung ist in Abb. 3.19 dargestellt. Er erhält als zusätzlichen Input die Textur mit den akkumulieren Schatten. Um diese Textur auf die Szene zu projezieren, wird die in Bildschirmkoordinaten transformierte Vertexposition $\vec{P}_B$ als Texturkoordinaten übergeben.

Listing 3.23: Zusätzliche Eingabe für den Shader für Rendering mit Schatten

```
// texture with accumulated shadows
uniform sampler2D shadowTex
// fragment position in screen coordinates
float4 texCooScreen: TEXCOORD3,
```

Die Koordinaten $\vec{P}_B$ werden von Bildschirmkoordinaten in Texturkoordinaten $\vec{P}_T$ gebracht, da Bildschirmkoordinaten im Bereich zwischen -1.0 und 1.0 liegen, Texturkoordinaten aber im Bereich zwischen 0.0 und 1.0 (siehe Abschnitt 2.5.1):

$$\vec{P}_T = \vec{P}_B * 0.5 + 0.5. \tag{3.4}$$

Die Texturkoordinaten $\vec{P}_T$ werden für den Texturzugriff auf die Schattentextur verwendet.

```
shadowCol = tex2D(shadowTex,
           (texCooScreen.xy/texCooScreen.w) * 0.5 + 0.5);
```

Da Schatten nur auf diffusen Oberflächen sichtbar sind, nicht jedoch in reflektiven, wird das Ergebnis der diffusen Beleuchtung mit dem Ergebnis des Schattentexturzugriffs multipliziert. Da in der Schattentextur beschattete Bereiche hell und beleuchtete dunkel dargestellt sind, wird der aus der Schattentextur ausgelesene Wert invertiert.

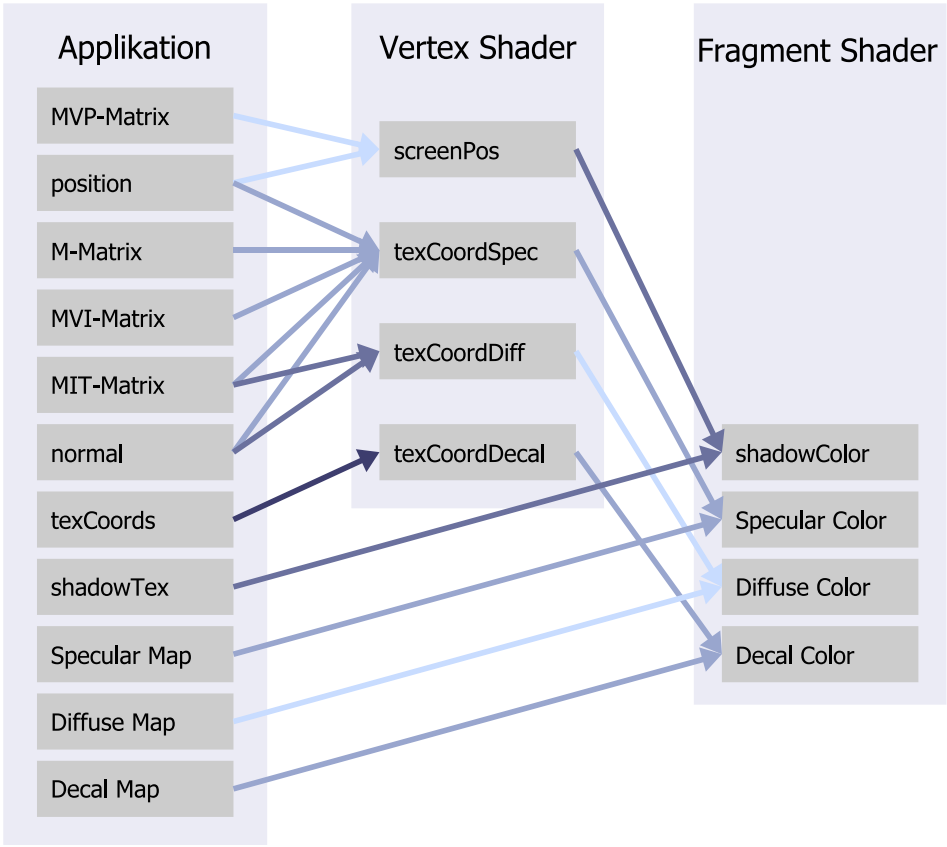


Abbildung 3.19: Beleuchtung mit Schatten.

```
diffuseColor *= 1.0 - shadowCol;
```

Der Rest des Shaders ist wie in Kapitel 3.5.1 beschrieben.

Verbesserungsmöglichkeiten

- Zur Zeit werfen immer alle Lichtquellen Schatten. In Situationen, in denen nur ein sehr kleiner Teil der Environment Map hell ist, sind nur wenige dieser Schatten sichtbar. Eine einfache Optimierung wäre daher, nur die Schatten dieser Lichtquellen zu berechnen, die hell genug sind, um sichtbare Schatten zu werfen. Je nach Kontrast der Environment Map könnten damit u.U. mehr als 50% aller Lichtquellen eingespart werden.

- Beim Rendering der Schatten muss für jede Lichtquelle jeder Vertex jedes Objektes mit der Transformationsmatrix der Lichtquelle transformiert werden. Dazu kommt, dass die Tiefentextur jeder Lichtquelle jedes Frame erneuert werden muss Das bedeutet, dass bei 64

Lichtquellen pro Vertex 129 (64 Lichtquellen beim Tiefentextur erstellen, 64 Lichtquellen beim Rendern und die Kameratransformation) Matrixmultiplikationen durchgeführt werden müssen. Dadurch, dass pro Frame nur die Tiefentextur einer Lichtquelle erneuert wird, wird diese Zahl auf 66 reduziert. Beim Rendern in die Tiefentextur und beim Rendern der Schatten werden jedoch nahezu idente Matrizen verwendet. Es ist nun möglich, die Transformation nur einmal durchzuführen, die transformierten Vertices zwischenzuspeichern und sowohl beim Rendering in die Tiefentexturen als auch beim Schattenrendering direkt mit den transformierten Vertices zu arbeiten. Dazu müssten die transformierten Vertices direkt in ein Vertex Buffer object gerendert werden. (siehe [Render To VBO]). Der Vorteil dieser Methode wäre, dass die Transformation der Vertices mit der Matrix der Lichter nur durchgeführt werden muss, wenn die Schattentextur des Lichtes erneuert wird, also nur für ein Licht pro Frame.

