Natürliche Beleuchtungsberechnung in virtuellen Szenen

Studienarbeit im Fach Informatik

vorgelegt von

Thomas Kemmer

geb. am 29. Oktober 1980 in Miltenberg

angefertigt am

Institut für Informatik
Lehrstuhl für Graphische Datenverarbeitung
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Manfred Ernst

Betreuender Hochschullehrer: Prof. Dr. Marc Stamminger

Beginn der Arbeit: 01. Januar 2005

Abgabe der Arbeit: 30. September 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Die Idee zum Projekt

Natürliche Beleuchtungsberechnung hoher Qualität ist bis heute eine große mathematische und rechenintensive Herausforderung. Die Entwicklung schnellerer Hardware ermöglicht es, mehr und mehr visuelle Effekte in immer weniger Zeit zu berechnen. In dieser Arbeit soll es darum gehen ein aufwändiges Beleuchtungsverfahren hardwarebeschleunigt zu implementieren. Eine abgewandelte Form dieser Art der Lichtberechnung gibt es bereits in aktuellen Ray-Tracing Systemen. Bekanntermaßen sind heutige Ray-Tracer unter vertretbarem Aufwand noch nicht schnell genug, um Echtzeit Rendering zu betreiben. Ich habe den Versuch unternommen wesentlich schneller zu einem guten Ergebnis zu gelangen, auch wenn es wegen einiger Limitierungen in der Rastergrafik nicht so perfekt sein kann wie das, was man mit einem Ray-Tracer erreicht. Nicht immer ist die höchste Güte das alleinige Ziel!

Mit Hilfe klassischer Algorithmen zur Beleuchtungsberechnung ist ein flexibles, kleines Renderingsystem entstanden, in das man beliebige Szenendateien und eine Environmentmap laden kann. Die Environmentmap bestimmt die Beleuchtungssituation für die aktuell anzuzeigende Szene. Ein Sampling-Verfahren wählt aus dieser Map besonders helle Stellen aus. Dies dient dazu, Positionen für Lichtquellen vorzugeben, die die Szenerie beleuchten. Jede der gewählten Lichtquellen strahlt die Szene an und trägt seinen Teil zur Gesamtbeleuchtung bei. Um den Realismusgrad und den dreidimensionalen Eindruck weiter zu verstärken, sind Schatten unbedingt nötig. Leider kann man diese in der Rastergrafik nicht auf direktem Wege berechnen. In der Geschichte der Rastergrafik wurden verschiedene Algorithmen zur harwarebeschleunigten Berechnung von Schatten entwickelt. Ich verwende in meiner Implementierung das Shadowmapping (s. 5.3). Für jede in der Szene vorkommende Lichtquelle wird eine Shadowmap generiert und anschließend damit ein Einzelbild gerendert. Das Endergebnisbild entsteht, indem in einem letzten Schritt alle Einzelbilder aufsummiert werden.

In dieser Arbeit soll es vor allem um natürliche Beleuchtung gehen. Vergleicht man die Helligkeit in einem künstlich erleuchteten Raum mit der an einem sonnigen Tag am Strand, stellt man zwangsläufig fest, dass die Sonne wohl tausendfach stärker ist, als die Deckenbeleuchtung. Die interaktive Computergrafik hat sich erst in den letzten Jahren mit diesem Thema auseinander gesetzt. Mit

der Einführung von Fließkommaberechnungen auf der Grafikkarte und programmierbaren Shadern wurde Rendering mit High Dynamic Range (HDR) möglich. HDR ist maßgeblich für einen natürlich wirkenden Gesamteindruck verantwortlich. Aus diesem Grunde verwende ich in meinem Projekt eine eigene HDR-Beleuchtungsberechnung mit anschließendem Tonemapping, das nötig ist um HDR-Bilder auf einem Low Dynamic Range Monitor anzeigen zu können.

1.2 Vorgehensweise und Ziele des Projektes

Folgende Voraussetzungen waren die Grundlage für das Rendersystem:

- API zum Anzeigen von OpenGL unter Linux (GLUI)
- Möglichkeit um Szenendateien einzulesen (.obj - File Parser)
- Einlesen von HDR-Radiance-Angularmaps
- Render to Texture Implementierung

Zunächst mussten die Grundlagen für ein Rendersystem mit OpenGL unter Linux geschaffen werden. Ich habe mich für eine einfache API mit dem Namen “GLUI” entschieden. Diese wird zwar schon mehrere Jahre nicht mehr weiterentwickelt, aber sie ist einfach und bringt die Funktionalität mit, die man für kleine Demoprojekte mit einfacher Benutzerschnittstelle benötigt. Als nächsten Schritt habe ich einen vorgegebenen Dateiparser für Szenendateien im Wavefront-OBJ-Format für meine Zwecke angepasst und um Shaderkompatibilität erweitert. OBJ-Dateien können praktsich von allen 3D-Modelling Programmen wie “3DSMax” oder “Blender3D” erzeugt werden, wodurch es ein Leichtes ist, Modelle für mein Programm anzupassen. Für eine natürliche Beleuchtung muss es möglich sein, eine Environmentmap einzulesen, die die Beleuchtungsssituation an einem bestimmten Punkt in der Natur repräsentiert. *Paul Debevec* hat einige solcher Aufnahmen gemacht und sie in *Greg Wards* Radiance-Dateiformat (.hdr) veröffentlicht. “Radiance” ist im Übrigen auch der Name für einen Raytracer von ihm, den er in seinem Buch [11] ausführlich vorstellt. Dieser ist Open Source und kann .hdr-Dateien einlesen. Ich habe es als hilfreich empfunden, mich genauer mit dem Dateiformat auseinander zu setzen und habe einen minimalen File Reader für HDR-Dateien geschrieben (s. Kapitel 2.6), der die gängigen Optionen des Formats unterstützt. Die nächste große Herausforderung war es, eine Möglichkeit zu schaffen, mit einer höheren Präzision als nur 8 Bit pro Farbkanal zu rendern. Dies gestaltete sich relativ einfach mit Hilfe eines Beispieles aus den Nvidia OpenGL Tutorials für Linux. Das wesentlich größere Problem war der Umgang mit den Nvidia OpenGL Headern.
Nachdem die Grundlagen geschaffen waren, war es möglich diese Ziele zu formulieren:

- Implementierung eines Importance-Sampling Algorithmus zur Auswahl von Lichtquellen
- Echtes High Dynamic Range Phong-Lighting
- Einsetzen von Shadowmaps zur Schattenberechnung
- Erstellung eines interaktiven Rendersystems mit progressiver Qualitätsverbesserung
- Benchmarks

Der erste Punkt bildet die Grundlage für die Beleuchtungsberechnung. Angenommen, die vorgegebene Umgebung ist eine Tagesszene mit klarem Himmel, dann ist es das Ziel, möglichst viele Lichtquellen an der Stelle zu setzen, an der die Sonne steht, da sie ja hautpverantwortlich ist für Licht und Schatten in der Szene. Zu diesem Zweck habe ich einen einfachen Importance-Sampling Algorithmus implementiert. Ich habe versucht, das Ganze modular zu gestalten, damit man das Programm an dieser Stelle möglichst einfach um weitere Algorithmen erweitern kann. Nachdem die Auswahl und Erzeugung der HDR-Lichtquellen funktioniert hat, habe ich mich dem nächsten Teil zugewandt: der Lichtberechnung. Mit der Fixed-Function Pipeline von OpenGL war das naturgemäß nicht möglich, da sie einfaches Phong-Lighting durchführt und am Ende alle Werte geclampt werden und damit die HDR-Information verloren geht. Aus diesem Grund war es nötig, einen eigenen Vertexshader zu schreiben, der die Lichtberechnung nach meinen Vorstellungen erledigt. Danach sollen die fertig gerenderten Bilder in Fließkommatexturen gesichert und anschließend in einem weiteren Durchlauf miteinander verknüpft werden. Wegen des limitierten Grafikspeichers gelten hier jedoch einige Einschränkungen.
Der Einsatz von Schatten ist unabdingbar für ein realistisches Erscheinungsbild. Shadowmapping ist eines der Verfahren um Schatten zu berechnen, wenngleich es auch einige Schwächen hat, wie z.B. starke Aliasing Artefakte in schlecht aufgelösten Bereichen der Shadowmap. In diesem Kontext spielt das aber eine eher untergeordnete Rolle. In den ersten Tests der Implementierung habe ich bereits festgestellt, dass eine Shadowmap-Auflösung von 512 x 512 in den meisten Fällen ausreicht, da eine große Anzahl von fertigen Bildern miteinander kombiniert werden und die Treppenstufen ab einer bestimmten Zahl von Einzelbildern verschwinden. Einer der Vorteile dieses Rendersystems soll es sein, dass trotz hoher Qualität die interaktive Benutzereingabe nicht leiden soll. Wünschenswert ist es außerdem, dass sich das Bild bei Stillstand immer weiter verbessert, bis sich keine signifikanten Änderungen mehr ergeben. Diesen Punkt kann man nur empirisch bestimmen, da er stark von der verwendeten Environmentmap abhängt.

1.3 Previous Work

Eine Reihe von Publikationen hatte einen großen Einfluss auf meine Arbeit. An dieser Stelle möchte ich diejenigen vorstellen, die mir am wichtigsten erscheinen.

1.3.1 Recovering High Dynamic Range Radiance Maps from Photographs

In [3] präsentieren *Debevec* und *Malik* eine Methode die es ermöglicht, mit konventionellem Fotoequipment eine HDR-Environmentmap zu erzeugen. Grundlage für das Verfahren ist, dass man eine Szene mehrfach mit unterschiedlicher Belichtungszeit aufnehmen kann. Es wurde ein Algorithmus entwickelt, der die Einzelbilder zu einer HDR-Environmentmap verschmelzen kann, deren Farbwerte proportional zur tatsächlichen Helligkeit der realen Szene sind. Dieses Verfahren hat es ermöglicht, Environmentmaps mit vertretbarem Aufwand zu erzeugen, um sie für die Computergrafik nutzbar zu machen. *Paul Debevec* hat eine Gallerie mit solchen Environmentmaps im Internet [4] für jeden zugänglich gemacht. Sehr bemerkenswert ist, dass praktisch alle wissenschaftlichen Arbeiten über HDR die Environmentmaps aus dieser Gallerie verwenden.

1.3.2 Rendering Synthetic Objects into Real Scenes

Paul Debevec stellte zur SIGGRAPH 1998 in [5] ein Verfahren vor, das es erlaubt, synthetische Objekte in eine reale Szene zu rendern. Genauer gesagt bietes es die Möglichkeit ein neues Objekt in ein Foto oder in einen Film einzupassen, sodass für den Betrachter nicht das Gefühl der Fremdartigkeit entsteht. Voraussetzung dafür ist ein Verfahren zur globalen Beleuchtungsberechnung basierend auf einer HDR-Environmentmap. Dazu teilt er die Szene in drei Komponenten ein: die entfernte Szene, die lokale Szene und die synthetischen Objekte. Der Einfachheit halber nimmt man an, dass die synthetischen Objekte die entfernte Szene nicht beeinflussen können. Die lokale Szene jedoch erhält ein Reflektionsmodell, sodass sich die neuen Objekte darin spiegeln und einen Schatten darauf werfen können. Die neuen Objekte werden anschließend mit einem differentiellen Renderverfahren in das bestehende Bild eingebracht. Dank dem Einsatz von HDR-Rendering erzielt man mit diesem Verfahren sehr gute Ergebnisse.

1.3.3 Hardware Shadowmapping

Shadowmapping ist eines der Verfahren um den Schattenwurf einer Lichtquelle zu berechnen. *Cass Everitt, Ashu Rege* und *Cem Cebenoyan* von Nvidia beschreiben in [8] ein hardwarebeschleunigtes Verfahren, das ab Nvidia GeForce3 GPUs unterstützt wird. Obwohl das Schadowmapping auch einigen Limitierungen unterliegt, ist es dennoch ein sehr beliebtes Verfahren zur Schattenberechnung, da man keine zusätzliche Geometrie erzeugen muss, wie z.B. für *Stenciled Shadow Volumes*. Shadowmapping nutzt hardwarebeschleunigte Texturierung und die Verwendung eines Tiefenpuffers. Das Neue an dem vorgestellten Verfahren ist, dass der Vergleich der projizierten Shadowmap mit dem Tiefenwert des zu rendernden Fragments mit Hilfe von Register Combinern durchgeführt wird und somit auch hardwarebeschleunigt ist. Obwohl dieses Verfahren dank Vertex- und Pixelshadern wieder veraltet ist, gibt das Paper trotzdem einen guten Einblick in die Funktionsweise, Vorteile und Limitierungen des Shadowmapping.

1.3.4 Structured Importance Sampling of Environment Maps

In [1] stellen *Sameer Agarwal et al.* eine Samplingmethode für Environmentmaps vor. Sie erhalten damit deutlich bessere visuelle Ergebnisse als mit den üblichen Monte Carlo-Verfahren. Das Sampling und die Aufteilung der Environmentmap in einzelne Bereiche berücksichtigt die Beleuchtungsstärke und die erwartete Varianz auf Grund von Verdeckungen in der Szene. Ein hierarchischer Stratifizierungsalgorithmus ermöglicht das Aufteilen der Environmentmap in gleichmäßige Bereiche. Anschließend wird durch Präintegration dieser Bereiche ein schnellerer Lookup ermöglicht. Als wichtigstes Ergebnis wird die Reduktion der Samples um zwei Größenordnungen im Vergleich zum Monte Carlo-Sampling angegeben.

1.3.5 Infinite Area Light Source with Importance Sampling

Matt Pharr und *Greg Humphrys* nehmen in einem Paper [20], Bezug auf das *Structured Importance Sampling* [1] und stellen ein im Vergleich sehr einfaches Importance-Sampling für Environmentmaps vor, das in der Praxis ähnlich gute Ergebnisse bringt. Der verfolgte Ansatz ist der, dass man mit Hilfe

der Environmentmap eine Wahrscheinlichkeitsdichtefunktion aufstellt, durch die man helle Stellen mit größerer Wahrscheinlichkeit auswählt als dunkle. Die daraus resultierende Varianzreduktion ist enorm, verglichen mit einfachem Monte Carlo-Sampling. Weitere Vorteile dieser Methode sind die vergleichsweise geringe Vorberechnungszeit und der verschwindend geringe Rechenzeitverbrauch zur Ausführungszeit. Da ich dieses Verfahren in meiner Arbeit einsetze, wird es in aller Ausführlichkeit im Kapitel 4 erklärt.

1.4 Mögliche Einsatzbereiche

1.4.1 Preview

Das vorliegende Programm ist flexibel erweiterbar und könnte als Vorschau für Ray-Tracing-Systeme verwendet werden. Durch die objektorientierte Programmierung ist der Renderer unabhängig von der verwendeten Benutzerschnittstellenbibliothek (GLUI) einsetzbar; eine Portierung etwa nach QT oder GTK sollte einfach zu machen sein.

1.4.2 Bedeutung von natürlicher Beleuchtungsberechnung für die Industrie

Natürliche Beleuchtungsberechnung spielt bereits heute eine große Rolle für die Visualisierung von Prototypen. Der Bau eines Modells verschlingt Unsummen im Vergleich zur praktisch kostenlosen Rechenleistung moderner Computer. Deswegen wird Simulation ein immer wichtigerer Faktor bei der Produktentwicklung. Erstellte früher ein Produktentwickler auf dem Reißbrett eine Zeichnung, dauerte es lange bis er einen natürlichen Eindruck davon erhalten kann, nämlich genau dann, wenn seine Kollegen ein Modell in miniaturisierter Form gebaut haben. In der heutigen schnelllebigen Welt werden Innovationszyklen immer kürzer und darum liegt eine möglichst schnelle Visualisierung im Interesse eines Entwicklers und damit des ganzen Unternehmens. Somit sind Verfahren der natürlichen Beleuchtungsberechnung prädestiniert für den Einsatz als Produktvisualisierungssoftware, denn das Ziel muss sein möglichst früh einen Eindruck des fertigen Produktes zu erhalten, so wie es in Wirklichkeit aussieht. Das hilft Marktchancen abzuschätzen, teure Fehlentscheidungen zu vermeiden und kann schließlich zu einem größeren Erfolg des Unternehmens führen.

Kapitel 2

High Dynamic Range (HDR)

Dieses Kapitel ist der Erzeugung und Verwendung von High Dynamic Range Environmentmaps gewidmet. High Dynamic Range Rendering wird zunehmend in aktuellen Spielen eingesetzt, um die Szenerien deutlich natürlicher wirken zu lassen. Deswegen war es auch keine Frage, ob ich HDR in meiner Arbeit einsetzen würde oder nicht, sondern eine Voraussetzung! Dabei haben mich die Arbeiten von *Paul Debevec* sehr inspiriert, und natürlich bildeten seine, in diesem Bereich sehr bekannten Beispielmaps seiner Gallerie, die Grundlage für meine Beleuchtungsberechnung. Diese Environmentmaps bestimmen die Beleuchtungssituation an einer Stelle in einer realen Umgebung. Üblicherweise werden diese in der Rastergrafik zur Berechnung von Reflektionen verwendet. Details dazu befinden sich in Abschnitt 5.2. Zunächst einmal muss man sich aber die Frage stellen, um was es sich bei HDR überhaupt handelt und wieso es gerade in letzter Zeit so viel Zuspruch findet.