- Im Moment sind die Positionen und damit die Richtungen des Schattenwurfes unveränderbar. Bewegungen von Schatten können nur über Veränderung der Intensität erreicht werden. Wenn eine sehr kleine reale Lichtquelle zwischen zwei virtuellen Lichtquellen liegt, wird sie durch zwei mittlere Schatten dargestellt. Eine Verbesserungsmöglichkeit wäre, die hellste Stelle im Umkreis jeder Lichtquelle zu finden und die Lichtquelle dorthin zu verschieben.

- Bei sehr diffusen Lichtverhältnissen (bewölkter Himmel, Raum mit indirektem Licht) sind Schatten sehr weich. Wenn nur harte Schatten verwendet werden, ist ein große Anzahl an Lichtquellen notwendig, um diesen Effekt zu erreichen. Eine andere Möglichkeit wäre, anstatt einfachen Shadow Maps eine Soft Shadow Technik wie *Smoothies* (siehe [CD03]) zu verwenden. Dadurch könnte mit wesentlich weniger einzelnen (dafür weicheren) Schatten derselbe Effekt erreicht werden.

Kapitel 4

Ergebnisse

4.1 Reflexionen

In beiden Versionen der Applikation (1 Kamera und 2 Kameras) können Reflexionen der realen Umgebung in den virtuellen Objekten überzeugend dargestellt werden. Auch Bewegungen in der Umgebung werden unmittelbar in der Reflexion sichtbar. In einigen Situationen ist jedoch zu sehen, dass die Reflexion aus der Position der Spiegelkugel aufgenommen wurde und für weiter davon entfernte Objekte nicht ganz korrekt ist (siehe Abb. 4.1(a)).

Verchromte Oberflächen wirken im Ein-Kamera Setup meist sehr überzeugend, da die Environment Map aus demselben Kamerabild extrahiert wird, das auch die Hintergrundszene darstellt. Aus diesem Grund wirkt alles „wie aus einem Guss“. Im 2-Kamera-Setup ist die Farbgebung von Hintergrund und Reflexion oft unterschiedlich, da ein andere Kamera für die Environment Map verwendet wird. Reflexionen auf großen Objekten wirken hingegen im 1-Kamera-Setup verschwommen, wenn die Spiegelkugel im Kamerabild nur sehr klein zu sehen ist und die Auflösung der Environment Map daher sehr gering ist.

Wenn im 2-Kamera-Setup anstatt einer Spiegelkugel eine Kamera mit Fischaugen-Objektiv verwendet wird, wirken Reflexion klarer und schärfer als mit einer Spiegelkugel. Dadurch, dass das Fischaugen Objektiv nur einen Bereich von maximal 180° einsieht, ergeben sich aber Probleme. Entweder wird die entstehende Environment Map behandelt, als ob sie einen Bereich von 360° umfasst. Dadurch wird an den Rändern eines Objektes, an denen normalerweise die Umgebung hinter dem Objekt zu sehen sein müsste, die Umgebung neben dem Objekt sichtbar, was unter Umständen als falsch erkannt wird. Die andere Möglichkeit ist, jeden Teil der Umgebung an der richtigen Stelle in der Reflexion wiederzugeben (siehe Abschnitt 2.4.2), wodurch 50% der Environment Map undefiniert bleibt und interpoliert werden muss. Das ist in der Reflexion als stark unscharfer Bereich am Rand der Objekte zu sehen (siehe Abb. 4.3(b)).

(a) (b)

Abbildung 4.1: In der Reflexion der Teekanne in Bild (a) ist der weisse Marker links zu sehen, obwohl er sich auf der rechten Seite des Objektes befindet. In Bild (b) befindet sich die Kamera und das Objekt an derselben Position, daher ist die Reflexion korrekt.

Performance Die Arbeitsschritte bei der Berechnung der Reflexion sind grunsätzlich nicht sehr rechenaufwändig. Es handelt sich um einen Render-To-Texture Schritt beim Ausschneiden der Sphere Map und um Rendering mit Environment Mapping. Beim 2-Kamera-Setup kommt die Umwandlung von Sphere Maps in Cube Maps dazu, wobei 6 Render-To-Texture Schritte mit leichten Fragment Shadern durchgeführt werden müssen. Das 1-Kamera-Setup ist selbst auf einer nicht shaderfähigen Grafikkarte wie einer nvidia Geforce 2 lauffähig, das 2-Kamera Setup ist auch auf „low-end“ Shader Model 2.0 Grafikkarten nur durch die Framerate der Kamera begrenzt.

(a) Spiegelkugel (b) Fischauge

Abbildung 4.2: Weiter hinten liegende Objekte werden im Setup mit dem Fischauge nicht gesehen. Die orange Getränkepackung ist im Rendering mit der Spiegelkugel in der Reflexion zu sehen, beim Rendering mit Fischaugenobjektiv jedoch nicht.

Vergleich echter und virtueller Objekte Um die Qualität des Renderings mit Reflexionen zu testen, wurde eine echte und eine virtuelle Getränkedose nebeneinander platziert. Die Ergebnisse sind in Abb. 4.1 zu sehen, einmal mit dem 1-Kamera Setup, einmal mit dem 2-Kamera Setup mit Spiegelkugel. Es ist zu sehen, dass die Reflexion im 2-Kamera Setup deutlich detaillierter ist.

(a) 2-Kamera Setup (b) 1-Kamera Setup

Abbildung 4.3: Vergleich von echter und virtueller Getränkedose. In (a) ist das Rendering der virtuellen Dose mit dem 2-Kamera Setup, in (b) mit dem 1-Kamera Setup zu sehen.

4.2 Diffuse Oberflächen

Diffuse Oberflächen werden mit den vorgestellten Algorithmen überzeugend dargestellt. Die Helligkeit und Farbe der Umgebung wird von den virtuellen Objekten übernommen , sie passen sich dadurch besser in die reale Umgebung ein, wie in Abb. 4.4 zu sehen ist. Noch wichtiger als bei Reflexionen ist hier ein korrekter Weissabgleich für beide Kameras im 2-Kamera Setup.

Abbildung 4.4: Die gelbe Getränkepackung auf der rechten Seite und die rote Dose auf der linken Seite beeinflussen die Farbe der Teekanne.

Für die Erstellung von Irradiance Maps für diffuse Oberflächen wurden zwei verschieden Arten von Filtern getestet. Der erste ist ein zweidimensionaler Unschärfefilter, der auf die Sphere-Map durchgeführt wird, als ob sie eine gewöhnliche 2D-Textur wäre (siehe Abschnitt 2.4.1). Der zweite ist eine sphärischer Filter, der bei der Filterung darauf Rücksicht nimmt, dass ein gleich großer Bereich der *Umgebung* auf jedes Texel wirkt (im Gegensatz zum 2D-Filter, bei dem ein gleich großer Bereich der *Textur* auf jedes Texel wirkt). In Abb. 4.2 ist ein Vergleich von diffusen Objekten mit den 2D-Filter und dem sphärischen Filter gerendert, zu sehen. Obwohl der sphärische Filter physikalisch korrekter ist, sind die optischen Unterschiede minimal. Da die gegenwärtige Implementierung des sphärischen Filters ausserdem sehr aufwändig ist, wurde in der letzten Version der Applikation nur der 2D-Filter verwendet.

(c) 2D Filter (d) sphärischer Filter

Abbildung 4.5: In der linken Spalte ist die mit 2D-Filter bearbeitete Textur zu sehen, in der rechten Spalte die mit sphärischem Filter bearbeitete. Darunter sind die Ergebnisse des Renderings zu sehen. Es sind zwar Unterschiede in der Beleuchtung der Objekte vorhanden, diese ändern aber nichts an der visuellen Qualität.

Bestehende Probleme In der Umwandlung von Sphere Maps in Cube Maps befindet sich ein Problem, das sich als ein Pixel breiter Streifen am Rand jeder Cube Map Seite auswirkt. Dies ist insbesondere bei diffusen Texturen störend, die in geringerer Auflösung gespeichert werden. Die Auswirkungen sind z. B. in Abb. 4.4 oder in Abb 4.2 zu sehen.

Performance Für den Rechenaufwand der diffusen Beleuchtung gilt ähnliches wie für die der Reflexionen. Zusätzliche Rechenleistung wird durch das Erstellen der Irradiance Map benötigt. Im Fall des 2D-Filters besteht diese aus zwei Render-To-Texture Vorgänge mit 4-8 Texturzugriffen pro Fragment. In der gegenwärtigen Implementierung des sphärischen Filters ist es (wie in Kapitel 3.4.1 erwähnt), notwendig, eine stark tesselierte Kugel insgesamt zwischen 9 und 15 mal zur rendern, wodurch ein beträchtlicher Performanceeinbruch zustande kommt (ca. 20 Millisekunden, um eine Textur zu filtern).

4.3 Schatten

Die Darstellung von Schatten war mit Abstand die größte Herausforderung dieser Applikation. Der Grund dafür ist, dass für die Berechnung von Schatten in der Computergrafik eindeutige Positionen von Lichtquellen erforderlich sind.

Der erste Versuch, Schatten darzustellen, verwendet keine Lichtquellenextrahierung, sondern manuelle Einstellung der Position und Härte eines Schattens. Die Ergebnisse sind durchaus ansehnlich, solange sich nur eine starke Lichtquelle in der Szene befindet und diese natürlich nicht bewegt wird. Das Ergebnis ist in Abb. 4.6 zu sehen.

Abbildung 4.6: Wenn sich die Bedingungen nicht ändern, können auch mit der einfachsten Schattentechnik gute Ergebnisse erzielt werden.

Der nächste Schritt stellt die automatische Extrahierung der Lichquellenpositionen dar, wie in Abschnitt 2.5.2 beschrieben. Die verwendete Technik arbeitete in der Praxis nicht sehr robust, oft werden helle Objekte in der Umgebung, die jedoch nicht hell genug sind, um Schatten zu verursachen, als Lichtquellen erkannt. Die Gründe hierfür liegen im beschränkten Kontrastumfang der verwendeten Kameras und sind ebenfalls in Abschnitt 2.5.2 beschrieben.

Im dritten Versuch zur Lösung des Schattenproblems wird eine große Anzahl an Schatten gerendert, deren Intensität von der Helligkeit der Umgebung abhängt. Damit muss keine Analyse der Environment Map durchgeführt werden.

Wie in Abb. 4.7(b) zu sehen, sind die entstehenden Schatten sehr weich und eignen sich sehr gut, um den Schattenwurf, der in Umgebungen mit viel indirektem Licht auftritt, zu simulieren. Wenn die Intensität nicht variiert wird, werden Bereiche, die start verdeckt sind, stark abgedunkelt, es entsteht also eine Art „Ambient Occlusion“ [Hay02].

Probleme treten bei sehr kleinen Lichtquellen auf, da diese in der Realität einen sehr deutlichen Schatten werfen, der durch diese Schattentechnik verloren geht.

(a) 16 Schatten (b) 64 Schatten

Abbildung 4.7: Mit 16 Einzelschatten sind noch deutliche Ränder zu erkennen.

In Abb. 4.3 ist zwei Mal die selbe Szene zu sehen, einmal mit 64 Schatten und einmal mit 16 Schatten gerendert. In der Version mit 16 Schatten sind die einzelnen Schatten noch sehr deutlich zu sehen, mit 64 Schatten verschmelzen sie zu einer weichen Penumbra.

Performance Die erste Methode der Schattendarstellung ist sehr schnell, es muss keine Lichtquellenextrahierung durchgeführt und nur ein Schatten gerendert werden. Die Bildrate ist durch die Bildrate der Kamera auf 60 Bilder/Sekunde begrenzt, selbst wenn bildqualitätsverbessernde Massnahmen wie Anti-Aliasing exzessiv durchgeführt werden.

Die Analyse der Lichtquellenpositionen in der zweiten Schattentechnik ist in ihrer momentanen Version relativ aufwändig, v.a. die Implementierung des Verkleinerungsfilters ist sehr langsam (ca. 10 msec um aus einem Bild Lichtquellen zu extrahieren).