2.1 Was ist Dynamic Range?

Die reale Welt hat einen sehr großen Bereich (Dynamik) von möglichen Helligkeitswerten, z.B. ist das Licht der Sonne um ein Tausendfaches stärker als das einer Deckenleuchte und diese wiederum ist viel stärker als das Licht eines Sterns in einer klaren Nacht. Als physikalisches Maß, um die Abstrahlstärke eines Objektes zu messen, verwendet man die Leuchtdichte (engl. luminance):

Helligkeit des Himmels	
menschliche Wahrnehmungsgrenze	$3 * 10^{-6} \frac{cd}{m^2}$
Nachthimmel (bewölkt)	$100 * 10^{-6} \frac{cd}{m^2}$
Nachthimmel (klar)	$0,001 \frac{cd}{m^2}$
bedeckter Himmel	$2 * 10^{3} \frac{cd}{m^2}$
klarer Himmel	$8 * 10^{3} \frac{cd}{m^2}$

Tabelle 2.1: Leuchtdichte des Himmels

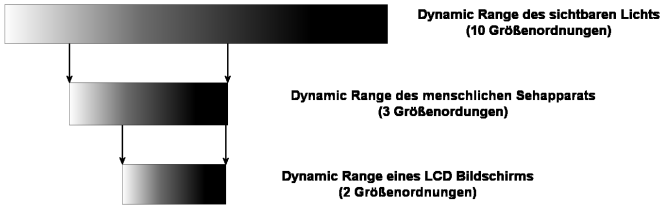
Helligkeit von Strahlern	
Oberfläche des Mondes	$2,5 * 10^3 \frac{cd}{m^2}$
Glühbirne (60 Watt)	$120 * 10^3 \frac{cd}{m^2}$
Sonnenscheibe am Horizont	$6 * 10^6 \frac{cd}{m^2}$
Sonnenscheibe am Mittag	$1,6 * 10^9 \frac{cd}{m^2}$

Tabelle 2.2: Leuchtdichte von Strahlern

Mit Dynamic Range bezeichnet man das Verhältnis zwischen zwei physikalischen Messungen. Abhängig vom gemessenen Medium unterscheidet sich die Definition. An dieser Stelle sollen nur visuell interessante Medien erwähnt werden:

- Bild: Verhältnis zwischen höchstem und niedrigstem Tonwert.
- Kamera: Verhältnis zwischen der Intensität, die eine Farbsättigung erzeugt, und dem kleinsten Wert der vom Rauschen der Kamera unterschieden werden kann.
- Reale Szene: Verhältnis zwischen dem hellsten und dunkelsten ankommenden Lichtstrahl

Studien [9] haben ergeben, dass es in der Natur in etwa eine Varianz von zehn Größenordnungen gibt. Das menschliche Auge ist dazu in der Lage, drei dieser Größenordnungen gleichzeitig wahrzunehmen. Die Größe der Pupille moderiert die Menge der einfallenden Lichtstrahlen, sodass man im gleißenden Sonnenlicht aber auch in einer düsteren Nacht noch etwas sehen kann. Zusätzlich beeinflussen chemische und kognitive Prozesse die Wahrnehmung. Ein typisches LCD-Display kann etwa einen Bereich über 2 Größenordnungen anzeigen:



(a)

Abbildung 2.1: Dynamic Range in Abhängigkeit vom Medium

2.2 Vorteile von High Dynamic Range

Wozu der ganze Aufwand, wenn man am Ende doch nur ein Low Dynamic Range-Anzeigegerät hat? Der Vorteil liegt darin, dass man mit natürlich vorkommender Beleuchtung rechnet und anschließend eine Transformation in einen Bereich durchführt, der vom Mensch als natürlich wahrgenommen wird. So ein Verfahren bezeichnet man als *Tonemapping* (s. Kapitel 5.4). Zudem vereinfacht es die

Erzeugung weiterer visueller Effekte wie man sie in der Realität auch des öfteren wahrnimmt, z.B. bei der Ausfahrt aus einem Tunnel, wenn sich die Beleuchtungssituation schlagartig ändert sieht man im ersten Moment relativ wenig, da man von der Sonne geblendet wird solange die Pupille den Lichteinfall nicht ausreichend moderiert. Bei dem Computerspiel “Far Cry” (mit HDR-Erweiterung) greift man solche Wahrnehmungserscheinungen auf und verwendet einen ähnlichen Effekt wenn man zuvor in Richtung der Sonne geschaut hat.

2.3 Technische Voraussetzungen für echtes HDR-Rendering

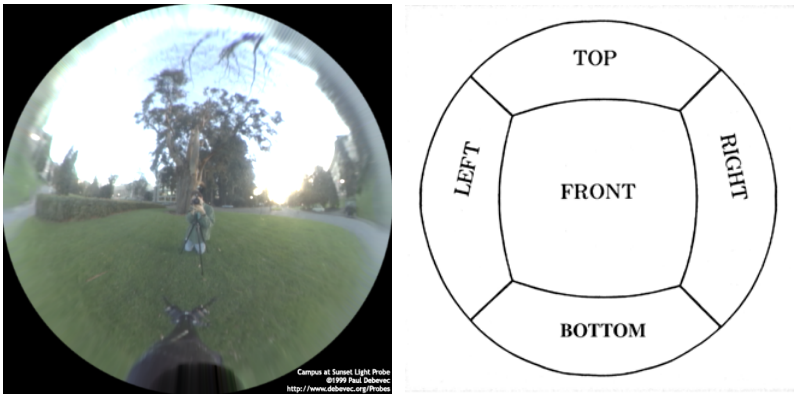
Konventionelles Rendering mit insgesamt 32 Bit pro Pixel (R/G/B/A - 8/8/8/8) kann keinen großen dynamischen Bereich erfassen, dafür sind 256 Abstufungen pro Farbkanal einfach nicht ausreichend. Mit höheren Bandbreiten und schnellerer Speicheranbindung aktueller Grafikhardware (z.B. Nvidia NV40) wird es auch ohne Tricks möglich, 64 und 128 Bit pro Pixel zu verwenden. Man kann einen Off-Screen Buffer (s. Kapitel 3.5) mit bis zu 32 Bit pro Farbkanal erzeugen und nach dem Rendervorgang das Ergebnisbild in einer Fließkommatextur abspeichern. Hat man zuvor mit einer 8 Bit Quantisierung je Farbe einen Bereich zwischen $[0..1]$ abgedeckt, sind nun fast sämtliche Schranken aufgehoben. Die Genauigkeit ist höher und man kann Farbwerte größer als 1.0 verwenden. Das bildet die Grundlage für den in dieser Arbeit verwendeten Ansatz des HDR-Lighting. Übersicht über verschiedene Bildformate:

Format	**Bits pro Farbkanal**	**max. Dynamic Range**
JPEG	8	256:1
16-Bit TIFF	16	65.536:1
HDR image	32	∞

Tabelle 2.3: Bittiefe im Hinblick auf HDR verschiedener Dateiformate

2.4 Erstellung von HDR-Environmentmaps

Für die Lichtberechnung benötigt man zu Beginn eine *Environmentmap*, welche die Beleuchtungssituation in einem natürlichen Umfeld widerspiegelt. Wie gewinnt man nun diese Information? Im Grunde geht es darum, eine sphärische Projektion, ausgehend von nur einem Betrachterstandpunkt, aufzunehmen. Dazu eignet sich eine spiegelnde Kugel. Wenn man sich weit genug von ihr entfernt und sie mit einem starken Teleobjektiv fotografiert, bekommt man annähernd eine $360°$ Rundumsicht. Lediglich das was sich hinter der Kugel verbirgt, kann man auf der Spiegelung nicht erkennen. Dann kommt noch hinzu, dass man zumindest die Kamera (in Abbildung 2.2 sieht man auch den Fotografen alias *Paul Debevec*) auf dem Spiegelbild sehen wird und sich die Auflösung des Spiegelbildes nicht linear auf die Raumwinkel verteilt:



(a) (b)

Abbildung 2.2: Campus Lightprobe und sich spiegelnde Würfelinnenseiten an einer spiegelnden Kugel

Wie man das Problem etwas abmildern kann, beschreibe ich im nächsten Abschnitt (s. Kapitel 2.5). *Paul Debevec* beschreibt in einer seiner Arbeiten [3] ein Verfahren zur Erstellung von HDR-Environmentmaps, das ich an dieser Stelle kurz beschreiben möchte. Eine Aufnahme mit einer Digitalkamera produziert ein Bild mit einer Auflösung von 8 Bit pro Farbkanal, d.h. für den Einsatz als HDR-Environmentmap viel zu wenig. Alle Werte, die eine bestimmte Helligkeit über oder unterschreiten, werden nicht weiter beachtet, also genau die Information geht verloren, an der man eigentlich interessiert ist. Um die abgeschnittenen Werte zu rekonstruieren macht man mehrere Aufnahmen der gleichen Szene mit unterschiedlicher Belichtungszeit, natürlich unter der Voraussetzung, dass sich an der Beleuchtungssituation während dieser Zeit nichts ändert! Das Verfahren basiert auf der Idee, dass der Sensor einer Kamera die einfallenden Photonen zählt. Verkürzt man die Belichtungszeit, wird die Auflösung der hellen Bereiche höher, da weniger schnell die maximale Anzahl der zählbaren Photonen erreicht wird. In der Praxis funktioniert das auf folgende Art: Zunächst bestimmt man den hellsten Punkt der Szene und stellt die Belichtungszeit so ein, dass an diesem Punkt genau die maximale Sättigung erzeugt wird. Anschließend stellt man die Belichtungszeit auf den 4-fachen Wert des vorherigen und nimmt erneut die Szene auf. Das wiederholt man so lange, bis die Belichtungszeit höher ist als für den dunkelsten Punkt der Szene nötig ist. Somit erhält man je nach Situation zwischen 3 und 5 Bildern, die man mit Hilfe der Software “HDRShop” in eine HDR-Environmentmap umwandeln kann.

2.5 HDR Angularmaps

Verglichen mit einer normalen Environmentmap, die ein Abbild einer spiegelnden Kugel ist, hat eine Angularmap nicht das Problem mit der schlechten Auflösung der Bereiche hinter der Kugel. Eine Angularmap entspricht einer Beleuchtungssituation von echten 4pi Steradian. Man erreicht dies indem man zwei Bilder einer spiegelnden Kugel aufnimmt. Eines wird auf herkömmlichem Wege und das zweite im $90°$ Winkel rotiert um die Kugel aufgenommen. Diese beiden Bilder werden

so kombiniert, dass das Zentrum des Bildes der Lichtfarbe von vorne entspricht und der Rand der Lichtfarbe von hinten. Ein weiterer Vorteil der Aufnahme zweier Bilder aus unterschiedlichen Richtungen ist außerdem, dass man die Kamera entfernen kann. Diese ist sonst bei Fotografien eines Spiegels immer sichtbar.

Der Algorithmus um aus einem normalisierten Richtungsvektor (D_x, D_y, D_z) den entsprechenden Punkt der Angularmap zu errechnen ist folgender:
Ich gehe hier davon aus, dass die Lookup-Koordinaten $u, v \epsilon [-1..1]$ sind.

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} D_x \\ D_y \end{pmatrix} \cdot \frac{1}{\pi} \cdot \frac{\arccos(D_z)}{\sqrt{D_x^2 + D_y^2}}$$

Durch Skalieren des Bereichs auf $[0..1]$ kann man auch ganz einfach Texturkoordinaten generieren, falls das einmal die Anwendung erfordern sollte. In meinem Programm verwende ich die Angularmaps aus *Paul Debevecs* Light Probe Gallery.

2.6 Das '.hdr' Datei-Format

Wie zuvor im Abschnitt 2.3 beschrieben ist die Voraussetzung ein Dateiformat mit außreichend Spielraum bei den Farbwerten, um man den großen dynamischen Bereich der in der Natur vorkommenden Lichtstrahlstärke auch sicher erfassen zu können. Dafür bietet sich ein Fließkommaformat mit mindestens 16 Bit Präzision an. Es gibt ein paar Formate, die diese Voraussetzungen erfüllen:

- Portable FloatMap

  4 Byte pro Farbkanal

  1 Vorzeichenbit, 8 Exp Bits, 23 Mant Bits

- OpenEXR (.exr)

  16 oder 32 Bit pro Farbkanal

- LogLuv TIFF Format

  basiert auf menschlicher Farbwahrnehmung

  24 oder 32 Bit

- Radiance (.hdr)

Greg Wards Radiance-Dateiformat möchte ich an dieser Stelle etwas ausführlicher beschreiben, da es im Zuge meiner Arbeit sinnvoll erschien, einen kleinen Parser hierfür zu entwickeln.

2.6.1 Der File Header

Der File Header eines Radiance-Bildes hat einen charakteristischen Aufbau und ist in Klartext formuliert. Es gibt eine Reihe von Schlüsselwörtern, die bestimmte Merkmale des Bildes beschreiben. Jede Radiance-Bilddatei beginnt mit der Zeichenfolge “#?RADIANCE”.

Schlüsselwort	**Syntax**	**Erläuterung**
FORMAT	“32-bit_rle_rgbe” |“32-bit_rle_xyze”	Bestimmt RGBE oder XYZE als Format für die Datei.
EXPOSURE	Fließkommawert	Bestimmt die Belichtungszeit der gespeicherten Ausgangswerte. Alle Farbwerte werden durch diesen Wert dividiert.
COLORCORR	3 Fließkommawerte	Ähnlich der Exposure Variable. Multipliziert jeden der drei Farbkanäle mit dem jeweils dafür angegebenen Wert.
SOFTWARE	beliebige Zeichenkette	Gibt die Programmversion an mit der das Bild erzeugt wurde.
PIXASPECT	Fließkommawert	Das Verhältnis der Höhe zur Breite eines Pixels.
VIEW	beliebige Zeichenkette	Gibt Information darüber, mit welchen Parametern das Bild erzeugt wurde.
PRIMARIES	8 Fließkommawerte	Die CIE(x,y) Farbwertanteile der drei Farbkanäle.

Tabelle 2.4: Radiance-Bilddatei Schlüsselwörter

Eine Leerzeile schließt den Header ab und es folgt die Auflösung des Bildes in der Form:
A [+|-]Y *yres* [+|-]X *xres*

X und Y sind vertauschbar. Wenn man das tut, vertauscht man die x und y Koordinaten. Als Folge davon entspricht das Einlesen einer Scanline nicht einer Zeile, sondern einer Spalte. Mit dem “+” und “-” Zeichen wird bestimmt, in welcher Reihenfolge die Pixel vorliegen. Im Bezug auf Y bedeutet “-”, dass die erste vorliegende Scanline die oberste des Bildes ist. Ein “+” bei X sagt aus, dass die Reihenfolge der gespeicherten Pixels von links nach rechts ist. Dies ist auch der Normalfall und praktisch alle erhältlichen HDR-Radiance Dateien sind so gespeichert. Der Grund für die anderen Reihenfolgen ist, dass die Konvertierung von anderen Formaten auf diese Art einfacher sein kann.

2.6.2 Decodieren des Datenblocks

Der Rest der Datei besteht aus einem großen Datenblock. Dieser ist eine Aneinanderreihung einzelner Scanlines, also im Regelfall einer Zeile (bei vertauschtem X/Y ist es eine Spalte) des Bildes. Da der Lesevorgang für jede Scanline gleich ist, genügt es exemplarisch das Vorgehen bei einer zu beschreiben. Dem Radiance-Dateiformat sind drei Arten der Scanlinekodierung bekannt:

- unkomprimiert
- alte Run Length-Kodierung
- neue Run Length-Kodierung

Jedes dieser Formate ermöglicht es, die Pixel einer Scanline im *RGBE-Format* auszudrücken. "RGB" steht dabei für die drei Farbkanäle und "E" für einen Exponenten, der dazu dient, einen normalen Farbwert in einen anderen Bereich zu verschieben. Das ist der prinzipielle Trick am Radiance-Dateiformat. Man benötigt lediglich vier Bytes um einen Pixel zu beschreiben ohne auf HDR verzichten zu müssen. Eine kleine Einschränkung gibt es allerdings: es wird immer derselbe Exponent auf alle drei Farbkanäle angewandt. Enthält einer der Farbkanäle einen besonders großen Wert, kann es passieren, dass man die anderen beiden "wegrundet". Wenn man das verschmerzen kann, hat man mit 32 Bit pro Pixel sogar die Möglichkeit, Bilder dieser Art direkt als Textur auf die 3D-Hardware laden zu können. Die Berechnung eines Farbwertes kann in diesem Fall auch in einem Pixelshader (s. Kapitel 3.3) erfolgen und funktioniert generell so:

$$r = R \cdot 2^{E-128}$$
$$g = G \cdot 2^{E-128}$$
$$b = B \cdot 2^{E-128}$$

2.6.3 Unkomprimiert

Das ist der primitive Ansatz, einfach alle Pixel nacheinander abzulegen, etwa vergleichbar mit einem Array. Die Pixel einer Scanline werden als Folge von 4 Byte-Werten gespeichert.

2.6.4 Alte Run Length-Kodierung

Die Idee der Run Length-Kodierung besteht darin, dass sich in bestimmten Bildern gleichfarbige Flächen befinden können. Somit besteht die Möglichkeit, dass sich ein Farbwert in einer Scanline wiederholt. Man speichert die Farbwerte wie in der unkomprimierten Form. Sollte der Wiederholungsfall eintreten, speichert man schlichtweg eine RGB Folge von $\{1, 1, 1\}$, was wegen der Normalisierung eigentlich kein zulässiger Wert ist. Der Exponent bestimmt die Anzahl der Wiederholungen des nächsten angebenen Farbwerts.

2.6.5 Neue Run Length-Kodierung

Die Scanline beginnt mit einem speziellen Farbwert, der bestimmt, dass in dieser das neue Verfahren verwendet wird:

$$R = 2, G = 2, B * 128 + E = L$$

Wenn L der Länge einer Scanline entspricht, dann erfolgt eine adaptive Run Length-Kodierung, aber diesmal für jeden Farbkanal und den Exponenten getrennt. Das erste Byte (B_1) gibt vor, ob es sich bei

den folgenden um einen “run” (sich wiederholende Farbwerte) oder nicht handelt. Ist das Bit größter Ordnung gesetzt ($B_1 >= 128$), handelt es sich bei dem nächsten Wert um einen “run” mit der Länge $B_1 - 126$. Andernfalls gibt B_1 die Anzahl variierender folgender Farbwerte vor. Mit diesem sehr einfachen Kompressionsverfahren erreicht man typischerweise eine Einsparung von 50%.

Kapitel 3

Hardwarebeschleunigtes Rendering mit OpenGL - Grundlagen

In diesem Kapitel beschreibe ich einige Grundlagen zum Rendering mit OpenGL, die von mir genutzt werden. Da ich im Vorfeld meiner Arbeit beschlossen habe, mit einem Linux Betriebssystem zu arbeiten, kam als hardwarebeschleunigtes Rendersystem nur OpenGL in Frage und das auch nur mit Nvidia Grafikkarten. Seitens ATI gibt es bisher keinen Treiber der die Erweiterungen unterstützt die ich verwendet habe.

3.1 OpenGL Render Pipeline

Rendert man ein Dreieck mit OpenGL, dann durchläuft es die komplette Render Pipeline. Diese lässt sich grob in drei Stufen aufteilen:

3.1.1 Application

In dieser Stufe befindet man sich in der Endanwendung. Hier wird die Kameratransformation gesetzt. Das schließt das Bestimmen der Position, Blickrichtung und weiterer Eigenschaften der Kamera mit ein. Als Ergebnis der Stufe wird die sog. Viewtransformation weitergegeben. Mit ihr bringt man eine beliebiges Objekt aus dem World-Space in den View-Space (Camera-Space).

3.1.2 Geometry

Die Geometriestufe erledigt die Modelltransformation und führt eine Lichtberechnung entsprechend des Phong-Lighting-Modells für jeden Vertex durch. Bei moderneren Grafikkarten (ab Nvidia Geforce) wird das auf der Grafikkarte mit Hilfe der sog. *fixed function pipeline* berechnet. Die eingehenden Vertexkoordinaten werden zunächst mit einer zusammengesetzten Matrix multipliziert, die den Vertex aus dem Object-Space direkt in den Eye-Space transformiert. Das Gleiche passiert auch mit den Vertexnormalen mit einer ähnlichen Transformationsmatrix. Bei Bedarf werden an dieser Stelle auch zusätzlich Texturkoordinaten generiert. Anschließend erfolgt die Lichtberechnung und das Clipping. Letztlich wird jeder Vertex mit einer Projektionstransformationsmatrix in den “unit cube”

abgebildet. Die berechneten Vertexkoordinaten und Zusatzinformationen werden abschließend der Rasterisierungsstufe übergeben.

3.1.3 Rasterizer

In dieser Stufe wird jedes der Dreiecke Scanline für Scanline gezeichnet. Farbwerte und Texturwerte der Eckpunkte können mit verschiedenen Verfahren interpoliert werden. Überlichweise greift man beim Rasterisieren auf einen Tiefenpuffer zurück, der angibt, in welcher Tiefe sich das bisher näheste Objekt befindet, um zu vermeiden, dass Objekte in kürzerer Entfernung vom Betrachter von solchen mit größerer Entfernung übermalt werden.

3.2 Limitierungen der klassischen OpenGL Rendering Pipeline

Mit dem Aufkommen der hardwarebeschleunigten Geometriestufe, die man auch mit T&L (hardware transformation & lighting) bezeichnet, hat man einen ungeheuren Geschwindigkeitszuwachs erlangt, da nun die CPU für andere Berechnung frei war und nicht mehr den Flaschenhals darstellte. Leider hat man aber auch viel an Flexibilität einbebüßt, die man nur durch den Schritt der Integration komplexer Recheneinheiten in die GPU zurückgewinnen konnte. Der erste Versuch dieser Art wurde von Nvidia mit der “Geforce 3” Grafikkarte unternommen und bis heute sukzessive fortgeführt. Die moderne Grafikpipeline kann man mit einem Modell schematisch darstellen:

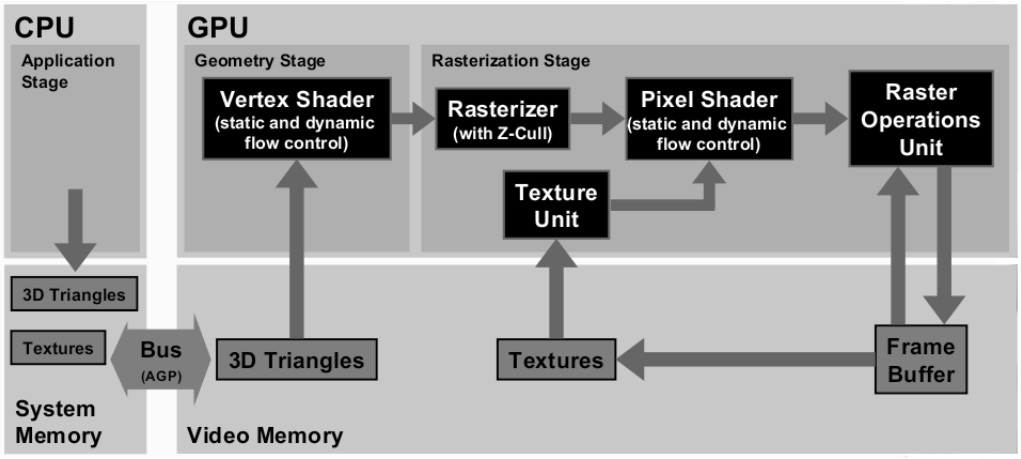


Abbildung 3.1: Render Pipeline nach Shader Modell 3.0

Die neu hinzugewonnenen Recheneinheiten werden mit Hilfe von Assembler-ähnlichen Befehlen programmiert. Um das Programmieren etwas einfacher zu gestalten, hat Nvidia eine Sprache entwickelt. Mittlerweile gibt es auch ähnliche Projekte der OpenGL-Entwickler mit dem Namen “GL Shading Language” (GLSL) und Micorsofts “High Level Shading Language” (HLSL).

3.3 Nvidias Programmiersprache CG (C for Graphics)

Bei Nvidia hat man versucht, eine C-ähnliche Programmiersprache zu entwerfen, die es ermöglicht, einen Shader in einer Hochsprache für Grafikkarten zu entwickeln. Darin sind u.a. Funktionsdeklarationen mit Rückgabewerten erlaubt, sowie Branching, Loops, Typprüfung und temporäre Variablen.
Die Existenz verschiedener Shaderversionen und unterschiedliche Hardwarefähigkeiten einzelner Grafikkarten legen es sogar sehr nahe, alle Programme in Cg zu schreiben und dann während der Laufzeit des Programmes erst eine für dieses System optimierte Version zu kompilieren. In einigen Fällen sind Compiler auch besser, da sie eventuell mehr architekturspezifisches Wissen besitzen. Mit CG lassen sich sowohl Vertexshader- als auch Pixelshader- (auch: *Fragmentshader*-) Programme erzeugen. Im Folgenden möchte ich darauf eingehen, was die Funktion dieser beiden Programmarten ist.

3.3.1 Vertexshader

Ein Vertexshader-Programm ersetzt die *hardware transformation & lighting* Stufe der “fixed function pipeline” der Grafikkarte (s. Kapitel 3.1.2). Alle Berechnungen werden mit 32 Bit Fließkommagenauigkeit ausgeführt. In diesem Programm kann man die Koordinaten eines Vertex abändern, Texturkoordinaten ausrechnen und Farbwerte für einen Vertex übergeben. Dabei ist zu beachten, dass Farbwerte grundsätzlich automaisch von der Hardware normalisiert werden, wohingegen Texturkoordinaten auch Werte größer als 1.0 annehmen können. Bei modernen Grafikkarten arbeiten mehrere Vertexshadereinheiten parallel, um bei gleicher Taktrate der GPU noch mehr Eckpunkte pro Zeiteinheit bearbeiten zu können. Eine Nvidia NV40 GPU hat 6 parallele Vertexshader.

3.3.2 Pixelshader

Ein Pixelshader-Programm wird nach der Geometriestufe ausgeführt und kann als Eingabe Werte von einem Vertexshader übergeben bekommen. Der Unterschied zu einem Vertexshader-Programm ist, dass es nicht auf Eckpunkte angewendet wird, sondern auf alle tatsächlich gezeichneten Punkte eines Dreiecks, die man auch als Fragmente bezeichnet. Pixelshader ersetzen die Shading Stufe der “fixed function pipeline”. In dieser wurde zuvor die Texturierung und Filterung mit vergleichsweise festen Einstellungen vorgenommen. “Pro Pixel”- Berechnungen ermöglichen schöne visuelle Effekte einschließlich besserer Beleuchtungsmodelle. Als Eingabewerte erhält man neben den Pixelkoordinaten auch Texturkoordinaten und andere beliebige konstante Werte oder Matritzen von der Applikation. Aktuelle Grafikkarten ermöglichen den Zugriff auf bis zu 8 verschiedene Texturen (s. Kapitel 3.4) in einem Durchlauf. Auch bei Pixelshadern setzt man auf eine parallele Ausführung um einen besseren Durchsatz zu erzeugen. Üblicherweise gibt es sogenannte “Quadpipes”, die jeweils 4 aneinanderliegende Pixel gleichzeitig berechnen. Diese führen das gleiche Shaderprogramm aus, auch wenn das zu zeichnende Dreieck nur einen der 4 Pixel belegt. Mit dem aktuellen Shader Modell 3 gibt es laut der Nvidia Dokumentation [19] echtes Looping und Branching bei der Ausführung von Pixelprogrammen. Das sollte unter bestimmmten Bedingungen große Performancevorteile bringen,

wenn man dadurch komplizierte Code-Teile überspringen kann, die für ein Fragment nicht anwendbar sind. Eine Nvidia NV40 GPU hat 16 parallele Pixelshader-Einheiten (4 Quadpipes).

3.4 Multitexturing

Multitexturing ist eine Technik, mit der man zur Berechnung des Farbwertes eines Pixels auf mehrere Texturen gleichzeitig zugreifen kann, um einen bestimmten Effekt zu erzielen. Mit aktueller Grafikhardware kann man auf bis zu 8 Texturen pro berechnetem Farbwert im Pixelshader zugreifen. Mit OpenGL funktioniert es so, dass man an jede dieser Textureinheiten eine Textur bindet mit den Kommandos:

```
1 glActiveTextureARB(GL_TEXTUREn_ARB);
2 glBindTexture(GL_TEXTURE2D, tex_n);
3 glEnable(GL_TEXTURE2D);
```

Texturkoordinaten müssen entweder für jeden Vertex angegeben werden (statisch), oder man kann sie generieren lassen (dynamisch). Dies geschieht mittels eines Vertexshader-Programms oder mit den entsprechenden GL-Kommandos. Hier ein Beispiel für die statische Variante:

```
1  /* Draw a textured quad */
2  glBegin( GL_QUADS );
3
4    // Bottom left of the texture and quad
5    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  0.0f);
6
7    // Bottom right of the texture and quad
8    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  0.0f);
9
10   // Top right of the texture and quad
11   glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  0.0f);
12
13   // Top left of the texture and quad
14   glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  0.0f);
15
16 glEnd();
```

Anschließend greift man mit dem Pixelshader-Programm auf die Texturen zu. In diesem Beispiel werden mit Hilfe der gleichen Texturkoordinaten zwei Texturen ausgelesen und als Ergebnis das arithmetische Mittel berechnet:

```
1 void main (      // in and out color value of a pixel
2                  inout double4 color : COLOR,
3
4                  // texture coordinate used for lookup
5                  float4 TexCoord0 : TEXCOORD0,
6
```

```
7                   // Texture2D sampler for the texture0 lookup
8                   uniform sampler2D Tex0: TEX0,
9
10                  // Texture2D sampler for the texture1 lookup
11                  uniform sampler2D Tex1: TEX1
12              )
13  {
14          // retrieve colorvalue of tex0 by lookup with TexCoord0
15          double temp = tex2D(Tex0, TexCoord0.st);
16
17          // retrieve colorvalue of tex1 by lookup with TexCoord0
18          double temp2 = tex2D(Tex1, TexCoord0.st);
19
20          color = ( temp + temp2 ) / 2.0;
21  }
```

3.5 Off-Screen Rendering

Nicht immer möchte man Berechnungen der Grafikkarte direkt anzeigen, z.B. wenn es sich um Zwischenergebnisse eines Multipass-Renderverfahrens handelt. Die Grafikhardware bietet die Möglichkeit ein Bild nicht direkt auf dem Bildschirm auszugeben, sondern in einen sog. *Off-Screen Buffer* zu rendern. Hierfür gibt es zwei Unterschiedliche Verfahren.

3.5.1 PBuffer

In meinem Programm habe ich das *Off-Screen Rendering* zunächst mit sog. *PBuffern* realisiert. Diese werden sowohl beim Shadowmapping (s. Kapitel 5.3) als auch beim Rendern der Szene mit HDR verwendet. Der Shadowmapping Algorithmus benötigt als Ausgangspunkt eine Tiefenkarte, generiert durch das Rendern der Szene mit der Kamera an Lichtquellenposition. Dieses Bild möchte man natürlich nicht sichtbar machen, deswegen ist hier der optimale Einsatzbereich für einen Off-Screen Buffer. Ein weiterer Vorteil ist, dass die Größe eines solchen Buffers nicht von der tatsächlich dargestellten Fenstergröße der OpenGL-Anwendung abhängt. Die Shadowmapgröße kann somit unabhängig gehalten werden. Möchte man den Buffer als Textur binden, ist das nicht direkt möglich, sondern man muss ihn umständlich in eine Textur kopieren, da Nvidia mit seinen Linuxtreibern keine direkte *render to texture* Unterstützung bietet.

Beispiel für einen PBuffer to Texture Kopiervorgang:

```
1   /* Create a new texture */
2   GLuint texture[1];
3   glGenTextures( 1, &texture[0] );
4
5   /* Bind a texture */
6   glBindTexture( GL_TEXTURE2D, texture[0] );
7
8
```

```
9     /* Set texture parameters for creation */
10    glTexImage2D( GL_TEXTURE2D, 0, GL_RGBA , xres, yres,
11                  0,  GL_RGB, GL_UNSIGNED_BYTE, NULL);
12
13    /* copy buffer to texture */
14    glCopyTexSubImage2D( GL_TEXTURE2D, 0, 0, 0, 0, 0, xres, yres );
```

In diesem Beispiel wird eine GL_TEXTURE2D erzeugt. Zu beachten ist, dass als Auflösung eine Zweierpotenz im Bereich [1..4096] gewählt werden muss.

3.5.2 OpenGL2.0 FBOs

Während der Entwicklungszeit meiner Arbeit wurde von Nvidia ein neuer Treiber für Geforce Grafikkarten veröffentlicht, der Framebuffer Objects (FBO) unterstützt. Diese kann man anstelle eines PBuffer verwenden, um einen Off-Screen Buffer zu erzeugen. Der große Vorteil daran ist, dass man das FBO direkt an eine Textur binden kann, ohne sie vorher aufwändig zu kopieren. Das bringt bei entsprechender Anwendung einen enormen Geschwindigkeitsvorteil. Zu diesem Thema habe ich einige Benchmarks angefertigt:

	FBO	PBuf	FBO	PBuf
Auflösung	**512x512**		**1024x768**	
Schloss	6,23	10,89	14,44	28,66
Voyager	19,62	22,94	27,93	41,12

Tabelle 3.1: Benchmark FPBuffer vs. FBO. Gemessen wurde die Zeit in [sec] bis 700 Einzelbilder akkumuliert wurden.

Diese Werte wurden ohne sonstige Optimierungen gemessen. Zusammengefasst kann man sagen, dass der Unterschied umso größer wird, je mehr die *copy to texture* Operation in den Vordergrund rückt und zum Flaschenhals wird. Das tritt dann bevorzugt auf, wenn das Modell relativ einfach und die Größe des Bildes relativ groß ist. Neben der Performanz gibt es noch einen weiteren Grund, der für die Verwendung von FBO's anstelle der PBuffer spricht. So war es mir nicht möglich, einen bereits erstellten PBuffer wieder vollständig auf der Grafikkarte freizugeben. Zunächst hatte ich eine Implementierung, die den PBuffer bei einer Änderung der Fenstergröße zerstört und danach mit der neuen Größe wieder erstellt. Ändert man die Größe des Fensters mehrmals, wird die Applikation plötzlich extrem langsam. Das liegt vermutlich daran, dass der Grafikspeicher vollläuft und die Daten zurück über den AGP-Bus in den Hauptspeicher ausgelagert werden. Nachdem einige Indizien dafür sprechen, dass FBOs nicht nur neuer und schneller sind als PBuffer, ist sicherlich ein Blick auf die Implementierung lohnenswert:

```
1     texTarget = [GL_TEXTURE2D | GL_TEXTURE_RECTANGLE_NV]
2
3     /* Bind another Framebuffer without complete context switch */
4     glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
5
6     /* Bind a texture */
7     glBindTexture(texTarget, tex);
```

```
8
9     /* Set parameters for the texture according to the framebufferproperties */
10    glTexImage2D(texTarget, 0, texInternalFormat, xres, yres, 0,
11                 GL_RGBA, GL_FLOAT, NULL);
12
13    /* Bind the texture on the framebuffer */
14    glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
15                              texTarget, tex, 0);
```

Dieses Beispiel illustriert das Verwenden eines FBOs als *render to texture* Ersatz. Die gebundene Textur enthält den Inhalt des Framebuffers und kann unmittelbar verwendet werden.

3.6 Rendering mit Fließkommawerten

Wie ich in Kaptitel 2 schon angedeutet habe, funktioniert echtes HDR-Rendering auf jeden Fall nur mit Fließkommawerten. Folgerichtig wird man nun feststellen, dass es wohl möglich ist, ein Bild mit Fließkommagenauigkeit im Shader zu berechnen und dann mit Hilfe eines Tonemapping-Verfahrens auf dem Bildschirm auszugeben. Das Problem entsteht erst dann, wenn ein adaptives Verfahren ins Spiel kommt, das ein Ergebnisbild sukzessive verbessert. Bei jedem nächsten Render-Durchlauf sollte man möglichst wenig an Information von den vorhergehenden Durchläufen verlieren. Aus diesem Grund rendere ich jedes Bild nicht direkt in den *Framebuffer* der angezeigt wird, sondern in einen Fließkomma-PBuffer (FPBuffer). Das ist eine OpenGL-Erweiterung von Sgi (GLX_SGIX_fbconfig und GLX_SGIX_pbuffer) und Nvidia (GL_NV_FLOATCOMPONENT). Auch dieses Verfahren konnte ich, nachdem ich es implementiert hatte, durch die FBOs vom vorherigen Abschnitt ersetzen, da diese mit 32 Bit RGBA Genauigkeit umgehen können. Nach dem Rendervorgang mit HDR erhält man dann ein echtes HDR-Bild, das man sich aufheben kann, indem man es an eine Fließkommatextur bindet. Nvidia hat in diesem Bereich ganze Arbeit geleistet und unterstützt Texturen mit 32 Bit pro Farbkanal, sodass die Genauigkeit nach einem Renderpass erhalten bleibt. Einen schönen Artikel über die unterschiedlichen Fließkommaformate und deren Besonderheiten in Shadern gibt es im 3DCenter [23].

Kapitel 4

Sampling

Sampling, oder das Entnehmen einer Stichprobe. Sampling ist ein Teilbereich der mathematischen Statistik, die dem Zweck dient, aus einer Gesamtmenge eine möglichst repräsentative Untermenge auszuwählen. Ein Sample bezeichnet genau den Teil der Gesamtmenge der tatsächlich ausgewählt wurde.

4.1 Grundlagen zum Sampling

Wie steht nun das Sampling in Zusammenhang mit der Computergrafik? Angenommen man betrachtet ein Bild als zweidimensionale kontinuierliche Funktion, dann benötigt man für die Darstellung dieses Bildes auf einem Medium mit diskreten Werten (Pixel) eine Samplingfunktion, die passende Stellen des Bildes auswählt, um sie anzuzeigen. Wie aus der Signaltheorie bekannt sein dürfte (Abtasttheorem), hängt eine günstige Abtastrate von der Frequenz des Eingangssignals ab. Stehen beide Werte in einem schlechten Verhältnis zueinander, bekommt man vor allem bei Unterabtastung seltsame Ergebnisse, da die Information zwischen den Abtastpunkten verloren geht. Um das zu verhindern, verwendet man Filter, die die verlorene Information mit einer bestimmten Gewichtung dem Abtastpunkt hinzufügen.
Nicht immer ist es sinnvoll gefilterte Werte zu verwenden. Im Zusammenhang mit meiner Arbeit, in der es darum geht eine Environmentmap abzutasten, steht von Anfang an nicht fest wie groß die Abtastrate sein wird, und deshalb ist es schwer möglich einen geeigneten Filter zu entwickeln. Um dennoch eine Aussage über das zu sampelnde Objekt machen zu können, gibt es progressive Sampleverfahren. Diese basieren auf der Idee, nach und nach immer mehr Abtastpunkte zu wählen, um der korrekten Lösung näher zu kommen. Eine Methode, die nach diesem Verfahren arbeitet, ist die Monte Carlo-Integration.

4.2 Monte Carlo-Integration

Die Monte Carlo-Integration ermöglicht es, den Wert eines Integrals abzuschätzen. Es genügt dabei vollkommen die Fähigkeit den Integranden auswerten zu können. In der Physik wird dieses Integrationsverfahren häufig eingesetzt. Die Auswertung des Integranden entspräche dabei der Ermittlung eines Messwertes. Die generelle Funktionsweise des Verfahrens ist so, dass man zufällig

bestimmte Stellen innerhalb des Integrals auswählt, den Wert des Integranden an dieser Stelle bestimmt und anschließend mit den berechneten Werten das Ergebnis des Integrals abschätzt. Das zu bestimmende Integral hat folgendes Aussehen:

$$\int_a^b f(x)dx$$

Eine Schätzung für das Integral erhält man, indem man N Samples zieht:

$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i)$$

Vorraussetzung für eine unabhängige Schätzung ist, dass die Messwerte zufällig und gleichverteilt aus dem Intervall $[a, b]$ gewählt werden. Ein großer Vorteil des Verfahrens ist es, dass man an keine feste Abtastrate gebunden ist. Somit gilt der Grundsatz, dass mit steigender Anzahl der Samples der relative Fehler zum richtigen Ergebnis kleiner wird. Der Fehler verhält sich wie $\frac{1}{\sqrt{n}}$ und hängt nicht von der Dimension des Integrals ab.