Mit Abstand am aufwändigsten ist die letzte Methode der Schattenberechnung. In Tabelle 4.1 ist eine Auflistung der Bildraten mit unterschiedlichen Parametern zu sehen. Je nach Anzahl der gerenderten Lichtquellen sinkt die Bildrate auf bis zu 12 Bilder/sec auf dem Testsystem. In Tabelle 4.1

Auflösung	**Anzahl an Schatten**		
unveränderte Einstellungen (60 Elemente)			
	0	16	64
640*480	41fps	28fps	14fps
1280*1024	41fps	25fps	12fps
verringerte Environment Map Auflösung ($32 \cdot 32$)			
640*480	48fps	28fps	14fps
1280*1024	48fps	25fps	12fps
verringerte Shadow Map Auflösung ($32 \cdot 32$)			
640*480	48fps	28fps	14fps
1280*1024	48fps	25fps	12fps
Nur ein Element (1000 Vertices)			
640*480	56fps	53fps	26fps
1280*1024	56fps	53fps	26fps

Tabelle 4.1: Bildraten der Applikation.

sind einige Parameter zu sehen, die auf ihre Auswirkung auf die Bildrate getestet wurden. Neben der Anzahl an verwendeten Lichtern wirkte sich v.a. die Anzahl an gerenderten Elementen auf die Bildrate aus. In der normalen Version wurden 60 Elemente (in 3 Objekten) mit insgesamt ca. 40000 Vertices gerendert, in einer vereinfachten Version wurde 1 Element mit ca. 1000 Vertices gerendert. Die erste Vermutung war, dass der Grund dafür der relativ lange Vertex Shader für die Berechnung der Schatten (9 Matrixmultiplikationen) ist. Da sich bei testweiser Verkürzung des Shaders auf eine Multiplikation an der Bildrate nichts änderte, liegt der Grund dafür wahrscheinlich an der ineffizienten Übergabe der Vertexdaten (die noch immer mit Vertex Arrays anstatt von VBOs oder Display Listen bewerkstelligt wird) oder von Shader-parametern an die GPU liegt.

Wenig Einfluss auf die Bildrate hat hingegen die Größe des Renderfensters und damit der berechneten Fragments, obwohl die Berechnung der Schatten einen einigermaßen langen Fragment Shader erfordert. Weitere Parameter die keinen Einfluss auf die Bildrate hatten, waren die Größe der Tiefentexturen, die Größe der Sphere- und Cube Maps sowie aktivierte oder deaktivierte Unschärfefilter.

Kapitel 5

Ausblick und Resümee

5.1 Zusammenfassung

In dieser Arbeit wurden einige Ansätze vorgestellt, um die Realtätstreue von virtuellen Objekten in Augmented Reality Szenen mit Image Based Lighting zu verbessern. Die vorgestellten Techniken verwenden entweder das Bild einer Spiegelkugel oder einer Fischaugenkamera, um Information über die reale Umgebung zu bekommen. Aus diesen Bildern werden Environment Maps für diffuse und glänzende Objekte generiert.

Neben der Darstellung von Beleuchtung wurde ein großes Augenmerk der Arbeit auf Schatten gelegt. Unterschiedliche Techniken wurden vorgestellt, die entweder versuchen, Positionen realer Lichtquellen zu extrahieren oder mit einer großen Anzahl an virtuellen Lichtquellen die globale Beleuchtungssituation anzunähern.

Auch wenn die Darstellung von Schatten noch viele Verbesserungsmöglichkeiten bietet, sind die dargestellten Techniken doch in der Lage, die Darstellung virtueller Objekte besser an reale Bedingungen anzupassen.

5.2 Verbesserungen

5.2.1 Allgemein

Sowohl bei Spiegelkugeln als auch bei Fischaugenobjektiven gibt es einen mehr oder weniger großen „toten Winkel“, also einen Bereich der Umgebung, aus dem keine Information vorhanden ist. Man könnte die Spiegelkugel mit 2 gegenüberliegenden Kameras abfilmen und eine kombinierte Environment Map aus beiden Kameras erzeugen, in der die schlecht aufgelösten Bereiche der einen Kamera durch die andere Kamera kompensiert werden.

5.2.2 Glänzende Oberflächen

In dieser Applikation wurden nur metallisch spiegelnde Oberflächen gerendert. Um glänzende Oberflächen wie Plastik, in deren Reflexion nur helle Lichtquellen sichtbar sind, rendern zu können, müssten die Environment Maps für Glanzlichter nachbearbeitet werden. Ausserdem zeigen die meisten Materialien einen *Fresnel* Effekt (siehe [FK03, Kap. 7.4] bzw. [Len04, Kap. 6.9.3], d. h. die Glanzstärke hängt vom Betrachterwinkel ab. Komplexere Materialien können durch Bearbeiten der Environment Map mit BRDFs erreicht werden (siehe [Fer04, Kap. 18]).

Bei all diesen Vorgängen entsteht jedoch das Problem, dass das gerenderte Bild „fahl“ wirkt, da helle Glanzpunkte durch Faltung und Abdunklung verloren gehen. Eine Lösung dieses Problems könnten Bilder mit einem höheren Konstrastumfang und einem höheren Dynamikbereich sein, die allgemein als HDR-Bilder (High Dynamic Range) bezeichnet werden.

5.2.3 Schatten

Wie bereits in Kapitel in Abschnitt 2.5.2 erwähnt, scheitert die automatische Extrahierung von Lichtquellenpositionen daran, dass übliche Kameras den Kontrastumfang realer Szenen nicht darstellen können. Auch hier bietet sich an, die Lichtquellen aus einem Bild mit höherem Kontrastumfang zu extrahieren. Solch ein Bild kann z. B. aus mehreren Bildern generiert werden, die mit unterschiedlichen Belichtungseinstellungen aufgenommen wurden (siehe Debevec [Deb02]. Um auf Veränderungen der Umgebung zu reagieren, muss dieses Bild jedoch relativ oft, im Idealfall öfter als ein Mal pro Sekunde erzeugt werden. Dafür ist eine Kamera notwendig, deren Belichtungseinstellungen programmatisch gesteuert werden können. Auch bei der Schattendarstellung mit vielen Lichtquellen könnte eine HDR-Environment Map Vorteile bieten, da dadurch stärkere Lichtquellen intensivere Schatten werfen würden.

Die Ermittelung der Intensität und Farbe von Schatten könnte besser gelöst werden, indem die Stärke der einzelnen Schatten von der Grundhelligkeit und -farbe der gesamten Szene abhängig gemacht wird. Momentan sind in hellen Szenen Schatten sehr dunkel und in dunklen Szenen fast unsichtbar.

Die Berechnung der Schatten könnte durch eine Form von *ambient Occlusion* ersetzt werden. Eine echtzeitfähige Version, das zu realisieren, ist in [Pha05, Kap. 14] beschrieben.

Anhang A

Shader Sourcecode

Hier wind die verwendeten Vertex- und Fragment Shader in alphabetischer Reihenfolge angeführt.

A.1 BlurEffect

Vertex Shader

```
struct vert2frag {
  float4 position   : POSITION;
  float2 blurCoord1 : TEXCOORD0;
  float2 blurCoord2 : TEXCOORD1;
  float2 blurCoord3 : TEXCOORD2;
  float2 blurCoord4 : TEXCOORD3;
};

vert2frag vp_blur(
  // position of vertex in object space
  float4 position : POSITION,
  // original texture coordinates of vertex
  float2 texCoord: TEXCOORD0,
  // distance between 2 texel in map in texture coord
  uniform float2 texCoordStep
){

  vert2frag OUT;
  OUT.position = position;
  // calculate displaced texture coordinates
  OUT.blurCoord1 = texCoord +     texCoordStep;
  OUT.blurCoord2 = texCoord + 2 * texCoordStep;
  OUT.blurCoord3 = texCoord + 3 * texCoordStep;
  OUT.blurCoord4 = texCoord + 4 * texCoordStep;
  return OUT;
}
```

Fragment Shader

```
void fp_blur(float4 position : POSITION,
             float2 blurCoord1  : TEXCOORD0,
             float2 blurCoord2  : TEXCOORD1,
             float2 blurCoord3  : TEXCOORD2,
             float2 blurCoord4  : TEXCOORD3,
             uniform sampler2D inTexture,

             out float4 outColor : COLOR
) {

  // look up the texture at 4 positions
  float4 color0 = tex2D(inTexture, blurCoord1);
  float4 color1 = tex2D(inTexture, blurCoord2);
  float4 color2 = tex2D(inTexture, blurCoord3);
  float4 color3 = tex2D(inTexture, blurCoord4);
  // take the average of the four texture lookups
  outColor = (color0 + color1 + color2 + color3) * 0.25;
}
```

A.2 SphereToCubeMap

Vertex Shader

```
struct vert2frag {
  float4 position : POSITION;
  float4 texCoord : TEXCOORD0;
};

vert2frag vp_sphereToCubeMap(
  // vertex position in object space
  float4 position : POSITION ,
  // vector from cube origin to vertex
  float3 cubeVector : TEXCOORD0 ,
  // transformation matrix from sphere map to world space
  uniform float4x4 texCoordTrafo
){

    vert2frag OUT;
    OUT.position = position;
    // multiply the texture coordinates with the
    // matrix from Sphere Map to Cube Map space
    OUT.cubeVector = mul(texCoordTrafo,
                         float4(texCoord, 1.0f));
    return OUT;
}
```

Fragment Shader

```
void fp_sphereToCubeMap( float4 position,
  // the vector to look up in the sphere map
  float4 cubeVector : TEXCOORD0,

  // 2D texture with sphere map
  uniform sampler2D sphereTex ,
  // determines how much of the environment is visible
  uniform float wrapValue
  ) {

  // normalize interpolated view-vector
  float3 cubeVectorNorm = normalize( cubeVector);

  // create the vector from the sphere to the eye
  float3 view = float3(0.0, 0.0, wrapValue);

  // create normalized halfway vector between
  // view and cubeVector
  float3 newNormal = normalize(view + cubeVectorNorm);

  // check if newNormal points to a valid texel
  if( newNormal.z < 0.0) {
    newNormal = normalize(float3(newNormal.xy, 0.0));
  }
  // look up spheremap texture at correct position
  outColor = tex2D(sphereTex, newNormal.xy * 0.5 + 0.5);

}
```

A.3 CubeMapLightingWithShadows

Vertex Shader

```
struct vert2frag {
   // position in screen space
   float4 position     : POSITION;
   // texture Coordinates for diffuse texture lookup
   float3 texCoordDiff : TEXCOORD0;
   // texture Coordinates for specular texture lookup
   float3 texCoordSpec : TEXCOORD1;
   // texture Coordinates for decal texture lookup
   float2 texCoordDec  : TEXCOORD2;
};

vert2frag vp_cubeMapDiffSpecShadow(
  // position in object space
  float4 position : POSITION,
```

```
  // texCoord of decal texture
  float4 texCoord : TEXCOORD0,
  // normal in object space
  float4 normal   : NORMAL,

// object to screen space
uniform float4x4 modelViewProj,
// view to object space
uniform float4x4 modelViewInv,
// obj to world inv trans
uniform float4x4 modelToWorldIT,
// object to world
uniform float4x4 modelToWorld)

{
    vert2frag OUT;

  // lighting part
  // diffuse
  float3 normalWSpace = mul(modelToWorldIT, normal).xyz;
  normalize(normalWSpace);
  OUT.texCoordDiff  = normalWSpace;

  // specular
  float4 camPosMSpace = mul(modelViewInv,
                            float4(0,0,0,1));
  float4 viewMSpace = camPosMSpace - position;
  float4 viewWSpace = mul(modelToWorld, viewMSpace);
  OUT.texCoordSpec = reflect(viewWSpace.xyz,
                                  normalWSpace);

  // decal texture coordinates
  OUT.texCoordDec = texCoord;

  // calculate the transformed position
  OUT.position = mul(modelViewProj, position);

  // put the transformed position to the FS too
  OUT.texCoordScreen = OUT.position;

  return OUT;
}
```