4.3 Monte Carlo-Sampling

Das Monte Carlo-Sampling ist eine Anwendung der Monte Carlo-Integration. Man geht davon aus, dass die Wahrscheinlichkeitsdichte der zu sampelnden Gesamtmenge eine konstant verteilte Funktion ist:

$$p(X_i) = const$$

Das bedeutet, dass jedes Sample mit gleicher Wahrscheinlichkeit unabhängig voneinander gezogen wird. Somit sind die Voraussetzungen für die Integrationsmethode erfüllt. Jedes gezogene Sample verbessert das Ergebnis. Der Nachteil dieser Methode ist, dass es bei zufälliger Verteilung zu Sampleklumpen kommen kann, wohingegen andere Teile überhaupt kein Sample erhalten. Das hat zur Folge, dass zwar statistisch gesehen das Ergebnis mit jedem dazukommenden Sample besser wird, in der Praxis aber führt dies zu einer sehr hohen Varianz der Ergebnisse. Beim reinen Monte Carlo-Sampling kann man das nur mit einer Vielzahl von Samples ausgleichen. Andere Ansätze, wie etwa *Stratified-Sampling* haben dieses Problem nicht so stark, da sie die Samples nicht zufällig, sondern gleichmäßig verteilt wählen, sind aber deshalb nur eingeschränkt adaptiv verwendbar.

4.3.1 Quasi Monte Carlo-Sampling

Die Quasi Monte Carlo-Methode zieht Samples nicht rein zufällig wie beim Monte Carlo-Sampling, sondern man verwendet eine deterministische Zahlenfolge die möglichst gleichverteilt Samples aus einem gewünschten Bereich zieht, um die Sampleklumpenbildung und das Auftreten von schlecht gesampelten Bereichen zu vermeiden. Das soeben erwähnte *Stratified Sampling* ist z.B. so ein Ansatz. Eine adaptiv verwendbare Zahlenfolge, mit der man bessere Ergebnisse erzielt, ist das *Sobol Sampling* Verfahren. Besprechungen weiterer Algorithmen kann man in [15] und [16] nachlesen. Eine weitere wichtige Arbeit zu diesem Thema im Hinblick auf die Computergrafik mit globaler Beleuchtungsberechnung stammt von *Alexander Keller* [12]. Eines der zentralen Ergebnisse seiner

Arbeit ist, dass Quasi Monte Carlo-Methoden schneller konvergieren als echte Zufallsverteilungen. In meinem Anwendungsfall kann ich dieses Ergebnis auch bestätigen, da die durchschnittliche Varianz der Unterschiede zwischen einzelnen Rendervorgängen dadurch vermindert wird.

4.4 Importance-Sampling

Importance-Sampling ist ein Ansatz die Monte Carlo-Abschätzung schneller konvergieren zu lassen, also eine Methode zur Varianzreduktion. Dabei nutzt man sämtliches Wissen über den Integranden und versucht Stellen bevorzugt auszuwählen, die in starkem Maße die Größe des Integranden bestimmen. Zu diesem Zweck erstellt man eine Wahrscheinlichkeitsverteilung (p.d.f), die es ermöglicht eben diese Samples vermehrt auszuwählen, die *wichtig* erscheinen (Stellen, an denen der Integrand sehr groß ist):

$$p(X_i) \neq const$$

Im Regelfall verwendet man eine einfache Verteilungsfunktion, die in etwa dem Verlauf des Integrals entspricht und von der das Integral bekannt ist. Desto ähnlicher sich die beiden Funktionen sind, umso genauer wird die Abschätzung:

$$\int_a^b f(x)dx = \int_a^b \frac{f(x)}{p(x)}p(x)dx \tag{4.1}$$

$$F_N = \frac{b-a}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)} \tag{4.2}$$

Damit gelingt es schon mit den ersten Samples eine gute Näherung zu berechnen. Dieses Verfahren kann natürlich nur Verwendung finden, wenn es möglich ist eine p.d.f aufzustellen. In der Computergrafik kann man dieses Verfahren meistens verwenden, da es selten darum geht den Wert des Integrals zu bestimmen, sondern lediglich darum, zur Ausführungszeit schnell repräsentative Samples zu gewinnen. Im Optimalfall wählt man den Integranden selbst:

$$p(x) = cf(x)$$

Durch Normalisierung der kumulativen Verteilungsfunktion ($\int p(x)dx = 1$) folgt:

$$p(x) = \frac{f(x)}{\int f(x)dx}$$

Da c eine Konstante ist, folgt für jedes Sample der Wert des Integrals als Ergebnis. Die Varianz ist minimal und liegt bei 0. Etwas absurd klingt das Ganze zunächst, da man das Ergebnis welches man ursprünglich erhalten wollte zum Aufstellen der Wahrscheinlichkeitsdichtefunktion verwendet. Übertragen auf meine Anwendung zum Sampeln von Environmentmaps macht dieses Verfahren allerdings sehr wohl Sinn. Dieses wird im Folgenden ausführlich erläutert.

4.5 Sampling von Environmentmaps

Das Sampling von Environmentmaps ist komplexer als es zunächst erscheinen mag. Eine Vereinfachung wurde getroffen, indem die Maps im Latitude/Longitude (LL) Format vorliegen. In

den folgenden Abschnitten stelle ich zunächst meine erste Idee vor und anschließend die korrekte mathematische Herleitung.

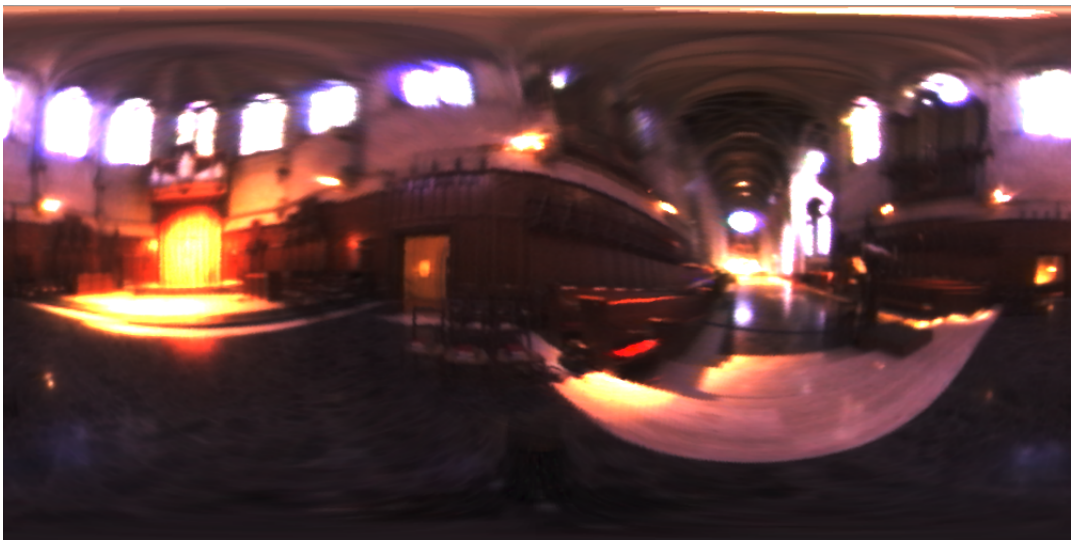


Abbildung 4.1: Die Grace Cathedral HDR-LL Map

4.5.1 Der intuitive Ansatz

Offensichtlich handelt es sich beim LL-Format nicht um eine konstante Verteilung der Kugelfläche im Verhältnis zur Pixelgröße. Aus diesem Grund ist kein direktes Monte Carlo-Sampling möglich, da die Gewichtung der einzelnen Pixel nicht gleich ist:

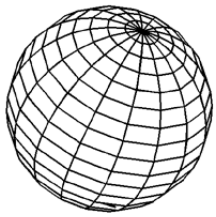


Abbildung 4.2: Die Größe von LL-Map Punkten auf einer Kugel.

Für den Zugriff auf eine LL-Map verwendet man Kugelkoordinaten (Raumpolarkoordinaten) (θ, ϕ). Wie man in der Grafik 4.2 gut erkennen kann, ist eine Runde um die Kugel in Äquatornähe deutlich länger als an den Polen, wird aber auf die selbe Anzahl von Pixeln in der LL-Map abgebildet. Diese Einschränkung könnte man mit einer geeigneten p.d.f ausgleichen:

$$p(X_i) = sin(\theta)$$

Entsprechend des Breitengrades wird der Beitrag des Lookups zum Gesamtergebnis verkleinert. Das ist der erste Schritt, um ein echtes Monte Carlo-Sampling auf einer Environmentmap zu

realisieren. Nach wie vor gibt es das Problem, dass bei zufälliger Auswahl wesentlich öfter ein Funktionswert an den Polen gewählt wird. Eine korrekte Lösung findet man durch eine mathematische Herangehensweise.

4.5.2 Grundlagen

4.5.2.1 Kugelkoordinaten

Die Transformationsgleichungen von kartesischen Koordinaten in Kugelkoordinaten werden benötigt um von einem bestimmten Punkt auf der Einheitskugel ($r = 1$) die entsprechnenden Koordinaten auf der LL-Map ausrechnen zu können.

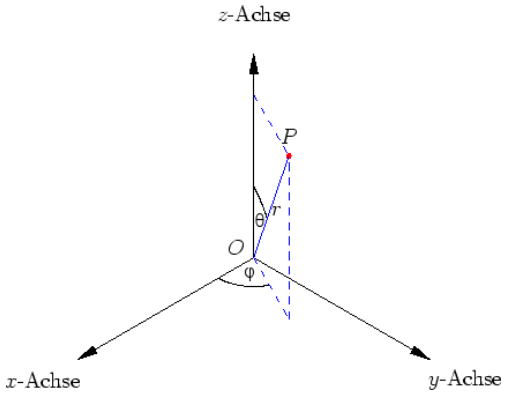


Abbildung 4.3: Kugelkoordinaten

Im Allgemeinen werden Kugelkoordinaten in der Form $P(r, \theta, \phi)$ angegeben. Für die einzelnen Komponenten gelten die Definitionen:

r :	*Radius* ist der Abstand des Punktes P vom Koordinatenursprung O.
$\theta \, \epsilon \, [0, \pi]$:	*Polarwinkel* ist der Winkel zwischen der positiven z-Achse und r.
$\phi \, \epsilon \, [0, 2\pi]$:	*Azimutwinkel* ist der Winkel zwischen der positiven x-Achse und r_{xz}.

Für die Transformation eines Punktes von kartesischen Koordinaten in Kugelkoordinaten gelten folgende Gleichungen:

$$r = \sqrt{x^2 + y^2 + z^2} \tag{4.3}$$

$$\theta = arccot \frac{z}{\sqrt{x^2 + y^2}} \tag{4.4}$$

$$\phi = \begin{cases} arccos \frac{x}{\sqrt{x^2+y^2}} & \text{für } y >= 0 \\ 2\pi - arccos \frac{x}{\sqrt{x^2+y^2}} & \text{für } y < 0 \end{cases} \tag{4.5}$$

Die Rücktransformation eines Punktes in Kugelkoordinaten in kartesische Koordinaten erfolgt nach den Gleichungen:

$$x = r \sin\theta \cos\phi \tag{4.6}$$
$$y = r \sin\theta \sin\phi \tag{4.7}$$
$$z = r \cos\theta \tag{4.8}$$

Für die meisten Berechnungen genügt die Betrachtung der Einheitskugel. Das bedeutet, dass man r durch 1 ersetzt und nicht als eigenständige Koordinate mitführt, was dazu führt, dass man sich einige Berechnungen sparen kann.

4.5.2.2 Zweidimensionales Sampling

Unter zweidimensionalem Sampling versteht man die Auswahl zweier Zufallsvariablen (X, Y) die eine Dichte $p(x, y)$ besitzen. Nicht immer besteht die Möglichkeit, dass X und Y unabhängig voneinander gewählt werden können. Damit man trotzdem ein Sample bestimmen kann, integriert man eine Dimension in die andere hinein (Bestimmung der Marginaldichte):

$$p(x) = \int p(x, y)dy$$

Die Wahrscheinlichkeit ein X zu wählen ($p(x)$) ergibt sich aus der durchschnittlichen Wahrscheinlichkeit aller y für dieses x. Somit kann man bereits ein X ziehen. Der nächste Schritt ist die Auswahl eines y unter der Bedingung x:

$$p(y|x) = \frac{p(x, y)}{p(x)}$$

Jetzt gelingt es auch ein passendes Y zu finden. Übertragen auf das Samplen einer Environmentmap kann man sich das so vorstellen, dass man zunächst eine Spalte der Map auswählt und anschließend in dieser Spalte eine Zeile wählt.

4.5.2.3 Die Inversionsmethode

Die Generierung uniform verteilter Zufallszahlen ist im Regelfall sehr einfach. Nicht immer hat man allerdings den Luxus einer konstanten Verteilungsfunktion, somit ist es zweckmäßig eine Möglichkeit zu finden, uniform verteilte Zufallszahlen in Zufallszahlen einer bestimmten Wahrscheinlichkeitsverteilung umwandeln zu können. Ein solches Verfahren wird in [21] beschrieben und als die *Inversionsmethode* bezeichnet:

- Bestimme eine kumulative Verteilungsfunktion $P(x) = \int_0^x p(x')dx'$
- Berechne die Umkehrfunktion der kumulativen Verteilungsfunktion $P^{-1}(x)$
- Bestimme eine uniform verteilte zufällige Zahl ξ
- Berechne die Zufallsvariable $X_i = P^{-1}(\xi)$

Anschließend kann für das ξ ein Wert im Intervall $[0..1]$ gewählt werden.

4.5.3 Monte Carlo-Sampling von Environmentmaps

Eine Environmentmap beschreibt die Helligkeitseinstrahlung auf den Mittelpunkt einer Kugel. Das Ziel des Sampling ist es einzelne Punkte dieser Map auszuwählen um damit bei einer ausreichenden Anzahl eine repräsentative Helligkeitsverteilung (eine korrekte Approximation entspricht dem Integral) für die gesamte Map zu erhalten. Diesen Prozess kann man in zwei Teile aufgliedern. Es muss zunächst einmal gelingen, gleichmäßig auf einer Kugeloberfläche Samples auszuwählen. Anschließend kann man mit der Approximation der Gesamthelligkeit beginnen.

4.5.3.1 Uniformes Sampling einer Kugel

Die Kugeloberfläche hat einen Flächeninhalt von $4\pi r^2$. Dies entspricht dem vollen Raumwinkel von $4\pi sr$. Teilt man diesen nun gleichmäßig in einzelne Parzellen auf, erhält man eine konstante Wahrscheinlichkeitsdichte ($p(\omega) = const$). Das bedeutet, dass die Wahrscheinlichkeit für die Auswahl einer jeden Parzelle gleich ist. Erstellt man daraus eine kumulative Verteilungsfunktion, sollte man normieren:

$$\int p(\omega) d\omega = 1 \tag{4.9}$$

$$c \int d\omega = 1 \tag{4.10}$$

$$c = \frac{1}{4\pi} \tag{4.11}$$

Für den Zusammenhang zwischen Raumwinkel und Kugelkoordinaten gilt:

$$\Omega = \int_{\phi_1}^{\phi_2} \int_{\theta_1}^{\theta_2} \sin\theta \; d\theta \; d\phi$$

Daraus kann man den Zusammenhang des differenziellen Raumwinkels ($d\omega$) und der differenziellen Fläche in Kugelkoordinaten (dA) ableiten:

$$d\omega = dA \tag{4.12}$$

$$d\omega = \sin\theta \; d\theta \; d\phi \tag{4.13}$$

Nachdem der Raumwinkel allerdings keine Aussage über die Position auf der Kugel macht, ist es sinnvoll die Wahrscheinlichkeitsdichte auf Kugelkoordinaten zu übertragen:

$$p(\omega) d\omega = p(\theta, \phi) \; d\theta \; d\phi$$

Durch Einsetzen für $d\omega$ erhält man:

$$p(\theta, \phi) = \sin\theta \; p(\omega)$$

Verwendet man Monte Carlo-Sampling ist $p(\omega) = const$ und wurde bereits berechnet:

$$p(\theta, \phi) = \frac{1}{4\pi} \sin\theta$$

Da es sich um eine zweidimensionale Verteilungsfunktion handelt, berechnet man die Marginaldichte:

$$p(\theta) = \int_0^{2\pi} p(\theta, \phi) d\phi = \int_0^{2\pi} \frac{\sin\theta}{4\pi} d\phi = \frac{1}{2}\sin\theta$$

Bedingte Wahrscheinlichkeit für ϕ unter der Hypothese, dass θ eingetroffen ist:

$$p(\phi|\theta) = \frac{p(\theta,\phi)}{p(\theta)} = \frac{\frac{1}{4\pi}\sin\theta}{\frac{1}{2}\sin\theta} = \frac{1}{2\pi}$$

Nun verwendet man die Inversionsmethode, um aus einer uniform verteilten Zufallszahl die gewünschte Verteilung zu erhalten (s. Abschnitt 4.5.2.3):

$$P(\theta) = \int_0^{\theta} \frac{1}{2}\sin\theta' \, d\theta' = \frac{1}{2}(1 - \cos\theta)$$

$$P(\phi|\theta) = \int_0^{\phi} \frac{1}{2\pi} d\phi' = \frac{\phi}{2\pi}$$

Setzt man für $P(\theta)$ die uniform verteilte Zufallsvariable ξ_1 und für $P(\phi|\theta)$ die uniform verteilte Zufallsvariable ξ_2 ein, folgt:

$$\theta = \arccos(1 - 2\xi_1)$$

$$\phi = 2\pi\xi_2$$

Somit gelingt es aus zwei uniform verteilten Zufallsvariablen eine gleichmäßige Auswahl von Samplen auf einer Kugel zu treffen. Dieses Verfahren ist natürlich auch leicht auf eine der beiden Hemisphären zu übertragen.

4.5.3.2 Approximation der Helligkeit

Um die Helligkeit der gesamten Map abzuschätzen, verwendet man die Monte Carlo-Integration (s. Abschnitt 4.2). Dazu summiert man alle bisher gezogenen Samples und dividiert anschließend durch deren Anzahl. Da in diesem Fall eine uniforme Verteilung der Samples über die Kugel vorliegt, genügt der Ansatz:

$$F_N = \frac{1}{N}\sum_{i=1}^{N} f(X_i)$$

4.5.4 Quasi Monte Carlo-Sampling von Environmentmaps

Analog zum Monte Carlo-Sampling von Environmentmaps kann man auch Pseudo-Zufallszahlen verwenden. In meiner Implementierung verwende ich einen 2D-Sobol Algorithums wie er in [21] beschrieben ist. Dieser verwendet für den ersten Zahlenwert die *Van der Corput* Zahlenfolge und für den zweiten die *Sobol*-Zahlenfolge. Es gibt bessere Zahlenfolgen für eine gleichmäßige Verteilung, aber dafür ist der *Sobol*-Algorithmus adaptiv einsetzbar und hängt nicht von der Gesamtzahl der zu ziehenden Samples ab.

4.5.5 Importance-Sampling von Environmentmaps

Das Importance-Sampling-Verfahren, das im folgenden Abschnitt beschrieben wird, stammt von *Matt Pharr* und *Greg Humphreys* [20]. Es basiert auf der Idee, dass man aus einer Environmentmap mit Helligkeitswerten eine zweidimensionale stückweise konstante Funkion erstellt.