Fragment Shader

```
void fp_cubeMapDiffSpec(
  float4 position     : POSITION,
  float3 texCoordDiff : TEXCOORD0,
  float3 texCoordSpec : TEXCOORD1,
  float2 texCoordDec : TEXCOORD2,
```

```
  // fragment position in screen coordinates
  float4 texCoordScreen: TEXCOORD3,
  out float4 outColor : COLOR,

  // decal texture of object
  uniform sampler2D   decalMap,
  // Irradiance Map as cube map
  uniform samplerCUBE diffMap,
  // specular map as cube map
  uniform samplerCUBE specMap
) {

  // look up the diffuse, specular and decal color
  float4 diffColor  = texCUBE(diffMap, texCoordDiff.xyz);
  float4 specColor  = texCUBE(specMap, texCoordSpec.xyz);
  float4 decalColor = tex2D(decalMap,  texCoordShin.xy);

  // look up the shadow value
  float4 shadowColor;
  shadowColor = tex2D(shadowTex,
      (texCoordScreen.xy/texCoordScreen.w) * 0.5 + 0.5);
  // multiply the diffuse color with the inv shadowColor
  diffuseColor *= 1.0 - shadowColor;

  // calculate the lighting color as the linear inter-
  // polation between the specular color and the diffuse
  // color, weighted with the alpha of the decal color
  float4 reflectColor = lerp(specColor, diffColor,
                             decalColor.a);
  // multiply the lighting color with the material color
  outColor.rgb = reflectColor.rgb * decalColor.rgb;
}
```

A.4 DepthOnlyEffect

Vertex Shader

```
struct vert2frag {
  float4 position : POSITION;
  float4 coord    : TEXCOORD0;
};

vert2frag vp_depthOnly(
  float4 position : POSITION,
  uniform float4x4 modelViewProj
){

  vert2frag OUT;
  // transform the position
```

```
  OUT.position = mul(modelViewProj, position);
  // to print out the written depth as color for
  // debugging comment this in:
  // ### OUT.coord = OUT.position; ###

  return OUT;
}
```

Fragment Shader

```
void fp_depthOnly(
  float4 position : POSITION,
  // screen position to print out depth as color
  float4 coord    : TEXCOORD0,

  out float4 outColor : COLOR
) {

  // this fragment shader is only bound for debugging
  // usually color writing is disabled during depth only
  // rendering

  // to print out the written depth as color for
  // debugging comment this in:
  // ### outColor = coord.z/coord.w; ###
}
```

A.5 ShadowOnlyEffect

Vertex Shader

```
struct vert2frag {
  float4 position : POSITION;
  // the position in screen coordinates
  float4 texCoordScreen : TEXCOORD0;
};

vert2frag vp_shadowOnly(
  float4 position : POSITION,
  uniform float4x4 modelViewProj
  ){

  vert2frag OUT;
  // transform the position
  OUT.position = mul(modelViewProj, position);
  // pass the transformed position to the fragment shader
  OUT.texCoordScreen = OUT.position;

  return OUT;
}
```

Fragment Shader

```
void fp_shadowOnly(
  float4 position  : POSITION,
  float4 texCoordScreen: TEXCOORD0,
  out float4 outColor : COLOR,

  // the texture with the shadow image
  uniform sampler2D shadowTex,
  // the texture with the image of the background scene
  uniform sampler2D backGroundTex
) {

  // set outColor as the inverted shadow image
  float4 shadowValue = 1.0 - tex2D(shadowTex,
       (texCoordScreen.xy/texCoordScreen.w) * 0.5 + 0.5);

  // multiply the shadow value with the background image
  // to cast shadows on the real background
  outColor = shadowValue * tex2D(backGroundTex,
       (texCoordScreen.xy/texCoordScreen.w) * 0.5 + 0.5);
}
```

A.6 ManyShadowsEffect

Vertex Shader

```
struct vert2frag {
  float4 position     : POSITION;
  float4 shadowCoord0 : TEXCOORD0;
  float4 shadowCoord1 : TEXCOORD1;
  float4 shadowCoord2 : TEXCOORD2;
  float4 shadowCoord3 : TEXCOORD3;
  float4 shadowCoord4 : TEXCOORD4;
  float4 shadowCoord5 : TEXCOORD5;
  float4 shadowCoord6 : TEXCOORD6;
  float4 shadowCoord7 : TEXCOORD7;
};

vert2frag vp_manyShadowsIBL(
  // the position of the vertex in object space
  float4 position : POSITION,
  // the mvp matrix of the camera
  uniform float4x4 modelViewProj,
  // the mvp matrix of light 0 to 7
  uniform float4x4 shadowMatrix0,
  uniform float4x4 shadowMatrix1,
  uniform float4x4 shadowMatrix2,
  uniform float4x4 shadowMatrix3,
```

```
  uniform float4x4 shadowMatrix4,
  uniform float4x4 shadowMatrix5,
  uniform float4x4 shadowMatrix6,
  uniform float4x4 shadowMatrix7
){

  vert2frag OUT;

        // transform with camera mvp matrix
  OUT.position = mul(modelViewProj, position);
  // transform with mvp matrices of all light sources
  OUT.shadowCoord0 = mul(shadowMatrix0, position);
  OUT.shadowCoord1 = mul(shadowMatrix1, position);
  OUT.shadowCoord2 = mul(shadowMatrix2, position);
  OUT.shadowCoord3 = mul(shadowMatrix3, position);
  OUT.shadowCoord4 = mul(shadowMatrix4, position);
  OUT.shadowCoord5 = mul(shadowMatrix5, position);
  OUT.shadowCoord6 = mul(shadowMatrix6, position);
  OUT.shadowCoord7 = mul(shadowMatrix7, position);
  return OUT;
}
```

Fragment Shader

```
void fp_manyShadowsIBL(
  float4 position : POSITION,
  float4 shadowCoord0 : TEXCOORD0,
  float4 shadowCoord1 : TEXCOORD1,
  float4 shadowCoord2 : TEXCOORD2,
  float4 shadowCoord3 : TEXCOORD3,
  float4 shadowCoord4 : TEXCOORD4,
  float4 shadowCoord5 : TEXCOORD5,
  float4 shadowCoord6 : TEXCOORD6,
  float4 shadowCoord7 : TEXCOORD7,

  out float4 outColor : COLOR,

  uniform sampler2D shadowMap0,
  uniform sampler2D shadowMap1,
  uniform sampler2D shadowMap2,
  uniform sampler2D shadowMap3,
  uniform sampler2D shadowMap4,
  uniform sampler2D shadowMap5,
  uniform sampler2D shadowMap6,
  uniform sampler2D shadowMap7,

  uniform float4 envCoord0,
  uniform float4 envCoord1,
  uniform float4 envCoord2,
  uniform float4 envCoord3,
```

```
  uniform float4 envCoord4,
  uniform float4 envCoord5,
  uniform float4 envCoord6,
  uniform float4 envCoord7,

  uniform samplerCUBE envMap,

  uniform float nrLights
) {

  outColor = float4(0.0f, 0.0f, 0.0f, 1.0f);

  outColor += (1.0 - tex2Dproj(shadowMap0, shadowCoord0))
              * texCUBE(envMap, lightPos0.xyz);
  outColor += (1.0 - tex2Dproj(shadowMap1, shadowCoord1))
              * texCUBE(envMap, lightPos1.xyz);
  outColor += (1.0 - tex2Dproj(shadowMap2, shadowCoord2))
              * texCUBE(envMap, lightPos2.xyz);
  outColor += (1.0 - tex2Dproj(shadowMap3, shadowCoord3))
              * texCUBE(envMap, lightPos3.xyz);
  outColor += (1.0 - tex2Dproj(shadowMap4, shadowCoord4))
              * texCUBE(envMap, lightPos4.xyz);
  outColor += (1.0 - tex2Dproj(shadowMap5, shadowCoord5))
              * texCUBE(envMap, lightPos5.xyz);
  outColor += (1.0 - tex2Dproj(shadowMap6, shadowCoord6))
              * texCUBE(envMap, lightPos0.xyz);
  outColor += (1.0 - tex2Dproj(shadowMap7, shadowCoord7))
              * texCUBE(envMap, lightPos7.xyz);

  // nrLights / 8 is the number of passes
  outColor /= 8.0;
  outColor *= 8.0 / nrLights;
}
```

A.7 SphereMapLightingEffectShadows

Vertex Shader

```

struct vert2frag {
  // position in screen space
  float4 position   : POSITION;
  // texture Coordinates for diffuse texture lookup
  float2 texCoordDiff : TEXCOORD0;
  // texture Coordinates for specular texture lookup
  float2 texCoordSpec : TEXCOORD1;
  // texture Coordinates for decal texture lookup
  float2 texCoordDec  : TEXCOORD2;
  // texture coordinates for shadow map lookup
```

```
  float3 texCoordShadow : TEXCOORD3;
};

vert2frag vp_sphereMapDiffSpec(
  float4 position : POSITION,  // position in obj space
  float4 texCoord : TEXCOORD0, // texCoord of decal tex
  float4 normal   : NORMAL,    // normal in obj space

  uniform float4x4 modelViewPrj, // obj to screen space
  uniform float4x4 modelView     // obj to view space
  uniform float4x4 modelViewIT,  // inverse transposed mv
  uniform float4x4 lightMatrix   // matrix of light
  ){

  vert2frag OUT;
  OUT.position = mul(modelViewPrj, position);

  float3 posVSpace;
  posVSpace = normalize(mul(modelView, position).xyz);
  float3 viewVSpace = -posVSpace;
  float3 normalVSpace;
  normalVSpace = normalize(mul(modelViewIT, normal).xyz);

  // for diffuse lighting
  float3 texCoordDiff;
  texCoordDiff = normalize(normalVSpace + viewVSpace);
  OUT.texCoordDiff = float3(texCoordDiff * 0.5 + 0.5);

  // for specular lighting
  OUT.texCoordSpec = float3(normalVSpace * 0.5 + 0.5);

  OUT.texCoordDec = texCoord;

  // shadow part
  OUT.texCoordShadow = mul(lightMatrix, position);

  return OUT;
}
```

Fragment Shader

```
void fp_sphereMapDiffSpec(
  float4 position       : POSITION,
  float4 texCoordDiff   : TEXCOORD0,
  float4 texCoordSpec   : TEXCOORD1,
  float4 texCoordDec    : TEXCOORD2,
  float4 texCoordShadow : TEXCOORD3,

  out float4 outColor : COLOR,
```

```
  uniform sampler2D diffuseMap    : TEXUNIT0,
  uniform sampler2D specularMap   : TEXUNIT1,
  uniform sampler2D decalMap      : TEXUNIT2,
  uniform sampler2D shadowTexture : TEXUNIT3
  ) {

  // look up the diffuse, specular and decal color
  float4 diffCol  = tex2D(diffuseMap,  texCoordDiff.xy);
  float4 specCol  = tex2D(specularMap, texCoordSpec.xy);
  float4 decalCol = tex2D(decalMap,    texCoordDec.xy);

  // look up the shadow intentsity
  float4 shadowValue =
      tex2Dproj(shadowMap, texCoordShadow) * 0.25 + 0.75;
  // multiply the diffuse color with the shadow intensity
  diffColor *= shadowValue;

  // calculate the lighting color as linear interpolation
  // between diffuse and specular color
  float4 lightingColor =
         lerp(specCol, diffCol, decalColor.a);

  // multiply the lighting color with the material color
  outColor.rgb = lightingColor.rgb * decalColor.rgb;
}
```

Anhang B

Inhalt der CD-ROM

File System: Joliet

B.1 Diplomarbeit

Pfad: /Diplomarbeit/

da_peter_supan.pdf . . .	Diplomarbeit (PDF-File)
latex/	Latex Sourcecode, verwendete Bilder und Graphiken als eps-Dateien

B.2 Implementierung

Pfad: /Implementierung/

bin/ARIBLshadows.exe	ausführbare Applikation
bin/	zusätzliche DLLs
data/	verwendete Texturen und Modelle
dependencies/	verwendete Bibliotheken
include/	Headerdateien
shader/	cg-Dateien
source/	Quelltext
visualStudio/	Visual Studio 2005 Solution - ARIBLshadows.sln

B.3 Referenzen

Pfad: /Referenzen/

pdf/pcf.pdf	Original Paper of Percentage Closer Filtering for Shadow Maps (from 1987)

pdf/gi_jacobs.pdf	Global Illumination Techniques used in feature films
pdf/illumap.pdf	The use of Image Based Lighting to render diffuse surfaces (from 1984)
pdf/Gibson_shadows.pdf	Using many Shadow Map Lights and phantome objects to render photorealistic objects in Augemented Reality
pdf/blinn-newell.pdf . .	The original Environment Mapping Paper (from 1974)
pdf/LightDome.pdf . .	Using a Light Dome to render realistic outdoor lighting
pdf/Parth-062004.pdf .	*Making of* of a short film using Image Based Lighting
pdf/ibl-tutorial.pdf . . .	Introduction into Image Based Lighting
pdf/MedianCut.pdf . .	An Algorithm to extract positions of Light Sources out of an Envrionment Map
pdf/NVDemos7800.pdf	A Detailed overview over the techniques used in the demos of nvidias Geforce 7800 graphics cards
pdf/SAT.pdf	A short paper about the technique of summed area tables on the gpu
pdf/SATsketch.pdf . . .	A presentation about how to perform summed area tables on the gpu
pdf/smoothie.pdf . . .	A shadow algorithm that renders realtistic appearing soft shadows efficiently

B.4 Bilder und Videos

Pfad: /Screenshots/ Screenshots der Applikation

Pfad: /Videos/ Videos der Applikation

Literaturverzeichnis

[AMH02] Akenine-Möller, Tomas und Eric Heines: *Real-Time Rendering*. A K Peters, 2. Auflage, 2002.