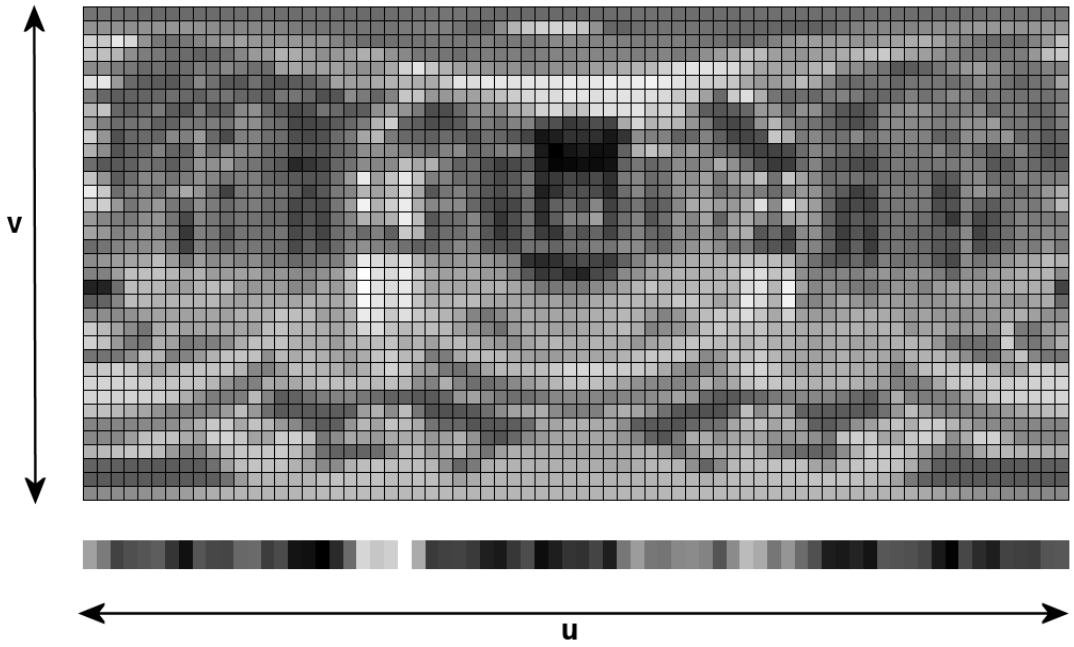


Abbildung 4.4: Quantisierung einer LL-Map in diskrete Helligkeitswerte. Darunter die akkumulierten Spaltenhelligkeiten.

Eine LL-Map entspricht einer 2-dimensionalen Funktion $f(u, v)$. u ist der Lookupwert für die Spalte und v für die Zeile. n_u und n_v geben die jeweils maximalen Quantisierungsstufen in der jeweiligen Dimension an. Man geht davon aus, dass beide Werte u und v im Intervall $[0..1]$ liegen. Der Funktionswert ist die Helligkeit des Pixels multipliziert mit $\sin(v\pi)$. Die Multiplikation mit dem Sinuswert ist erforderlich, um die Verzerrung an der LL-Map in Richtung der Pole auszugleichen. Dadurch erreicht man ein uniformes Sampling über die Kugelfläche. Das Ziel des Importance Sampling ist es zwei uniform verteilte Zufallsvariablen so umzuwandeln, dass vermehrt sehr helle Stellen der Environmentmap ausgewählt werden, da sie hauptsächlich für die Beleuchtungssituation der Szene verantwortlich sind. Um das zu erreichen, stellt man zunächst eine Wahrscheinlichkeitsdichtefunktion für die Auswahl eines Pixels der LL-Map auf:

$$p(u, v) = \frac{f(u, v)}{\sum_{u=0}^{1} \sum_{v=0}^{1} f(u, v)}$$

Da es sich um zweidimensionales Sampling handelt, integriert man die Zeilen in die Spalten hinein und erhält damit die Wahrscheinlichkeit für die Auswahl einer Spalte (marginale Dichte):

$$p_u(u) = \frac{\sum_{v=0}^{1} f(u,v)}{\sum_{u=0}^{1}\sum_{v=0}^{1} f(u,v)}$$

Für die bedingte Wahrscheinlichkeit der Auswahl einer Zeile v in der Spalte u erhält man:

$$p_v(v|u) = \frac{p(u,v)}{p_u(u)}$$

Analog zum Monte Carlo-Sampling von Environmentmaps verwendet man auch hier die Inversionsmethode um uniform verteilte Zufallszahlen ξ_1 und ξ_2 in u und v umzuwandeln. Hierfür kann man keine allgemein gültige Lösung berechnen, da die Wahrscheinlichkeitsverteilung von der Environmentmap abhängt. Zu diesem Zweck erstellt man kumulative Verteilungsfunktionen. Eine für die Auswahl der Spalte und anschließend für jede Spalte eine eigene zur Auswahl der Zeile. Unter der Voraussetzung, dass diese Funktionen normiert sind, ist es ein Leichtes die Umkehrfunktion zu bestimmen. Die kumulativen Verteilungsfunktionen werden repräsentiert durch eindimensionale Arrays. Für die Auswahl der Spalten erzeugt man:

$$P_u(u) = \int_0^u p_u(u')du'$$

Für jede Spalte zur Auswahl einer Zeile in dieser Spalte erzeugt man:

$$P_v(v|u) = \int_0^v p_v(v'|u)dv'$$

Ist ein bestimmter Wert besonders wahrscheinlich, nimmt er entlang der y-Achse der Funktion besonders viel Platz ein und wird von uniform verteilten Zufallszahlen häufiger getroffen. Deswegen ist das Vorgehen so:

- Bestimme eine Zufallszahl $\xi_1 \epsilon [0..1]$
- Laufe der Reihe nach alle Werte des Arrays, das $P_u(u)$ repräsentiert ab, bis $\xi_1 > P_u(x)$
- Bestimme eine Zufallszahl $\xi_2 \epsilon [0..1]$
- Laufe der Reihe nach alle Werte des Arrays, das $P_v(v|x-1)$ repräsentiert ab, bis $\xi_2 > P_v(y|x)$

Die erhaltenen Zufallszahlen sind $X = x - 1$ und $Y = y - 1$. Der Lookup des Helligkeitswertes könnte mit den beiden Werten erfolgen. Das allerding würde zu einem falschen Ergebnis führen! Stellt man sich vor, dass eine Map einen relativ kleinen, sehr hellen Bereich enthält, dann hat das viele gezogene Samples in diesem Teil zur Folge. Bildet man mit Hilfe der Monte Carlo-Integrationsmethode eine Abschätzung für die gesamte Leuchtkraft der Enironmentmap, würde unter diesen Voraussetzungen ein viel zu großes Ergebnis herauskommen. Man kann es sich so vorstellen, dass bei dem Importance-Sampling-Verfahren nicht die hellen Bereiche “heller strahlen” sollen als die dunklen, sondern einfach öfter ausgewählt werden. Der gedankliche Fehler den man an dieser Stelle begehen kann ist, dass man vergisst die Wahrscheinlichkeitsdichte für die Auswahl des Samples mit einzubeziehen.

Für die Dichte der Zufallsvariablen zur Auswahl eines Samples auf einer Kugel gilt:

$$p(\omega) = \frac{p(\theta, \phi)}{\sin \theta}$$

Als Transformationsfunktion für ein gezogenes Sample (u, v) in Kugelkoordinaten (θ, ϕ) gilt:

$$g(u, v) = (\frac{\pi v}{n_v}, \frac{2\pi u}{n_u})$$

Mit Hilfe eines Verfahrens, das in [21] (S. 648) beschrieben wird, kann man die Wahrscheinlichkeitsdichte einer Funktion auf eine andere übertragen, wenn deren Beziehung bekannt ist. Dazu bestimmt man die Determinante der Jacobi-Matrix der Ausgangsfunktion:

$$|J_g| = \frac{2\pi^2}{n_u n_v}$$

Für die neue Wahrscheinlichkeitsdichte $p(\theta, \phi)$ gilt:

$$p(\theta, \phi) = p(u, v)|J_g|$$

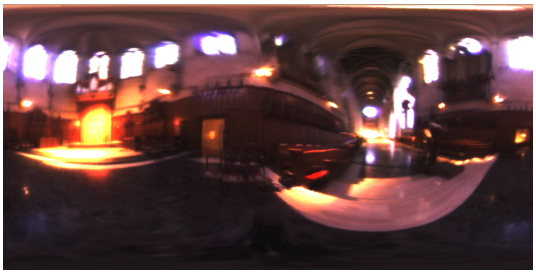
Setzt man alle Terme ein, folgt daraus der Wert für $p(\omega)$:

$$p(\omega) = p(u, v)\frac{n_u n_v}{2\pi^2 \sin \theta}$$

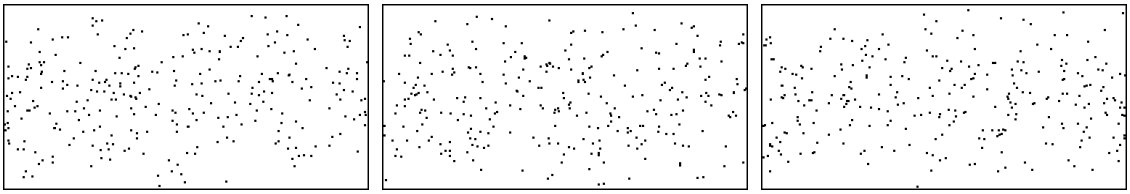
Multipliziert man $p(\omega)$ mit dem Lookupwert g(u,v) erhält man den korrekten Helligkeitswert. Dieser ist auch immer gleich, was nicht verwunderlich ist, da die kumulative Verteilungsfunktion mit Hilfe des Integranden selbst erzeugt wurde (s. Abschnitt 4.4) und daraus folgernd eine Varianz von 0 entsteht.

4.6 Die Methoden im Vergleich

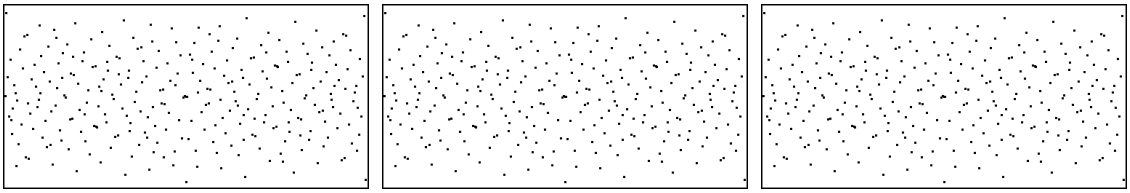
Das *Random-Sampling-Verfahren* bringt die schlechtesten Ergebnisse. Hier ist die Varianz zwischen unterschiedlichen Rendervorgängen sehr groß. Das verstärkt sich noch, wenn die Environmentmap lediglich einige wenige sehr helle Bereiche enthält, dann hängt es davon ab, ob diese zufällig getroffen werden.
Das *Sobol-Sampling-Verfahren* vermindert die Varianz zwischen den einzelnen Durchläufen, da immer die gleiche Folge von Zufallszahlen verwendet wird. Die zu erwartende Verbesserung durch eine gleichmäßigere Abtastung der Environmentmap wird nur in speziellen Fällen erreicht, etwa wenn es nur eine sehr helle Lichtquelle auf einer gesamten Map gibt. Dann kann man davon ausgehen, dass eines der Samples in diesen hellen Bereich fallen wird, da eine gleichmäßigere Verteilung als beim Random-Sampling vorliegt.
Wie zu erwarten war, erhält man mit dem *Importance Sampling* das mit Abstand beste Ergebnis. Durch die zusätzliche Kombination des Importance Sampling mit dem Sobol-Verfahren wird kaum eine Verbesserung erzielt. Somit genügt für die meisten Environmentmaps bereits eine geringe Anzahl von Abtastwerten, um einen repräsentativen Eindruck zu erhalten.



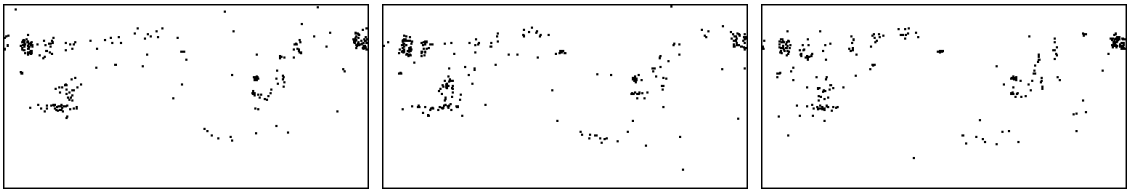
(a) Grace Cathedral HDR-Environmentmap



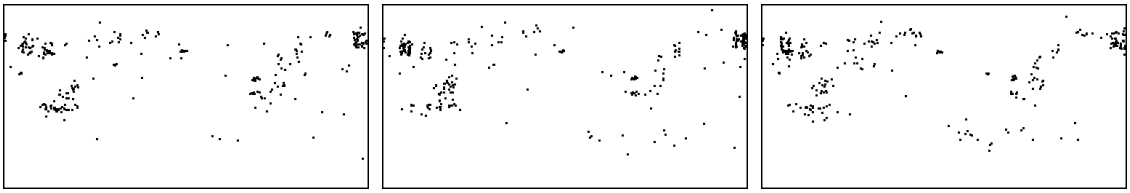
(b) Random 1 (c) Random 2 (d) Random 3



(e) Sobol 1 (f) Sobol 2 (g) Sobol 3

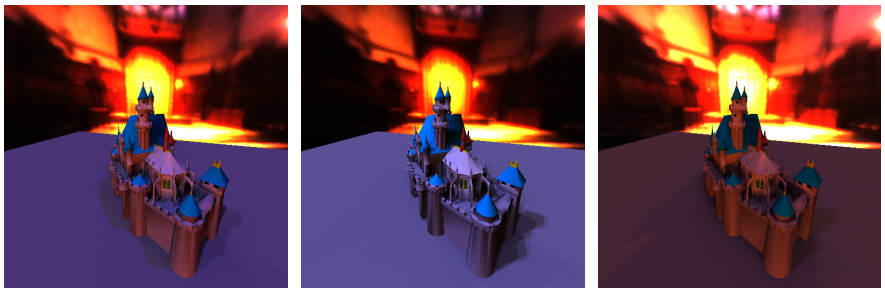


(h) Importance 1 (i) Importance 2 (j) Importance 3

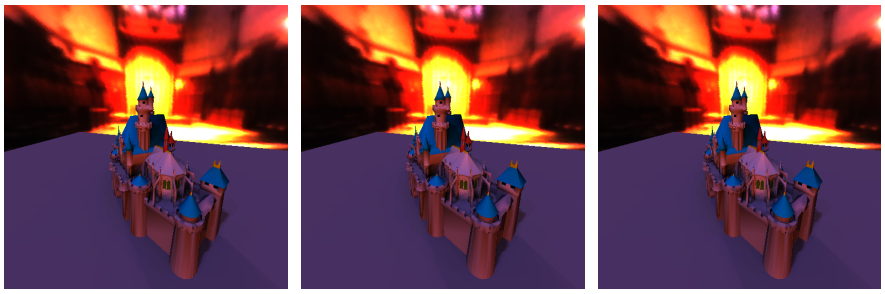


(k) Importance+Sobol 1 (l) Importance+Sobol 2 (m) Importance+Sobol 3

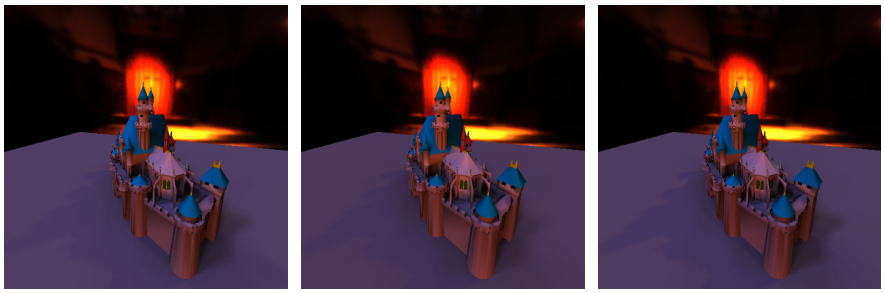
Abbildung 4.5: Gewählte Samples unterschiedlicher Verfahren auf der *grace_probe*-HDR-Environmentmap.



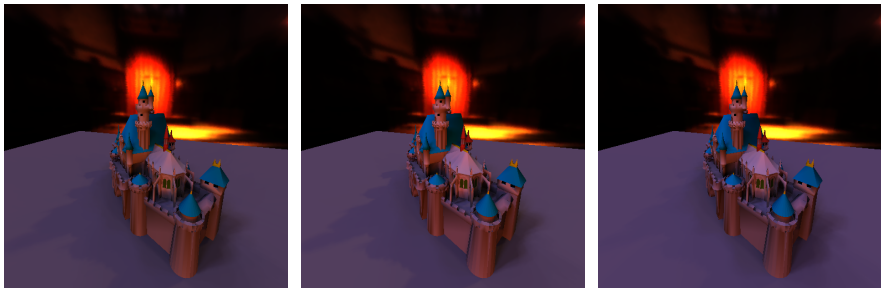
(a) Random 1 (b) Random 2 (c) Random 3



(d) Sobol 1 (e) Sobol 2 (f) Sobol 3



(g) Importance 1 (h) Importance 2 (i) Importance 3



(j) Importance+Sobol 1 (k) Importance+Sobol 2 (l) Importance+Sobol 3

Abbildung 4.6: Renderings mit den in 4.5 gewählten Lichtquellen.

Kapitel 5

Algorithmen zur Beleuchtungs- und Schattenberechnung

In diesem Kapitel möchte ich grundlegende Konzepte zur Beleuchtungs- und Schattenberechnung vorstellen, die in meiner Arbeit verwendet werden.

5.1 Phong-Lighting

Das Phong-Lighting-Modell ist bis heute der De-facto-Standard zur Lichtberechnung. Das liegt vor allem daran, dass es ein einfaches Verfahren ist, das schöne Ergebnisse hervorbringt, obwohl es physikalisch nicht korrekt ist. Fast alle Grafikkarten (seit Nvidia Geforce bzw. ATI Radeon) können dieses Verfahren hardwarebeschleunigt mit einer T&L-Einheit berechnen. Erst durch die Einführung von Vertex- und Pixelshaderprogrammen wurde es möglich auch andere Modelle hardwarebeschleunigt einzusetzen. In meiner Arbeit verwende ich das Phong-Lighting-Modell für die HDR-Lichtberechnung.
Das Phong-Lighting-Modell besteht aus drei Komponenten, die jeweils einen Lichtbeitrag zum Endergebnis einbringen:

$$\mathbf{i}_{phong} = \mathbf{i}_{dif} + \mathbf{i}_{spec} + \mathbf{i}_{amb}$$

5.1.1 Diffuse Komponente

Diffuses Licht entsteht durch die Reflektionseigenschaft einer matten Oberfläche. Man kann es sich so vorstellen, dass ein Photon, das von einer Lichtquelle abgestrahlt wird auf die Oberfläche trifft und dort absorbiert wird. Abhängig von der Farbe der Oberfläche wird das Photon entweder reflektiert oder geschluckt. Da man von einer matten Fläche ausgeht, werden die reflektierten Photonen gleichmäßig in alle Richtungen abgegeben. Somit ist die Stärke des emittierten Lichtes bestimmt vom Kosinus des Winkels der Oberflächennormalen n zum Lichtvektor l und nicht von der Kameraposition:

$$\mathbf{i}_{diff} = max((n \cdot l), 0) m_{diff} \otimes s_{diff}$$

Die Lichtfarbe s_{diff} und die Materialfarbe m_{diff} spielen insofern auch eine Rolle, sodass z.B. ein rotes Objekt nur rotes Licht reflektieren kann.

5.1.2 Specular Komponente

Spiegelndes Licht entsteht durch die Reflektion von glänzenden Oberfächen. Die Spiegelstellen, an denen der Betrachter sozusagen eine Reflektion der Lichtquelle sieht, bezeichnet man als Highlight. Highlights ermöglichen die verbesserte Wahrnehmung von Wölbungen des betrachteten Objekts. Die Stärke des *specular light* wird bestimmt durch die Richtung des Reflektionsvektors und den Vektor zum Betrachter:

$$\mathbf{i}_{spec} = (r \cdot v)^{m_{shi}}$$

Der zusätzliche Exponent m_{shi} hat den Zweck die Größe des Highlights zu bestimmen. Je größer der Wert umso kleiner das Highlight. Da es recht aufwändig ist den Reflektionsvektor zu berechnen ($r = 2(n \cdot l)n - l$) verwendet man die Halfwayvektoroptimierung:

$$h = \frac{l + v}{||l + v||}$$

Vorher war das specular light maximal, wenn der Betrachtervektor dem Reflektionsvektor entsprochen hat. Nun hat man den normalisierten Zwischenvektor aus dem Lichtvektor und dem Betrachtervektor gebildet. Das specular light ist maximal wenn der Halfwayvektor der Oberflächennormalen entspricht:

$$\mathbf{i}_{spec} = max((n \cdot h), 0)^{m_{shi}} m_{spec} \otimes s_{spec}$$

5.1.3 Ambient Komponente

Die Ambient Komponente entspricht der indirekten Beleuchtung in einer Szene. Das sorgt dafür, dass Oberflächen, die nicht direkt von einer Lichtquelle angestrahlt werden, trotzdem nicht vollkommen schwarz sind:

$$\mathbf{i}_{amb} = m_{amb} \otimes s_{amb}$$

5.2 Environmentmapping

Environmentmapping wurde ursprünglich von *Blinn* und *Newell* bereits im Jahr 1976 entwickelt [2]. Ihre Idee war es, den Betrachtervektor an einem spiegelnden Objekt reflektieren zu lassen und mit dessen Richtung einen Lookup in einer *Environmentmap* durchzuführen. Die Reflektionsrichtung entspricht dem umgekehrten Vektor, des von der Umgebung eintreffenden Lichtstrahls, der vom Objekt zurückgeworfen wird. Ein Raytracer würde an dieser Stelle einen Reflektionsstrahl abschicken um den Schnittpunkt mit der nähesten Oberfläche zu finden. Die Reflektionsrichtung bestimmt man mit der Oberflächennormalen wie in Abbildung 5.1 zu sehen ist.

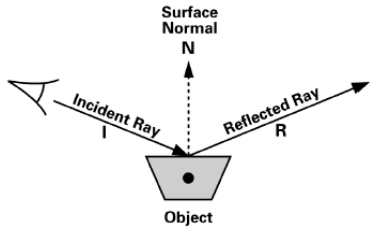


Abbildung 5.1: Reflektion eines Lichtstrahls an einem Objekt

Da Reflektionen in der Computergrafik eine sehr große Rolle spielen, gibt es dafür von Nvidia eine direkte Unterstützung in der Shadersprache CG. So genügen die folgenden Anweisungen um die Reflektionsrichtung eines Strahls zu berechnen:

```
1  void main (        float4  N     : TEXCOORD0, // object normal
2                     float4  I     : TEXCOORD1  // incident vector
3             ) {
4
5      // calculate Reflection Vector R
6      half3 R = reflect(I, N);
7
8  }
```

Der Vorteil des Environmentmapping ist es, dass man unmittelbar mit einem Lookup einen Farbwert erhält ohne weitere Berechnungen durchführen zu müssen. Das Verfahren hat aber auch ein paar Einschränkungen. Bei statischen Environmentmaps ist es nicht möglich, dass sich das Objekt selbst reflektiert, da jede Reflektion einen Farbwert aus der Map erhält. Weiterhin geht man davon aus, dass die Umwelt, die von der Environmentmap beschrieben wird, unendlich weit entfernt ist, da es sich bei der Map um etwas Statisches handelt, auch wenn man das Objekt bewegt. Das ist ein großer Nachteil, weshalb sich das Environmentmapping auch kaum für Szenen eignet, die sich verändern. Dennoch gibt es einige Ansätze die Maps erst zur Laufzeit zu bestimmen [10], indem man mit den Positionen der Objekte des vorhergenden Frames eine Environmentmap erzeugt.

5.3 Shadowmapping

Schatten bringen Realismus! Wer schon einmal ein gerendertes Bild mit und ohne Schatten gesehen hat, der kann das zweifelsfrei bestätigen. Aus diesem Grund wurden zahlreiche Algorithmen entwickelt, um mit der Rastergrafik Schatten darzustellen. Einer der prominentesten ist das Schadowmapping, entwickelt 1978 von *Lance Williams* [24]. Es ist ein Verfahren, das zwei Renderdurchläufe benötigt, wie man in Abbildung 5.2 sieht.

Im ersten Durchgang setzt man eine virtuelle Kamera an die Position der Lichtquelle und rendert die Szene von dort aus mit den Eigenschaften der Lichtquelle. Die dahinter steckende Idee ist, dass man von der Lichtquelle ausgehend all das sieht, was auch beleuchtet wird. Ist etwas im Schatten, dann wird es nicht von der Lichtquelle angestrahlt. Man unterscheidet drei Typen von Lichtquellen:

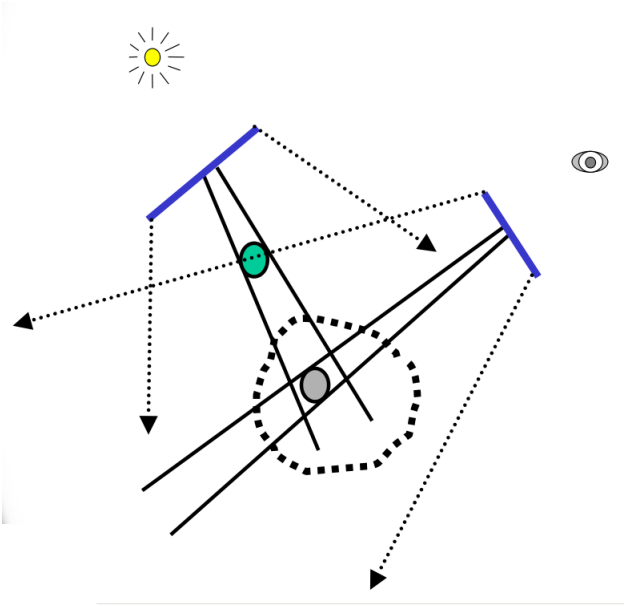


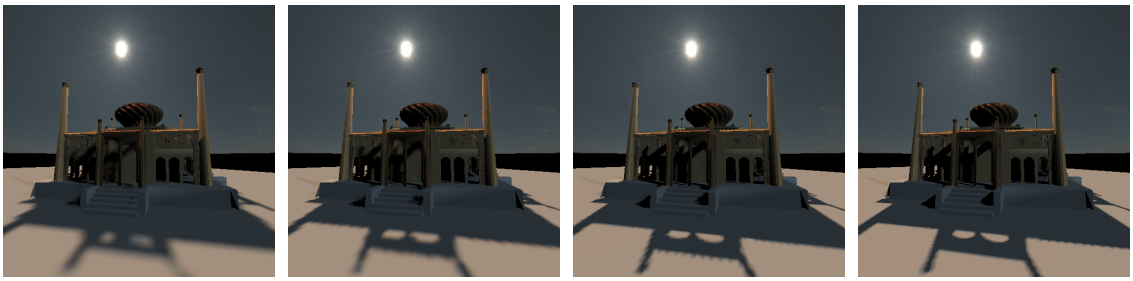
Abbildung 5.2: Skizze zur Funktionsweise des Shadowmapping

- **Parallele Lichtquelle**
  Für eine parallele Lichtquelle stellt man eine orthogonale Projektion ein.
- **Scheinwerferlichtquelle**
  Für ein sog. *Spotlight* stellt man eine perspektivische Projektion ein. Der Öffnungswinkel des Frustrums entspricht dabei dem Abstrahlwinkel des Lichts.
- **Punktlichtquelle (omnidirektional)**
  Hierfür werden im Regelfall sechs Spotlights verwendet, die jeweils eine eigene Shadowmap erhalten. Man stellt sich das so vor, dass man einen Würfel um die Position der omnidirektionalen Lichtquelle legt und durch jede Würfelseite ein Spotlight strahlen lässt. Damit erreicht man eine gute Auflösung der Shadowmaps, muss aber später für jeden Pixel berechnen auf welche Shadowmap man zugreifen muss.

Der Rendervorgang der Schadowmap erfolgt mit deaktiviertem Colorbuffer ausschließlich in den Tiefenpuffer (auch *depth-buffer, z-buffer*) der Grafikkarte, da man lediglich an den Tiefenwerten interessiert ist. Das ist auch der Grund warum sie auch als *shadowdepthmaps* bezeichnet werden. Hierfür kann man auch ein FBO (s. Kapitel 3.5.2) verwenden, insofern man Nvidia Treiber in der Version 80+ verwendet, die leider zur Bearbeitungszeit dieser Arbeit nicht mehr erschienen sind. Anschließend bindet man die Shadowmap als Textur und rendert damit die Szene aus Betrachtersicht erneut. Bei jedem gerenderten Pixel überprüft man nun, ob man sich im Schatten befindet oder nicht. Das funtkioniert so, dass man den Tiefenwert des aktuell gerenderten Pixels mit dem Tiefenwert in der Schadowmap vergleicht. Dieser Vorgang setzt voraus, dass man den Tiefenwert des aktuellen Pixels in das Koordinatensystem der Lichtquelle transformiert hat. Entsprechen die Tiefenwerte in etwa einander, dann steht fest, dass man von der Lichtquelle angestrahlt wird. Eine Shadowmap ist

vom Betrachterstandpunkt unabhängig und muss nicht neu berechnet werden, solange sich die Szene oder die Position der Lichtqelle nicht verändert.

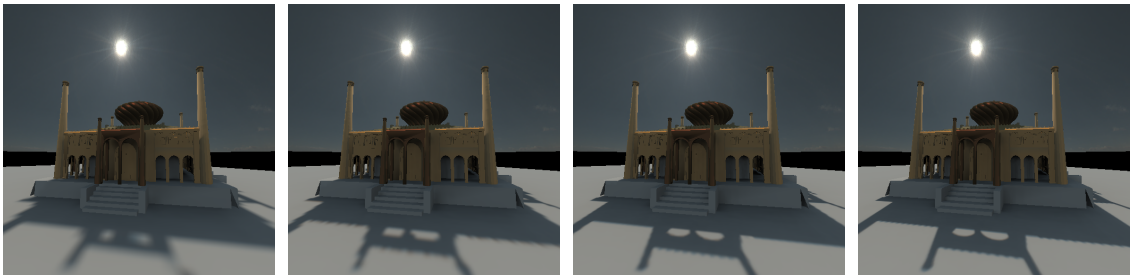
Leider hat das Verfahren auch einige Schwächen. Ein erstes Problem stellt die Umrechnung des Shadowmap-Tiefenwertes dar. Das ist im Allgemeinen nicht so präzise möglich. Deshalb kann es passieren, dass durch Rundungsfehler ein Punkt eines angestrahlten Objektes versehentlich als “im Schatten liegend” erkannt wird. Das Problem kann man mit Hilfe eines Toleranzwertes lösen, sollte dabei aber aufpassen, dass man ihn nicht zu groß wählt, sonst kehrt man das Problem um. Ein weiterer Nachteil des Schadowmap-Algorithmus ist die feste Größe einer Shadowmap. Es können große Bereiche im dargestellten Bild entstehen, die mit einem kleinen Teil der Schadowmap abgedeckt werden. Das Problem entsteht immer dann, wenn sich das Objekt, das den Schatten wirft, zwischen dem Betrachter und der Lichtquelle befindet und der Winkel zwischen Betrachter- und Lichtvektor sehr groß ist. Man sieht in solchen Bereichen unschöne Aliasing Artefakte. Ein kleines Trostpflaster stellt das *percentage closer filtering* dar, das von aktuellen Grafikkarten hardwarebeschleunigt unterstützt wird. Damit erzielt man einen leichten Softshadoweffekt und verhindert die harten Treppenstufen. Diesen Effekt kann man sehr gut in Abbildung 5.3 bei der niedrigsten Auflösung erkennen.



(a) 128x128 (b) 256x256 (c) 512x512 (d) 1024x1024

Abbildung 5.3: Einfluss der Schadowmap-Größe auf die Bildqualität unter der Verwendung von 7 Lichtquellen.

In meinem Programm verwende ich auch Shadowmaps zur Schattenberechnung. Die gerenderte Szene enthält sehr viele Lichtquellen, für die jeweils ein Einzelbild erzeugt wird. Anschließend werden diese Bilder miteinander verbunden. Allein durch die hohe Zahl der Einzelbilder verschwindet das Aliasing meist vollständig, wie man in den Abbildungen 5.4 erkennen kann.



(a) 128x128 (b) 256x256 (c) 512x512 (d) 1024x1024

Abbildung 5.4: Einfluss der Schadowmap-Größe auf die Bildqualität unter der Verwendung von 700 Lichtquellen.

5.4 Tone Mapping

Mit Tone Mapping bezeichnet man den Vorgang, die tatsächlichen Helligkeitswerte eines erzeugten Bildes an das Ausgabemedium anzupassen. Das Verfahren hängt davon ab, um welches Medium es sich handelt, wie es erscheinen soll und nicht zuletzt von der menschlichen Wahrnehmung. Eine Vorstellung verschiedener Algorithmen findet sich in [6].
Das Tone Mapping oder auch *Tone reproduction* soll nicht nur die Darstellbarkeit eines Bildes garantieren, sondern auch das Bild so erscheinen lassen, dass es für den Betrachter authentisch wirkt. Das ist eine große Herausforderung, da es die vollständige Kenntnis der Adaptionsprozesse des menschlichen Auges voraussetzen würde.

5.4.1 Lineares Mapping

Ein einfaches, in meiner Arbeit eingesetztes Verfahren, ist ein lineares Mapping:

$$Col_{out} = \frac{Col_{in}}{div}$$

Der Divisor wird bestimmt durch maximale Helligkeit eines Pixels im Bild oder etwas darunter, je nachdem welcher Helligkeitsbereich detaillierter dargestellt werden soll. Problematisch ist, dass man diesen hellsten Pixel nur sehr schlecht herausfinden kann, wenn sich das Bild als Textur im Speicher der Grafikkarte befindet. Deswegen schätzt man diesen mit Hilfe der Lichtberechnung. Einen guten Durchschnittswert für die Beleuchtung durch eine Environmentmap erhält man mit Hife der Monte Carlo-Integration (s. Kapitel 4.5). Die Abschätzung des Integrals multipliziert mit der Anzahl der Lichtquellen sollte in etwa der maximalen Helligkeit eines Pixels entsprechen. Das lineare Mapping führt zu relativ schlechten Ergebnissen, da es kaum der menschlichen Wahrnehmung entspricht und man sollte es nur als “minimales Tonemapping” verstehen.

5.4.2 Tonemapping mit Gammakorrektur

Das Problem des linearen Mappings ist es, dass helle Bereiche des Bildes nicht in dem Maße komprimiert werden, wie es der menschlichen Wahrnehmung entspricht. Deshalb ist es sinnvoll, die Details im mittleren Helligkeitsbereich zu verstärken, indem man eine Potenzfunktion anwendet:

$$Col_{out} = \frac{Col_{in}^{a}}{div}$$

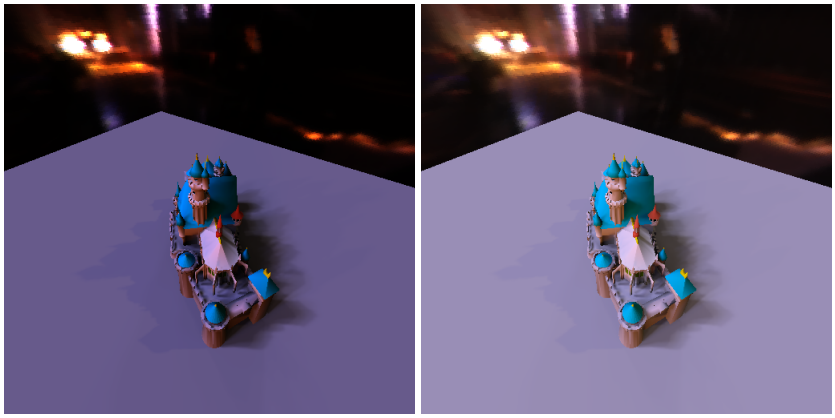
Für $a \epsilon [0..1]$ und einen angepassten Divisor erhält man relativ gute Ergebnisse, was mich dazu ermutigt hat, auch noch einen Bloom-Filter in die Implementierung mit einzubauen.

5.4.3 Bloom

Ein Bloom-Filter kann Bilder realistischer erscheinen lassen indem er sehr helle Stellen auf benachbarte Bereiche überträgt. Im Grunde ist es eine Art Weichzeichner für die Umgebung heller Bereiche. Dieser Effekt ist physikalisch nicht einfach erklärbar, kann aber praktisch nachempfunden werden. Sieht man Reflektionen einer sehr hellen Lichtquelle (z.B. der Sonne) dann scheint die nähere Umgebung auch überstrahlt zu sein, vor allem wenn die helle Stelle im Zentrum der Blickrichtung liegt. *Matt Pharr* und *Greg Humphreys* beschreiben in [21] einen Filter, der das ermöglicht:

$$f(x,y) = \left(1 - \frac{\sqrt{x^2 + y^2}}{d}\right)^4$$

Das d bestimmt die Breite des Filters. Wie man sieht ist er radialsymmetrisch und fällt nach außen hin ab. Wählt man x und y im Intervall $[-0.5..0.5]$ und möchte den Filter auf das gesamte Bild anwenden, kann man für d einfach 1 annehmen und spart sich eine Division. Meine Implementierung (s. Kapitel 6.4.2) verwendet diese Voraussetzung nicht, um maximale Flexibilität bei der Wahl der Parameter zu haben. Hat man mit $f(x,y)$ die Stärke es Effekts erhalten, wie ensteht dann der eigentliche Bloom-Effekt? Der Trick ist, dass man auf das Ergebnisbild einen Soft-Filter anwendet und dann mit einer Variablen $s \epsilon [0..1]$ zwischen den beiden Farbwerten interpoliert. Letztendlich multipliziert man noch mit $f(x,y)$ um den Effekt in der Mitte am stärksten auftreten zu lassen, da das Auge in Blickrichtung dafür am empfänglichsten ist.



(a) Linear Mapping

(b) Gamma corrected Mapping

(c) Gamma corrected Mapping mit Bloomeffekt

Abbildung 5.5: Das Schloss gerendert mit der *grace_probe* HDR-Environmentmap

Kapitel 6

Implementierung

In diesem Kapitel möchte ich einen Überblick über die Implementierung und deren Funktionsweise geben.

6.1 Die Renderarchitektur

Das Rendersystem besteht aus den folgenden Komponenten:

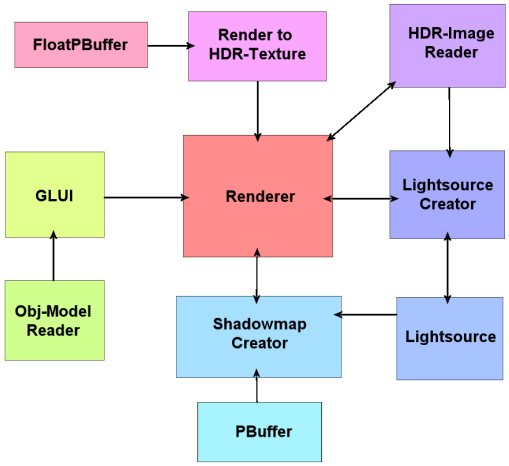


Abbildung 6.1: Schematische Darstellung des Rendersystems

Der *Renderer* wird von der Benutzerschnittstellenbibliothek *GLUI* mit Einstellungen gefüttert, die zu einer Statusveränderung im *Renderer* führen kann. Dieser bedient sich diverser Hilfsklassen um ein Bild zu erzeugen. Dafür liefert der *HDR-Imagereader* die Ausgangsbasis in Form einer HDR-Environmentmap. Der *Lightsource Creator* erzeugt daraus *Lightsources* mittels eines zuvor gewählten Verfahrens. Der *Schadowmap Creator* hat die Aufgabe zu einer *Lightsource* eine Shadowmap zu

erzeugen. Letztlich bleibt noch die *Render to Texture*-Einheit die dafür sorgt einen Off-Screen Buffer (FBO/FPBuffer) zu erzeugen und anschließend daraus eine Textur zu erstellen.

6.2 Wie entsteht ein fertiges Bild?

Die Ausgangsbasis bilden die mit einem Importance-Sampling oder mit einem Monte Carlo-Verfahren erzeugten Lichtquellen. Für jede Lichtquelle wird anschließend mit einem HDR-Lighting-Ansatz und unter Zuhilfenahme des Shadowmapping ein Einzelbild gerendert. Schematisch entspricht das folgendem Ablauf:

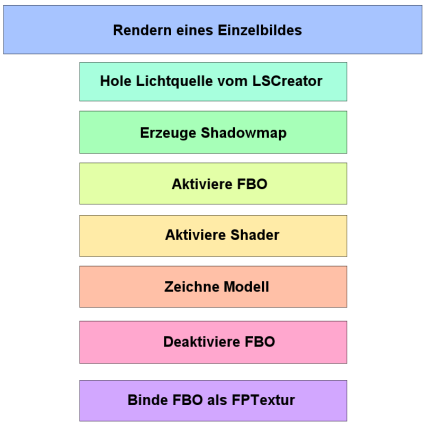


Abbildung 6.2: Schematische Darstellung des Rendervorgangs für ein Einzelbild

Anschließend werden diese Bilder kombiniert und mit einem Tonemapping-Verfahren zum Anzeigen aufbereitet. Die Trennung des Kombiniervorgangs vom Tonemapping ermöglicht ein progressives Rendering, das mit der Zeit sukzessive die Qualität verbessert:

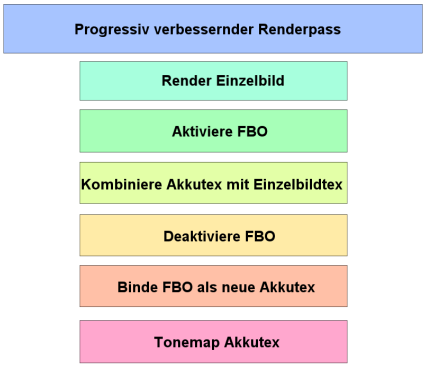
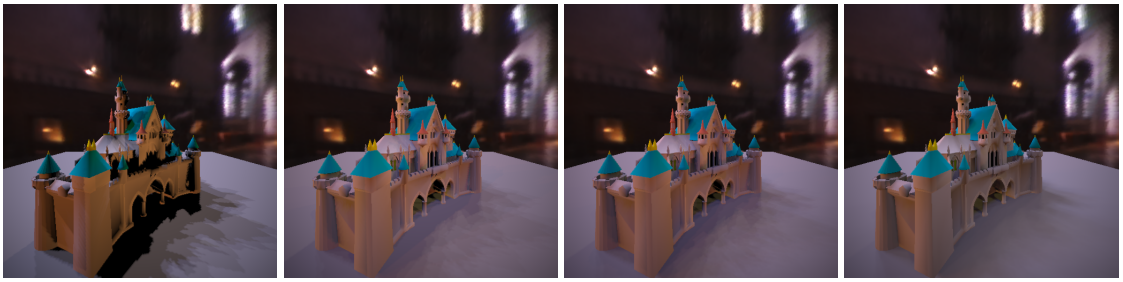


Abbildung 6.3: Schematische Darstellung des progressiven Rendervorgangs mit Tonemapping



(a) 7 Lichtquellen (b) 28 Lichtquellen (c) 56 Lichtquellen (d) 448 Lichtquellen

Abbildung 6.4: Progressiver Rendervorgang mit Zwischenergebnissen.

6.3 Lichtberechnung

Ich verwende das im vorherigen Kapitel (s. 5.1) beschriebene Phong-Lighting Modell. Der große Unterschied zum bekannten Vorgang ist, dass nun mit Lichtquellen, die Farbwerte größer als 1.0 haben, gerechnet wird. Zudem habe ich die Gelegenheit genutzt, die Materialeigenschaften um einen *emissive* Wert zu erweitern, damit es auch selbstleuchtende Objekte geben kann. Die Berechnung erfolgt in einem Vertexshader-Programm:

```
1  fragment main( vertex IN,
2
3                 // Modelview Projection Matrix
4                 uniform float4x4 modelViewProj,
5
6                 // Inverse Transposed Modelview Matrix
7                 uniform float4x4 modelViewInv,
8
9                 // Modelview Transformation matrix
10                uniform float4x4 modelView,
11
12                // Eye/Camera Position in Worldspace
13                uniform float4   eye,
14
15                // Light Position and Color
16                uniform Light    light,
17
18                // Material Paramters of Vertex
19                uniform gl_MaterialParameters Material,
20
21                // Global Ambient Color of Vertex
22                uniform float3   globalAmbient
23                )
24 {
25     fragment OUT;
```

```
26
27      // Transform vertex normal from model-space to view-space
28      float3 normal = normalize( mul( modelViewInv, IN.normal ).xyz );
29
30      // the lightvector is the direction to the light!
31      float4 lightpos = normalize( light.position );
32
33      // Calculate half angle vector
34      float3 half = normalize( lightpos.xyz + eye.xyz );
35
36      // Calculate diffuse component
37      float diffuse = max( dot( normal, lightpos.xyz ), 0.0f );
38
39      // Calculate specular component
40      float specular = max( dot( normal, half ), 0.0f );
41
42      specular = pow( specular, Material.shininess );
43
44      float3 ambient = Material.ambient * globalAmbient;
45
46      // Combine diffuse, specular, ambient and emissive contributions
47      OUT.hdrcolor.xyz = light.color * diffuse * Material.diffuse +
48                         light.color * specular * Material.specular +
49                         ambient + Material.emissive;
50
51      // calculate the homogenous position
52      OUT.position = mul( modelViewProj, IN.position );
53
54    return OUT;
55  }
```

Zu beachten ist dabei, dass man die berechnete Vertexfarbe nicht in der *COLOR* Variable des Shaders übergibt. Mir ist nach längerem Ausprobieren und einiger Verwunderung aufgefallen, dass der *COLOR*-Wert immer normalisiert wird!

6.4 Tonemapping

6.4.1 Linear Tonemap Shader

Das von mir verwendete Tonemapping basiert auf der Idee, dass der maximal zu erreichende Helligkeitswert eines Punktes des Objekts der Abstrahlstärke der Lichtquelle entspricht. Somit soll also ein Punkt, wenn er von allen Lichtqellen angestrahlt wird, im Ergebnisbild höchstens $(1.0, 1.0, 1.0)$ sein. Zu diesem Zweck speichere ich die kumulierten Farbwerte aller Lichtquellen, die an der beleuchteten Szene bisher beteiligt sind, und übergebe sie dem Tonemapping-Shader in der *multi* Variable. Dabei verwende ich den Kehrwert, da Divisionen rechenaufwändiger als Multiplikationen sind. Mit enthalten ist auch die gewählte *Exposure* Einstellung von der Applikation. Würde man dazu eine eigene Variable verwenden, hätte man eine unnötige Multiplikation mehr.

```
1  void main ( out half4 color2 : COLOR,
2              float4 TexCoord0 : TEXCOORD0,
3              uniform samplerRECT Tex0: TEX0,
4              uniform double multi
5              )
6  {
7      // Texture lookup to retrieve color value
8      double4 texc1 = texRECT(Tex0, TexCoord0.st);
9
10     float4 farbe= texc1 * multi;
11     farbe.w = 1.0;
12
13     // normalize color values!
14     farbe[0] = min(farbe[0], 1.0f);
15     farbe[1] = min(farbe[1], 1.0f);
16     farbe[2] = min(farbe[2], 1.0f);
17
18     color2 = farbe;
19 }
```

Das Pixelshader-Programm holt sich einen Farbwert aus einer Textur und teilt ihn durch den akkumulierten Farbwert der Lichtquellen. Anschließend werden die Farbwerte noch sicherheitshalber auf das zulässige Intervall $[0..1]$ überprüft.

6.4.2 Linear Bloom Tonemap Shader

Eine Verbesserung des Tonemappings kann erzielt werden, indem man einen Bloom-Filter mit einbaut um einen Überstrahleffekt von hellen Stellen in Blickrichtung zu erhalten.

```
1  void main ( out float4 color : COLOR,
2              float4 TexCoord0 : TEXCOORD0,
3              uniform samplerRECT Tex0: TEX0,
4              uniform float s,
5              uniform float d,
6              uniform float xres,
7              uniform float yres,
8              uniform double multi
9              )
10 {
11     double4 texc1 = texRECT(Tex0, TexCoord0.st);
12
13     float3 modifier = { -1, 0, 1};
14
15     // The soften matrix:
16     // ( 1 1 1 )
17     // ( 1 1 1 )
18     // ( 1 1 1 )
19
```

```
20      // Lookup filter values
21      double4 bloom1 = texRECT(Tex0, TexCoord0.st + modifier.xx);
22      double4 bloom2 = texRECT(Tex0, TexCoord0.st + modifier.yx);
23      double4 bloom3 = texRECT(Tex0, TexCoord0.st + modifier.zx);
24      double4 bloom4 = texRECT(Tex0, TexCoord0.st + modifier.xy);
25      double4 bloom6 = texRECT(Tex0, TexCoord0.st - modifier.zy);
26      double4 bloom7 = texRECT(Tex0, TexCoord0.st - modifier.xz);
27      double4 bloom8 = texRECT(Tex0, TexCoord0.st - modifier.yz);
28      double4 bloom9 = texRECT(Tex0, TexCoord0.st - modifier.zz);
29
30      // Calculate softened value
31      double4 softened = ( bloom1 + bloom2 + bloom3 + bloom4 + texc1
32                         + bloom6 + bloom7 + bloom8 + bloom9 ) * 0.111111111;
33
34      // Linear interpolation between original and softened color
35      float4 farbe = lerp( texc1, softened, s );
36
37      // Normalize TexCoordinates
38      TexCoord0.s /= xres;
39      TexCoord0.t /= yres;
40
41      // Shift coordinates to the interval [-0.5 .. 0.5]
42      TexCoord0.st -= 0.5;
43
44      // Calculate vignette effekt
45      float temp = 1 - ( dot ( TexCoord0.st, TexCoord0.st ) / d );
46      farbe *= pow( temp, 4);
47
48      farbe *= multi;
49      farbe.w = 1.0;
50
51      // Normalize color values!
52      farbe[0] = min(farbe[0], 1.0f);
53      farbe[1] = min(farbe[1], 1.0f);
54      farbe[2] = min(farbe[2], 1.0f);
55
56      color = farbe;
57  }
```

6.5 Klassenbeschreibung

Eine genaue Auflistung aller Methoden der Klassen und deren Funktion findet man in der Doxygen Quellcodedokumentation, die dem Programm beigefügt ist. Im Folgenden sind die von mir erstellten Klassen kurz charakterisiert.

6.5.1 Renderer

Die Renderer Klasse stellt das OpenGL-Rendersystem zur Verfügung und ist das Herz des Programms. Viel Wert wurde auf eine objektorientierte Programmierung gelegt, sodass die Klasse nicht vom verwendeten OpenGL Utility Toolkit (GLUT)[14] und der Benutzerschnittstellenbibliothek (GLUI) [22] abhängt. Sie verwendet ein zu Beginn initialisiertes Zustandssystem, damit keine undefinierten oder gar widersprüchlichen Werte für den Rendervorgang verwendet werden können. Der eigentliche Rendervorgang, die Kameratransformation, das Tonemapping und vieles mehr wird in dieser Klasse abgewickelt. Insbesondere hier lohnt der Blick in die Quellcodedokumentation.

6.5.2 ImageReader

Dies ist eine abstrakte Basisklasse für Bilddateien. Nur grundlegende Methoden zum Öffnen und Lesen von Datein sind darin implementiert.

6.5.3 HDRImageReader

Diese Klasse enthält einen File-Reader um .hdr-Dateien zu parsen und einlesen zu können. Abgeleitet ist sie von der Basisklasse ImageReader, mit deren Hilfe man weitere Dateiformate unterstützen könnte. Zusätzlich zur Datenpufferung der eingelesenen Datei unterstützt sie auch noch die Erstellung von OpenGL-Texturen in diversen Formaten.

6.5.4 Lightsource

Die Repräsentation einer Lichtquelle erfordert die Speicherung ihrer Farbe, Helligkeit, Art, Richtung und Position. Im Kontext auf die Verwendung des Shadowmapping war es sinnvoll, noch zusätzliche Informationen hierzu festzuhalten. So ist es möglich eine Lichtquelle zu kennzeichnen, wenn sie eine statische Shadowmap trägt. Mit dieser Information kann der Rendervorgang beschleunigt werden.

6.5.5 LSourceCreator

Ziel dieser Basisklasse für Samplingverfahren von Environmentmaps ist die Auswahl einer Stelle in der Map und die anschließende Erstellung einer entsprechenden Lichtquelle in Weltkoordinaten. Diese Klasse ist nicht abstrakt und kann instanziiert werden. Die Idee dahinter war eine Standardklasse für Testzwecke zu erhalten. Erstellt man Lichtquellen, haben diese alle die Farbe *(1.0f, 1.0f, 1.0f)*. Abgeleitete Klassen sollen ein Sampling-Verfahren implementieren. Eine weitere wichtige Funktionalität ist die Verwaltung der erstellten Lichtquellen, damit diese für jeden Rendervorgang immer die Gleichen sind. Das Puffern von Shadowmaps wird auch von dieser Klasse realisiert. Sie kennzeichnet entsprechende Lichtquellen und verwaltet die Texturen. Zusätzlich zu den Shadowmaps

kann diese Klasse bei entsprechendem Eintrag einer HDR-Environmentmap diese als LL-Map OpenGL-Textur ausgeben.

6.5.5.1 LSourceCreatorRandom

Das hier implementierte Samplingverfahren entspricht dem Monte Carlo-Verfahren aus Kapitel 4.5.3. Es wird gleichmäßig über die Kugeloberfläche zufällig ein Richtungsvektor ermittelt und dessen entsprechende Stelle auf der Environmentmap bestimmt. Die Position und der resultierende Farbwert wird für die Erstellung einer Lichtquelle verwendet.

6.5.5.2 LSourceCreatorSobol

Diese Klasse erzeugt mit Hilfe eines Quasi Monte Carlo-Verfahrens einzelne Samples auf der Kugeloberfläche. Der Unterschied zum Random- (Poisson-) Verfahren ist, dass die Samples gleichmäßiger verteilt werden und somit die Varianz des Ergebnisses etwas reduziert werden kann.

6.5.5.3 LSourceCreatorIALSIS

Diese Klasse implementiert das Importancesampling-Verfahren von *Matt Pharr* und *Greg Humphreys* [20] mit dem Namen “Infinite Area Light Source with Importance Sampling”. Im Konstruktor der Klasse werden einige STL-Vektoren erzeugt, die Funktionswerte für der Wahrscheinlichkeitsdichtefunktion repräsentieren. Aus diesem Grund kann es bei der Verwendung größerer Environmentmaps zu einem entsprechend hohen Speicherverbrauch kommen. Fordert man eine Lichtquelle von dieser Klasse an, verwendet sie das im Kapitel 4.5.5 beschriebene Samplingverfahren.

6.5.6 RendertoTex

In eine Textur rendern zu können ist eine wichtige Voraussetzung für HDR-Rendering. Dies wurde mit Hilfe zweier verschiedener Verfahren in dieser Klasse realisiert. Die Texturen werden entweder mit einem FPBuffer oder mit Framebuffer Objects erzeugt. Das Verfahren kann man mit dem Konstruktor bestimmen. FPBuffer bringen generell keinen Vorteil mehr, sodass sie nur noch zu Testzwecken implementiert sind.

6.5.7 Shadowmap_LS

Shadowmapping war einer der Ausgangspunkte dieser Arbeit und ist in dieser Klasse implementiert. Dies ist eine abgewandelte Version der Klasse wie sie in den InCG-Übungen 2004 verwendet wurde. Man hat mit ihr die Möglichkeit eine Shadowmap zu erzeugen und zum Rendern zu binden.

6.5.8 Hilfsklassen

Einige weitere Klassen sind hilfreich, haben aber keine bedeutende Rolle in der Funktion.

6.5.8.1 Vector

Um einen Punkt in Weltkoordinaten darstellen zu können benötigt man Vektoren. Um einen gewissen Komfort bei der Berechnung zu erhalten, wurde diese Klasse, die aus der Lehrveranstaltung GraPA stammt, mit einbezogen.

6.5.8.2 Matrix

Matrix-Vektor Multiplikationen spielen eine große Rolle bei der Transformation von Punkten im dreidimensionalen Raum. Diese Klasse ermöglicht dies und kennt darüber hinaus noch weitere Methoden.

6.5.9 OpenGL Hilfsklassen

Um sich das Leben im Umgang mit OpenGL etwas zu erleichtern erschien, es mir sinnvoll einige Teilfunktionen in eigenen Klassen zu realisieren.

6.5.9.1 FloatPBuffer

Ein Floatingpoint Off-Screenbuffer wird für HDR-Rendering benötigt. Diese Klasse bietet die Möglichkeit einen solchen zu erzeugen.

6.5.9.2 TexRectRenderer

Diese Klasse verwaltet eine Textur und kann sie mit Hilfe einer orthogonalen Projektion und einer Displayliste rendern.

6.5.9.3 CGContext / CgProgram

Pixel- und Vertexprogramme sind ein wesentlicher Teil meiner Arbeit. Mit diesen Hilfsklassen ist es möglich zur Laufzeit des Programms CG-Programme auf die Grafikkarte zu laden und auszuführen.

Kapitel 7

Optimierungen und Benchmarks

Im Laufe der Implementierungsphase sind mir Optimierungen eingefallen, wie man den Renderprozess hinsichtlich der Interaktivität und des Durchsatzes beschleunigen kann. Dieses Kapitel soll eine Evaluierung dieser Ideen sein. Alle in diesem Kapitel erscheinenden Benchmarks wurden auf einem AthlonXP 2400+ und einer Nvidia Geforce 6800 unter Debian/Linux gemessen.[1] Sofern nicht anders angegeben, sind folgende Einstellungen gewählt:

Shadowmap-Auflösung	512x512
Shadowmap Caching	Aus
Multi-Textur Optimierung	Aus
Multi-Shadowmap Optimierung	Aus
Boundingsphere	An
Reflektionen	Aus
Tonemapping	Linear

Tabelle 7.1: Einstellungen für das Benchmarking.

Das verwendete Schloss-Modell besteht aus 6600 Eckpunkten, das Raumschiff aus 127000 Eckpunkten.

7.1 Interaktivität

Interaktivität ist eine zentrale Eigenschaft dieser Anwendung. Gute Ergebnisse liefern bereits aktuelle Ray-Tracing-Systeme, im Gegensatz hierzu soll das Programm auch ohne einen Rechnercluster in Echtzeit bedienbar sein; zu diesem Zweck sollte man periodisch Benutzereingaben abfragen, indem der Renderer regelmäßig die Kontrolle abgibt und einen ”Zwischenstand” anzeigt. Erfolgen keine Eingaben, wird auf dem bestehenden Zwischenstand weitergearbeitet, ansonsten werden bisherige Ergebnisse verworfen, die keine Gültikeit mehr besitzen und man beginnt von Neuem. Im Gegensatz zu den Benchmarks unter dem *Durchsatz*-Aspekt sind diese eher ungenau. Das liegt daran, dass meine Implementierung keine fest vorgegebenen Kamerapfade unterstützt und die Benutzereingaben

[1] AthlonXP2400+@2300Mhz, Nvidia Geforce 6800@16/6 und 350Mhz mit 128MB@400Mhz

zeitabhängig sind. Somit ist es nicht möglich einen Kamerapfad exakt zu replizieren. Die zu erwartenden interaktiven Bildraten (FPS = frames per second) sollten ein gewisses Mindestniveau haben, damit man die Applikation nicht als ruckelig empfindet. Ich habe versucht, dies hauptsächlich mit den folgenden zwei Optimierungen zu erreichen:

7.1.1 Shadowmap Caching

Die Idee des Shadowmap Caching lag nahe und war praktisch ohne großen Aufwand implementierbar. Diese Optimierung beruht auf der Tatsache, dass Shadowmaps nur dann neu gebaut werden müssen, wenn sich die Szene bzw. die Lichtquellenposition ändert. Wenn lediglich die Kamera bewegt wird, bleiben die Shadowmaps gleich. Ein Nachteil der Methode ist, dass nur eine begrenzte Anzahl von Shadowmaps gecached werden kann, da der Speicher der Grafikkarte limitiert ist. Die besten Ergebnisse erzielt man, wenn so viele Shadowmaps zwischengespeichert werden, wie für das erste anzuzeigende Bild benötigt werden. Das Caching zahlt sich aus, wenn das Modell sehr komplex

	Cached	Non-C	Cached	Non-C	Cached	Non-C	Cached	Non-C
Shadowmap Auflösung	**128**		**256**		**512**		**1024**	
Schloss	175	150	175	135	175	125	175	82
Voyager	50	37	48	37	47	37	47	31

Tabelle 7.2: Benchmark Shadowmap Caching. Gemessen wurden FPS.

ist, oder eine hohe Shadowmap-Auflösung gewählt wurde. An diesem Benchmark kann man gut erkennen, dass der Rendervorgang des Modells wohl nicht der alleinige Flaschenhals ist, sonst würde der Wegfall eines kompletten Renderdurchlaufs für die Shadowmap einen noch größeren Ausschlag bringen.

7.1.2 FBO's

Wie ich bereits im Kapitel 3.5.2 beschrieben habe, bringt alleine die Umstellung von FPBuffern auf FBO's einen Performancevorteil von bis zu 50 % in der Gesamtdurchlaufzeit. Ähnlich verhält es sich mit der interaktiven Bildrate:

	FBO	PBuf	FBO	PBuf
Auflösung	**512x512**		**1024x768**	
Schloss	125	66	62	27
Voyager	37	30	30	18

Tabelle 7.3: Benchmark FPBuffer vs. FBO. Gemessen wurden FPS.

Der Performanceeinbruch mit FPBuffern ist dramatisch und liegt bei diesem Beispiel z. T. über 50 %.

7.2 Durchlaufzeit

Die zweite Herangehensweise ist der Versuch, die Rendergesamtzeit mit einer bestimmten vorgegebenen Anzahl von Lichtquellen zu verkürzen. Um das zu erreichen, kann man bestehende Hardwarefunktionen ausnutzen und die Anzahl der gezeigten Zwischenstände, was auch Zeit kostet, verringern.

7.2.1 Multitexturing

Die Multitextur-Optimierung besteht bereits seit dem ersten Entwurf der Implementierung. Unter der Verwendung der OpenGL-Erweiterung *GL_ARB_multitexture* kann man einfach mehrere Texturen binden und sie später mit einem Shader in einem Renderdurchlauf gleichzeitig verarbeiten. Dazu wird zunächst für sieben unterschiedliche Lichtquellen ein Einzelbild erzeugt und die Texturen inklusive der Akkumulationstextur vom vorhergehenden Durchlauf an die einzelnen Textureinheiten gebunden. Das Ergebnis des Rendervorgangs wird anschließend in eine neue Akkumulationstextur kopiert. Statt sieben einzelnen Renderdurchgängen zur Akkumulation hat man nunmehr einen! Das spart Zeit, wie folgende Tabelle belegt:

	1xTex	7xTex	1xTex	7xTex
Auflösung	**512x512**		**800x600**	
Schloss	13,9	9,1	20,6	13,0
Voyager	43,7	38,9	49,9	42,8

Tabelle 7.4: Benchmark Multitexturing. Gemessen wurde die Durchlaufzeit in [sec] für 1480 Lichtquellen.

Die Optimierung bringt einen geringeren Performancevorsprung als ich erwartet hatte. Zudem entstehen neue Limitierungen, die man nicht vernachlässigen sollte. Dabei geht es um den Speicherverbrauch, der durch die 32-Bit/Farbkanal Texturen erheblich anwächst. Eine solche RGB-Textur mit einer Auflösung von 512 x 512 Punkten benötigt einen Speicherplatz von genau 3 MB. Im Multitexturing Fall entsteht somit der folgende Speicherplatzbedarf:

	Anzahl	Größe für 1	Gesamtgröße
32 Bit Tex Einzelbild	7	3 MB	21 MB
32 Bit Akkutex (alt)	1	3 MB	3 MB
32 Bit Akkutex (neu)	1	3 MB	3 MB
FBO's RGB+DepthBuffer	9	3,75 MB	33,75 MB
Gesamt			60,75 MB

Tabelle 7.5: Multitexturing Speicherplatzverbrauch.

Rechnet man noch die Displaylisten, Shadowmaps, usw. hinzu, stößt man schnell an die Grenzen von 128MB Grafikkartenspeicher. Aus diesem Grund musste ich die Benchmarks mit einer Auflösung von 800 x 600 durchführen, da bei 1024 x 768 bereits dieses Problem auftritt.

7.2.2 Multi-Shadowmapping

Nach der kleinen Ernüchterung mit vorhergehender Optimierung, kam mir die Idee, dass man im Shader mehrere Shadowmaps gleichzeitig abarbeiten könnte. Auf Grund der limitierten Anzahl von Texturkoordinaten, ist meine Implementierung auf die Behandlung von drei Shadowmaps in einem Durchlauf beschränkt. Nun zu den Ergebnissen:

	1xSM/pass	3xSM/pass	3xSM/pass + 7xTex	1xSM/pass	3xSM/pass
Auflösung	**512x512**			**1024x768**	
Schloss	13,9	6,5	5,2	32,5	12,8
Voyager	43,7	22,6	21,3	120,0	64,6

Tabelle 7.6: Benchmark MultiShadowmapping. Gemessen wurde die Durchlaufzeit in [sec] für 1480 Lichtquellen.

Das ist eine schöne Optimierung, die noch einmal klar zeigt, dass der Flaschenhals nicht bei den Shaderoperationen liegt, sondern eher bei der *copy to texture* Operation zu suchen ist.

Kapitel 8

Abschlussbesprechung und Ausblick

8.1 Ergebnisse

Mit dieser Arbeit wurde gezeigt, dass hardwarebeschleunigtes Rendern in hoher Qualität gute Ergebnisse liefert. Der verwendete Importance-Sampling-Algorithmus ermöglicht selbst mit wenigen gewählten Lichtquellen eine gute Approximation der Beleuchtungssituation. Die interaktive Benutzerschnittstelle erlaubt jederzeit die Kontrolle über die Kamera, um das zu beleuchtende Objekt von allen Seiten begutachten zu können. Rendergeschwindigkeit und Qualität lassen folgende Schlüsse zu:

8.1.1 Rendergeschwindigkeit

Das ist wohl der Hauptvorteil des Systems. Durch einige Optimierungen an den Shadern und das Shadowmap-Caching-Verfahren ist das zu rendernde Objekt selbst bei hohen Polygonzahlen mit interaktiven Bildwiederholraten darstellbar. Die Benchmarks in Kapitel 7 belegen das eindrucksvoll. Ermöglicht wird die Geschwindigkeit durch das progressive Rendersystem.

8.1.2 Renderqualität

Im Vergleich mit einem flexiblen Raytracersystem kann die gebotene Qualität nicht mithalten. Die Schattenberechnung mit Shadowmaps wirkt vergleichsweise antiquiert. Bei ungünstigen Szenen erreicht man erst mit einer hohen Anzahl von Lichtquellen weiche Schatten. Andere visuelle Effekte, etwa die Reflektionen müssen in der Rastergrafik immer mit Tricks implementiert werden und sind daher meist beschränkt. Das verwendete Multipassverfahren erzeugt im Vergleich mit herkömmlichen einfachen Beleuchtungsalgorithmen eine sehr hohe Qualität, die mit steigender Samplezahl immer besser wird. Letztendlich entscheidet der Einsatzzweck, ob hohe Qualität oder Geschwindigkeit gefragt ist.

8.2 Zukünftige Erweiterungen

Es wäre sicherlich interessant eine Umsetzung des Programms für die Windows-Plattform zu implementieren. Die Treiberprogrammierung für das Windowsbetriebssystem scheint seitens der

Hardwarehersteller deutlich besser zu sein. Ein direkter Performancevergleich der beiden Systeme interessiert mich sehr.

Weitere denkbare Erweiterungen:

- Das Tonemappingverfahren entscheidet maßgeblich über die Wirkung des Ergebnisses. Hier könnte man andere Verfahren ausprobieren, evtl. auch mit zuvorigem Zurücklesen und Analyse des Bildes.
- Da einige Verdachtsmomente aufkamen, dass die copy to texture Implementierung den größten Flaschenhals darstellt, wäre es sicherlich interessant zu sehen, wie sich sowohl Qualität als auch Performance mit einem anderen Schattenalgorithmus, etwa mit Shadowvolumes, entwickeln.
- Der Effektumfang des Programms könnte um Transparenz erweitert werden. Dies würden eine weitere Portion Realismus hinzufügen.

Anhang A

Screenshots

Um die Qualität des Rendersystems zu demonstrieren, habe ich einige hochauflösende Screenshots zusammengestellt:

Abbildung A.1: *Vamp*-Modell mit der *uffizi_probe*-HDR-Environmentmap. Die von oben einfallenden Lichtstrahlen produzieren eine gleichmäßige Ausleuchtung. In diesem Bild kommen besonders die Reflektionen gut zur Geltung.

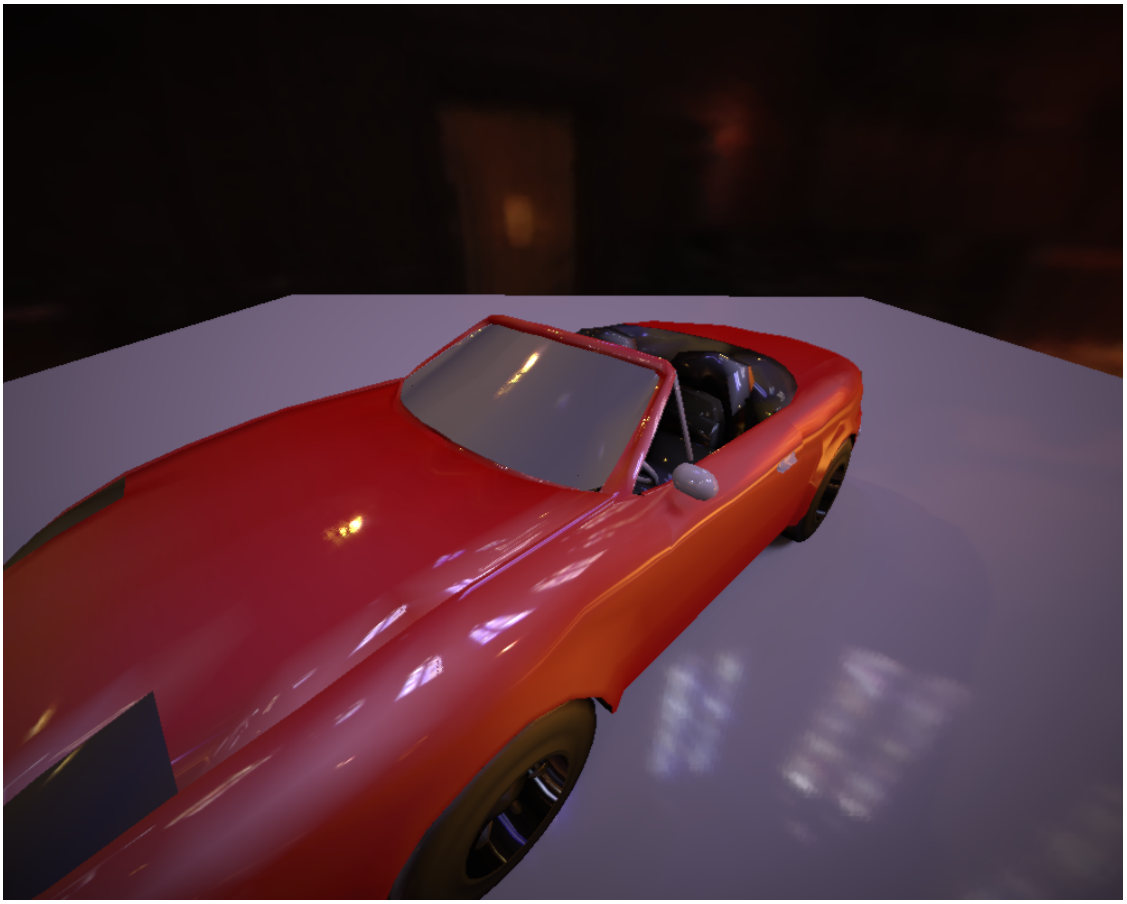


Abbildung A.2: *Roadster*-Modell mit der *grace_probe*-HDR-Environmentmap. Die weichen Farbverläufe auf der Motorhaube entstehen durch die Vielzahl der einflussnehmenden Lichtquellen. Der leuchtende Altar produziert die gelben Reflektionen auf der Fahrerseite des Autos.

Abbildung A.3: *Roadster*-Modell mit der *grace_probe*-HDR-Environmentmap. Dieses Bild wurde mit den gleichen Einstellungen erzeugt wie A.2. Es zeigt das Auto von der anderen Seite. Auf der Bodenfläche kann man die soften Schatten sehen, die durch die zahlreichen Fenster der Kathedrale erzeugt werden. Die Renderzeit für die Szene beträgt 43,5 Sekunden ohne Optimierungen.

Literaturverzeichnis

[1] S. Agarwal, R. Ramamoorthi, S. Belongie, and H. Jensen. Structured importance sampling of environment maps, 2003.

[2] J. F. Blinn and M. E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[3] P. Debevec and J. Malik. Recovering high dynamic range radiance maps from photographs, 1997.

[4] Paul Debevec. http://www.debevec.org/probes/.

[5] Paul Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. *Computer Graphics*, 32(Annual Conference Series):189–198, 1998.

[6] Kate Devlin. A review of tone reproduction techniques.

[7] Manfred Ernst. Photo-realistic rendering on programmable graphics hardware. Diploma thesis, University of Erlangen-Nuremberg, Erlangen, July 2003.

[8] C. Everitt, A. Rege, and C. Cebenoyan. Hardware shadow mapping, 2002.

[9] James A. Ferwerda, Sumanta N. Pattanaik, Peter Shirley, and Don Greenberg. A model of visual adaptation for realistic image synthesis. In *SIGGRAPH 1996*, pages 249–258, 1996.

[10] Ron Fosner. All aboard hardware t and l. *Game Developer*, 7(4):30–41, April 2000.

[11] Rob Shakespeare Greg Ward, Larson Shakespeare. *Rendering With Radiance: The Art And Science Of Lighting*. Booksurge Llc, 2004.

[12] A. Keller. Quasi-monte carlo methods in computer graphics: The global illumination problem, 1995.

[13] Alexander Keller. *Quasi-Monte Carlo Methods for Photorealistic Image Synthesis*. PhD thesis, University of Kaiserslautern, Kaiserslautern, June 1997.

[14] Mark J. Kilgard. *The GLUT 3.5 Manual Pages*. SGI, 1997.

[15] W. Morokoff and R. Caflisch. Quasi-monte carlo integration, 1995.

[16] William J. Morokoff and Russel E. Caflisch. Quasi-random sequences and their discrepancies. *SIAM Journal on Scientific Computing*, 15(6):1251–1279, 1994.

[17] Tomas Möller and Eric Haines. *Real-Time Rendering*. A K Peters, Natick, Massachusetts, 1999.

[18] NVIDIA. *Cg Toolkit User's Manual*, December 2002.

[19] NVIDIA. *NVIDIA GPU Programming Guide*, August 2005.

[20] Matt Pharr and Greg Humphreys. Infinite area light source with importance sampling, 2004.

[21] Matt Pharr and Greg Humphreys. *Physically Based Rendering*. Morgan Kaufmann Publishers, 2004.

[22] Paul Rademacher. *GLUI - A GLUT-Based User Interface Library*, June 1999.

[23] Arne Seifert. Das floating-point-format im detail.

[24] Lance Williams. Casting curved shadows on curved surfaces. In *SIGGRAPH 1978*, pages 270–274, 1978.

[25] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison-Wesley, Reading, Massachusetts, second edition, 1999.

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass die Arbeit veröffentlicht wird und dass in wissenschaftlichen Veröffentlichungen auf sie Bezug genommen wird.

Der Friedrich-Alexander-Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Graphische Datenverarbeitung, wird ein (nicht ausschließliches) Nutzungsrecht an dieser Arbeit sowie an den im Zusammenhang mit ihr erstellten Programmen eingeräumt.

Erlangen, 30. September 2005

(Thomas Kemmer)