[Azu04] Azuma, Ronald: *Overview of augmented reality*. In: *GRAPH '04: Proceedings of the conference on SIGGRAPH 2004 course notes*, Seite 26, New York, NY, USA, 2004. ACM Press.

[BB05] Burger, Wilhelm und Mark J. Burge: *Digitale Bildverarbeitung*. Springer-Verlag, 1. Auflage, 2005.

[BN76] Blinn, James und M.E. Newell: *Texture and Reflection in Computer Generated Images*. In: *Communications of the ACM 19(10)*, Seiten 542–547, October 1976.

[CD03] Chan, Eric und Frédo Durand: *Rendering Fake Soft Shadows with Smoothies*. In: *Proceedings of the Eurographics Symposium on Rendering*, Seiten 208–218. Eurographics Association, 2003.

[Deb98] Debevec, Paul: *Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography*. In: *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, Seiten 189–198, New York, NY, USA, 1998. ACM Press.

[Deb02] Debevec, Paul: *Image-Based Lighting*. IEEE Computer Graphics and Applications, 22:26–34, March–April 2002.

[Deb05] Debevec, Paul: *A Median Cut Algorithm for Light Probe Sampling*. Technischer Bericht 67, USC Institute for Creative Technologies, August 2005.

[Dra03] Drab, Stephan: *Echtzeitrendering von Schattenwurf in Augmented- und Mixed-Reality-Anwendungen*. Diplomarbeit, Fachhochschule Hagenberg, Medientechnik und -design, Hagenberg, Austria, Juli 2003.

[DTG+04] Debevec, Paul, Chris Tchou, Andrew Gardner, Tim Hawkins, Charis Poullis, Jessi Stumpfel, Andrew Jones, Nathaniel Yun, Per Einarsson, Therese Lundgren, Marcos Fajardo und Phillipe Martinez: *Estimationg Surface Reflectance Properties of a Complex Scene under Captured Natural Illumination.* Technischer Bericht ICT-TR-06.2004, University of Southern California Institute for Creative Technologies Graphics Laboratory, June 2004.

[Fer04] Fernando, Randima: *GPU Gems.* Addison-Wesley, 2004.

[FK03] Fernando, Randima und Mark J. Kilgard: *The Cg Tutorial.* Addison-Wesley, 2003.

[GHH03] Gibson, Simon, Toby Howard und Roger Hubbold: *Rapid Shadow Generation in Real-World Lighting Environments.* In: *Proceedings of the Eurographics Symposium on Rendering*, June 2003.

[Hay02] Hayden, Landis: *Production-Ready Global Illumination.* In: *Course 16 notes, SIGGRAPH 2002*, 2002.

[HSC+05] Hensley, Justin, Thorsten Scheuermann, Greg Coombe, Anselmo Lastra und Montek Singh: *Fast Summed-Area Table Generation and Its Applications.* Technischer Bericht, University of North Carolina at Chapel Hill and ATI Research, June 2005.

[KB99] Kato, Hirokazu und Mark Billinghurst: *Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System.* In: *IWAR 99: Proceedings of the 2nd International Workshop on Augmented Reality*, October 1999.

[KOI05] Kakuta, Tetsuya, Takeshi Oishi und Katsushi Ikeuchi: *Shading and Shadowing of Architecture in Mixed Reality.* In: *Proceedings of ISMAR 2005*, 2005.

[KS05] Karlsson, J. und M. Selegard: *Rendering Realistic Augmented Objects Using an Image Based Lighting Approach.* Diplomarbeit, Linköpings Universitet, Department of Science and Technology, Sweden, June 2005.

[Lan05] Langsdorf, Bea: *GPU Programming Exposed: The Naked Truth Behind NVIDIA's Demos.* Technischer Bericht, NVIDIA Corporation, August 2005.

[Len04] Lengyel, Eric: *Mathematics for 3D Game Programming and Computer Graphics.* Charles River Media, 2. Auflage, 2004.

[MH84] Miller, Gene S. und C. Robert Hoffman: *Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments*. In: *Course notes for Advanced Computer Graphics Animation, SIGGRAPH 84*, 1984.

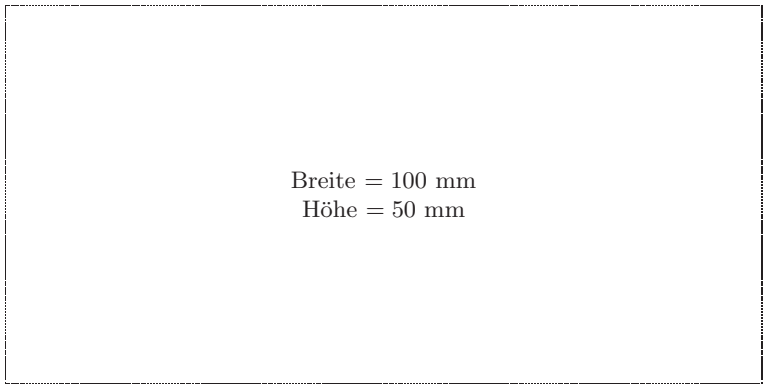
[Pha05] Pharr, Matt: *GPU Gems 2*. Addison-Wesley, 2005.

[Pho75] Phong, Bui Tuong: *Illumination for computer generated pictures*. Commun. ACM, 18(6):311–317, 1975.

[RSC87] Reeves, William T., David H. Salesin und Robert L. Cook: *Rendering antialiased shadows with depth maps*. In: *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, Seiten 283–291, New York, NY, USA, 1987. ACM Press.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —