Echtzeit-Verfahren zur Beleuchtung einer computergenerierten 3D-Szene mit Videotexturen

Carsten Juttner

Fachhochschule Düsseldorf, Fachbereich Medien
Düsseldorf University of Applied Sciences, Department of Media

Betreuer:
Prof. Jens Herder, Dr. Eng./Univ. of Tsukuba
Koprüfer:
Prof. Thomas Bonse, Dr. Ing.

4. Januar 2005

Zusammenfassung

Der Autor stellt eine Methode vor, um einer computergenerierten Szene auf neue Art und Weise zusätzlichen Realismus zu verleihen. Er tut dies unter Erweiterung des traditionellen "festen" Shadingmodells durch Bildsequenzen (nachfolgend "Videotextur" genannt), welche die Oberflächen anderer Objekte innerhalb einer Szene in Echtzeit beleuchten.

Im Rahmen der Diplomarbeit wurde eine Beispielanwendung erstellt, in der eine vorbeiziehende Landschaft (Videotextur) auf den Innenraum eines computergenerierten Zuges (3D-Polygon-Geometrie) einen Beleuchtungseinfluß ausübt. Diese Integration von real gefilmtem Material und computergenerierten Bildern ist eine übliche Vorgehensweise bei Spezialeffekten für Film und Fernsehen, aber erst seit kurzem bietet die durchschnittliche PC-Grafikhardware entsprechende Möglichkeiten unter Echtzeitbedingungen an.

Um dieses Vorhaben umzusetzen wird umfangreicher Gebrauch der OpenGL Shader-Hochsprache gemacht, durch die ein Shaderentwickler in der Lage ist, mit einem C-ähnlichen Programm die Pixelberechnungsfunktionalität der Grafikkarte seinen Wünschen entsprechend anzupassen.

Abstract

The author presents a novel approach to add an additional degree of realism to a computer generated scene by using a sequence of images (called "video texture") to light up other objects in realtime. He does this by enhancing the traditional "fixed" shading model using the video textures as part of the light properties.

An exemplary application showcases this approach by using a filmed landscape (video texture) which affects the lighting of the interior of a computer generated train (3D polygonal geometry). Combining filmed material and computer generated images has long been standard practice in special effects departments but it was not until recently that the average graphics hardware is becoming capable to produce these effects in realtime.

To accomplish the given task the author makes extensive use of the OpenGL high-level shader language (GLSL) which lets the shader developer write C-like programs to alter the behaviour of the graphics card's pixel computing functionality which previously was fixed to only a limited set of operations.

Danksagung

Mein Dank gilt Professor Dr.-Ing. Ralf Wörzberger (Fachbereich Architektur) und dem PCM-Team für die freundliche Unterstützung und die Überlassung des Zugmodells, Antje Müller für ihre tatkräftige Unterstützung bei den Blauraumaufnahmen, Professor Jens Herder für die Betreuung der Diplomarbeit und die Unterstützung bei wissenschaftlichen Fragen.

Ein Dankeschön geht auch an Mercury Systems für die Möglichkeit der Evaluation einer Betaversion von OpenInventor 5.0.

Ebenfalls danke ich Helmut Sieber für die Mitwirkung an der Zugpassagierszene. Ein spezieller Dank gilt Detlef Roettger von NVidia für den Support und die Informationen zu GLSL- und hardwarespezifischen Fragen. Auch Steve Streeting und dem gesamten Ogre-Team (speziell Jeff Doyle für die Implementierung des Ogre-GLSL-Plugins) gebührt ein Dank für den tatkräftigen Support.

Schließlich möchte ich auch meinen Eltern für die unablässige Unterstützung während des gesamten Studiums und insbesondere während der Diplomarbeitsphase meinen ganz besonders tiefen Dank aussprechen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Motivation

Der Computergrafikbereich ist sicherlich einer der spannendsten und innovativsten Forschungsbereiche der Neuzeit. Die Grundlagenforschung der sechziger und siebziger Jahre schuf Algorithmen, welche in den Folgejahren verfeinert wurden. Gesteigerte Rechenleistung erlaubte komplexere Berechnungen, gleichzeitig ermöglichte die verbesserte Hardware immer höhere Auflösungen und flüssigere Darstellungen.

Durch die PC-Revolution der achziger und neunziger Jahre wurde die Eigenart der Halbleiterindustrie, in immer kürzerer Zeit immer höhere Leistungen umzusetzen für jedermann deutlich. Das viel zitierte Moore'sche Gesetz (1965 von Gordon Moore, Mitbegründer von Intel aufgestellt) besagt, daß sich die Anzahl der Transistoren auf einem Chip alle 18 bis 24 Monate verdoppelt, momentan bewegt sich diese Prognose im 18 Monats-Bereich und es wird erwartet, daß dies auch noch eine Weile so bleibt.

Diese Entwicklung spiegelte sich natürlich auch im Computergrafikbereich wieder. Sie mündete in der Fähigkeit, virtuelle, computergenerierte Umgebungen in Echtzeit (engl. *real time*) generieren zu können.

Um das dort auftretende Speicherbandbreitenproblem zu lösen (die Datenmenge für eine Auflösung von 1024x768 mit 85 Hz entsprechen bei 32-Bit bereits 255 MByte/sek) stattete man die Grafikkarten mit schnellem lokalen RAM aus und entwarf Chipsätze, um häufig verwendete Operationen zu beschleunigen, zum Beispiel das Kopieren von Blöcken (eine Anwendung dafür wäre das schnelle Verschieben von Fenstern auf der Desktopoberfläche).

Virtuelle Umgebungen stellen ebenfalls typische Anforderungen. Die dort gezeigten Objekte sollen möglichst frei im Raum bewegt werden können. Man begann daher bereits Ende der 80er Jahre, Prozessoren mit spezieller Unterstützung für 3D-Operationen zu konzipieren, einer der ersten Vertreter war 1989 der Intel i860 [GKB89]. Es handelte sich dabei aber nur um eine Unterstützung für die benötigten Rechenoperationen, er enthielt noch keinen Displaycontroller.

Die Geburtsstunde des ersten 3D-Beschleuniger für den Heimgebrauch fand im Jahre 1996 statt, als die Firma 3Dfx die "Voodoo 1" veröffentlichte [Ecc00]. Diese Karte besaß allerdings keinerlei 2D-Funktionen, so daß zusätzlich noch eine Grafikkarte benötigt wurde.

Überschlägt man die 3D-Grafikkarten-Entwicklung der darauf folgenden 8 Jahre, so stellt man fest, daß die Leistungszunahme selbst im Vergleich mit PC-CPUs gewaltig ist. 1998 enthielt der damals aktuelle Riva TNT-3D Chip von NVidia 7 Millionen Transistoren, heute besitzt der Grafikprozessor (GPU) NV40 der 6x00er-Systeme davon 222 Millionen (zum Vergleich: der Pentium IV Northwood ist mit nur 55 Millionen Transistoren ausgestattet). Dies übertrifft die Vorhersage des Moore'schen Gesetzes um fast das vierfache.

Die Optimierung brachte Geschwindigkeitssteigerungen, allerdings waren weite Teile

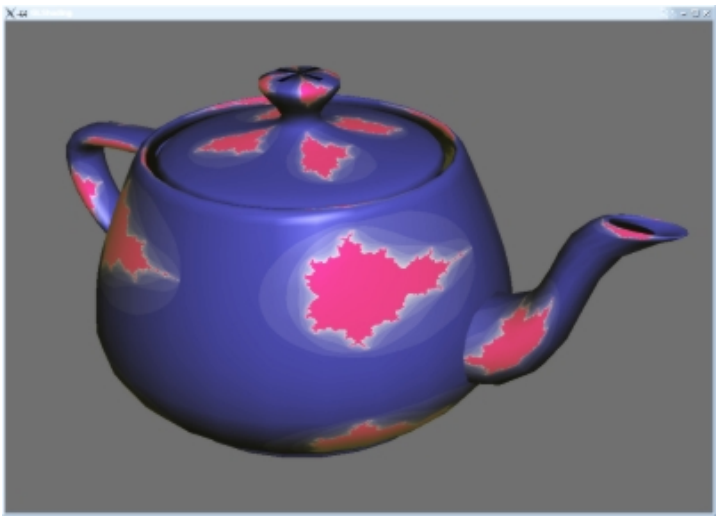

Abbildung 1.1.: Teekanne mit Fraktalshader

der Pixelberechnung fest definiert und konnten nicht den eigenen Bedürfnissen angepaßt werden. Um der Forderung nach größerer Freiheit und Flexibilität gerecht zu werden, begannen die Hersteller damit, Teile der GPU-Funktionalität programmierbar zu machen. Zunächst geschah dies nur in Form von herstellerspezifischen Assemblerprogrammen, später kamen dann Hochsprachen hinzu. Auf diese Art und Weise ist es möglich geworden, Karten verschiedener Hersteller auf Basis eines gemeinsamen Shader-Quelltextes zu programmieren.

Grafik-Algorithmen, die vor Jahren noch langwierige Berechnungen mit sich brachten sind nun in Echtzeit möglich (Prominentes Beispiel: die Mandelbrotmenge, auch liebevoll Apfelmännchen genannt, Abbildung 1.1). Dies zeigt deutlich, welche gewaltigen Rechenleistungen sich in der GPU verbergen.

Das Ausnutzen all dieser neuartigen Möglichkeiten war der Ansatz für die vorliegende Arbeit. Die GPU-Programmierung befindet sich derzeit noch in einer frühen Entwicklungsphase. Um so mehr ist es daher wichtig, in diesem Bereich Forschung zu betreiben, um Grundlagen für das Arbeiten mit dieser Technologie zu erhalten.

Für virtuelle Umgebungen ist neben der Echtzeitbedingung auch ein möglichst hoher Realitätsgrad gefordert (photorealistisch). Nur auf diese Weise kann eine vollständige Immersion in die Szene erfolgen. Der Autor hat in der vorliegenden Arbeit einen Ansatz dafür entwickelt und beschreibt die Realisierung zweier Komponenten, einmal der Integration von Videotexturen in eine Anwendung und der Programmierung von Shadern, um auf eben diese Texturen zugreifen zu können.

1.2. Verwendete Komponenten

Die Anwendung wurde durchgängig in C++ plattformunabhängig entwickelt, getestet wurde dies sowohl unter Linux Gentoo [Gen04] wie auch unter Microsoft Windows™ 2000/XP [Mic04]. Zur Kompilierung wurde auf Linuxseite der GNU-C++

Compiler [GCC04] verwendet, auf Windows-Seite kam zusätzlich der Microsoft Visual C++-Compiler in Version 6 zum Einsatz. Um die Konfiguration kümmerten sich bei den GNU-Compilern die GNU-Autotools [GNU04], beim Microsoft-Compiler geschah dies manuell über die Visual-C++-IDE. Für Skripte kamen die (Unix-)Standard Bourne Again-Shell und Python [Pyt04] zur Anwendung.

Da Windows keine Unix-kompatible Umgebung bietet wurde dort MinGW [min04] verwendet, welches eine minimalistische GNU-Umgebung nachrüstet. Dadurch konnte die Übersetzung des Quellcodes im Falle der GNU-Compiler auf beiden Systemen auf gleiche Art und Weise vorgenommen werden. Der Quellcode wurde dabei unter Linux mit Anjuta [Anj04], unter Windows mit Ultraedit [Ult04] bzw. der Visual-C++-IDE bearbeitet. Die endgültige Anwendung wurde auf Windows-Seite mit dem Microsoft-Compiler übersetzt, weil die zugrundeliegende 3D-Engine Ogre sich in der verwendeten Version 0.15.1 nicht zufriedenstellend in der MinGW-Umgebung kompilieren ließ[1].

Da der Texturcache unabhängig von der Darstellung puffern sollte, wurde er in einen eigenen Thread ausgelagert. Die dafür nötige Posix-Thread-Implementierung [PTh04] für die inkompatible Win32-Thread-API stammt von Redhat.

Die Low-Level-Schnittstelle zur Grafikkarte wurde durch die OpenGL-Bibliothek [Ope04a] bereitgestellt. Die Shaderprogramme wurden folgerichtig auch in GLSL geschrieben, der integrierten Shaderhochsprache von OpenGL.

Als Schnittstelle zwischen Anwendung und OpenGL wurde die 3D-Grafikengine Ogre [OGR04] verwendet. Für die Erstellung der Shader wurde zunächst der ATI Rendermonkey™ [Ren04] auf Windows-Plattform benutzt, später dann der OpenGL ShaderDesigner [Sha04] in einer Beta-Version für Linux. Die 3D-Objekte der Szene lagen bereits als Modelle für Alias Maya [May04] vor und wurden noch für die Anwendung optimiert. Sie entstanden im Rahmen eines interdisziplinären Forschungsprojektes [PCM04] an der FH Düsseldorf.

Bilder- und Texturenbearbeitung erfolgte durch Photoshop [Pho04] und TheGimp [GIM04]. Die Videotexturen für die Anwendung lagen in Einzelbildsequenzen vor, welche mit Hilfe der NVidia-Developer-Texture-Tools ins DXT-Format konvertiert wurden. Das Keying der Blauraumszenen erfolgte mit Discreet Combustion [Dis04], die Filmsequenzen wurden in After Effects [Pho04] aufbereitet.

Zur Erstellung der Diplomarbeit selber wurde LyX [LyX04] (basierend auf LaTeX) als Textprozessor verwendet. Grafiken und schematische Übersichten wurden mit Dia [Dia04] erzeugt, Diagramme mit Grace [Gra04].

1.3. Idee und Anwendung

Ausgangspunkt war die Idee, real gefilmte Videosequenzen in eine computergenerierte Szene zu integrieren. Um die Integration innerhalb der Szene noch zu erhöhen, sollte eine Sequenz als Beleuchtung der Szenenobjekte genutzt werden.

Als Umsetzung dieser Idee wurde die Simulation eines Zugmodells durchgeführt. Landschaft und Personen im Innenraum sollten dabei durch Videosequenzen dargestellt wer-

[1] Ogre ist eine reine C++-Bibliothek (DLL), das Linken einer MinGW-GCC-Anwendung gegen die MSVC-DLL ist daher nicht möglich.

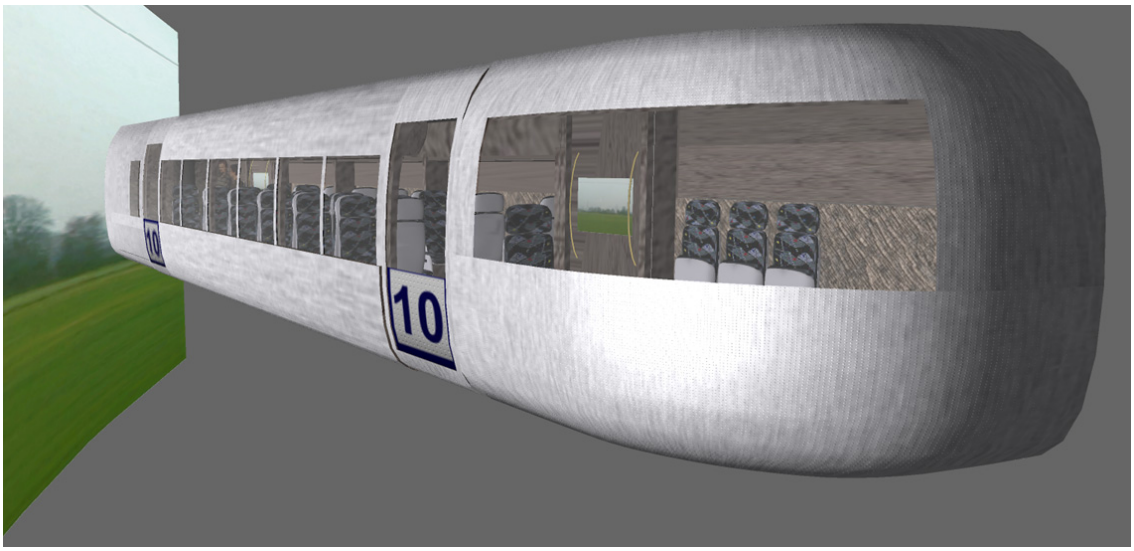

Abbildung 1.2.: Außenaufnahme People Cargo Mover

den. Die Landschaft sollte gleichfalls als Beleuchtung für den Innenraum dienen, um eine realistisch wirkende Helligkeitsänderung zu erzielen.

Abbildung 1.2 zeigt eine Aufnahme des Zuges von außen. Diese Einstellung ist eigentlich nicht Bestandteil der Visualisierung, daher ist hier der von innen nicht sichtbare graue Standardhintergrund erkennbar. Die Projektionsfläche für die Landschaft ist links im Hintergrund sichtbar.

1.4. Existierende Arbeiten

Die Verwendung von Texturen zur Beleuchtungsermittlung ist vielfach Gegenstand der Forschung. Auch über die spezielle Verwendung von Videotexturen wurden bereits Publikationen veröffentlicht. Eine der Arbeiten, die sich mit den Effekten des bewegten Bildes innerhalb einer computergenerierten Szene beschäftigen ist [Sch00]. Hauptzweck ist das Aufzeigen von Möglichkeiten, eine in sich geschlossene Sequenz (zum Beispiel bei einem Kaminfeuer) zu erzeugen, um den Realitätsgrad der Szene zu erhöhen.

[PMWS03] zeigt eine Anwendung, die in gleicher Weise wie die vorliegende Arbeit eine Textur als Umgebungsbeleuchtung einsetzt. Sie basiert allerdings auf einer speziellen Hardware (OpenRT), die in der Lage ist, Ray Tracing in Echtzeit durchzuführen. Dabei sind auch Effekte wie Spiegelungen und Schatten (Abbildung 1.3) möglich. Der physikalisch korrekten Berechnung steht aber noch ein hoher Aufwand entgegen (24 CPUs, welche per Netzwerk verbunden sind). Das System ermöglicht allerdings auch die direkte Interaktion der computergenerierten Szene mit einer realen Umgebung indem die Lichtverhältnisse durch eine HDR-Sonde erfaßt werden.

Abbildung 1.3.: in Echtzeit beleuchtete Tischszene (Quelle: CG Uni SB [Ope04b])

1.5. Gliederung der Folgekapitel

Zum Verständnis der Berechnung der Beleuchtung mit Hilfe der Videotexturen wird in Kapitel 2 eine Übersicht über grundlegende Beleuchtungstheorien und -modelle in der Computergrafik aufgestellt.

Um die Funktionalität der Shader innerhalb des Systems zu erläutern, wird in Kapitel 3 die Verarbeitung innerhalb der Grafikkarte dargestellt. Es wird gezeigt, in welchen Bereichen die Hardware programmierbar ist und wie die GPU die einzelnen Datenströme abarbeitet.

In Kapitel 4 wird auf den Aufbau und die Struktur der Shader eingangen. Es werden die verwendeten Hochsprachen und dabei speziell die OpenGL-Variante GLSL vorgestellt. Die GLSL-Sprachelemente werden grob definiert und es wird eine Übersicht über die zur Verfügung stehenden Möglichkeiten erstellt. Kapitel 5 beschreibt die Implementierung und Realisierung der Anwendung und geht dabei speziell auf die Videotextureinbindung und die Shaderprogrammierung ein. In Kapitel 6 wird dann das Ergebnis präsentiert und eine kurze Analyse durchgeführt. Kapitel 7 schließt mit einem Ausblick auf die Zukunft der Entwicklung in diesem Bereich und den Möglichkeiten, die sich dadurch für die vorliegende Anwendung ergeben werden.

2. Theorie der Beleuchtung

2.1. Lichtwahrnehmung

Das Vorbild von photorealistischen Computergrafiken stellt die Natur dar. Viele Phänomene sind durch die Physik und den mit ihr verbundenen Disziplinen erforscht und eingeordnet. Dies gilt auch für das Licht.

Bei dem Naturphänomen Licht handelt es sich um die elektromagnetische Strahlung im Bereich von 400 - 700 nm, die einen Reiz im menschlichen Auge auslöst, welchen wir als "Sehen" bezeichnen. Das menschliche Auge besitzt unterschiedliche Rezeptoren, Zapfen und Stäbchen genannt. Stäbchen sind für das Nachtsehen ausgeprägt (skotopisch), Zapfen für das Farbensehen (photopisch). Die Unterscheidung von Farben ist möglich, da es die Zapfen mit 3 verschiedenen Pigmenten mit jeweils unterschiedlichem Absorptionsmaximum gibt. Da das Spektrum durch die Zapfen bewertet wird, ist es möglich, daß eigentlich unterschiedliche Spektren für das menschlichen Auge gleich aussehen (metamere Farben). Die wahrgenommene Farbe ergibt sich durch die dominante Wellenlänge innerhalb des angebotenen Spektrums. Die Empfindlichkeit für die Wellenlängen ist dabei nicht konstant, eine Frequenz im blauen Bereich (um 700 nm) wird wesentlich schlechter wahrgenommen als eine Wellenlänge gleicher Amplitude im grünen Bereich (um 550 nm). Ein Spektrum, welches über den gesamten erfaßten Bereich homogon ist, erscheint dem menschlichen Auge als "weiß".

Da das menschliche Auge die Farb- und Helligkeitsinformation anhand von 3 Reizinformationen ermittelt (Tristimulus-Theorie), wurde erfolgreich versucht, das abgestrahlte Spektrum einer beliebigen Lichtquelle durch die Gewichtung von 3 festgelegten Spektren (Primärfarben) äquivalent darzustellen. Die 3 Phosphoren (Rot/Grün/Blau) eines Moni-

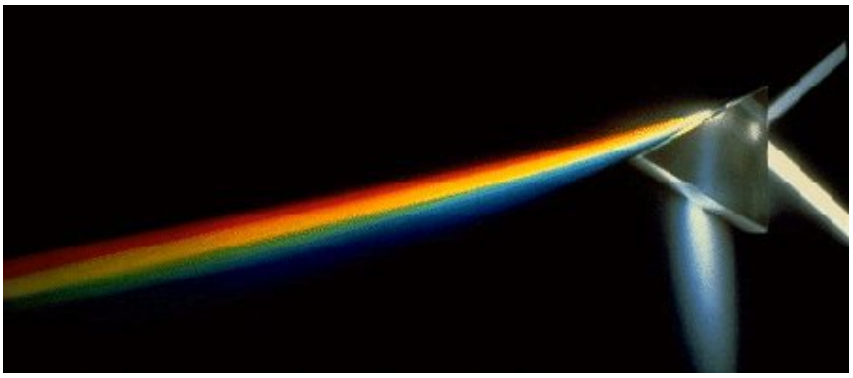

Abbildung 2.1.: Prisma (Quelle: NASA [NAS04])

tors sind ein Ergebnis dieser Versuche. Durch sie kann ein Großteil[1] der Farben in der Natur repräsentiert werden. Informationen dazu und darüber hinausgehende Details zu den hier nicht behandelten CIE Normvalenzen, welche die Basis für eine genaue Farbmessung darstellen finden sich in [FvDFH90].

Es gilt festzuhalten, daß die Farbe, in der eine Lichtquelle dem menschlichen Auge erscheint, mit Hilfe eines Zahlentripels ausgedrückt werden kann, so daß es nicht nötig ist, die Information über das komplette Spektrum zu speichern. Es handelt sich dabei im Prinzip um eine Irrelevanzkodierung, die auf die menschliche Wahrnehmung optimiert ist.

2.2. Beleuchtung

Im üblichen Fall wird man nicht nur eine Lichtquelle selber erfassen wollen, sondern auch die Objekte, welche von ihr beleuchtet werden. Die Farbe, unter der ein nichtselbstleuchtendes Objekt erscheint, ist der reflektierte Anteil des auf das Objekt einwirkenden Lichtes. Das Spektrum der Reflektion ist abhängig vom Material der Oberfläche. Ein Objekt, welches als "rot" bezeichnet wird, absorbiert die Wellenlängen im blauen und grünen Bereich und reflektiert bevorzugt den roten Spektralanteil.

Erinnern wir uns, daß die wahrgenommene Farbe einer Lichtquelle abhängig ist von ihrem Spektrum. Im Zusammenhang mit einer Objektoberfläche bedeutet dies, daß ein weißes Objekt ebenfalls als rot erscheinen kann wenn die beleuchtende Lichtquelle nur Wellenlängen im "roten" Bereich ausstrahlt. Sollte ein Objekt das Spektrum der Lichtquelle vollständig absorbieren, so würde es absolut "schwarz" erscheinen. Dieser Fall ist in der Natur unwahrscheinlich, da eine Oberfläche üblicherweise mit feinen reflektierenden Staub- und Schmutzpartikeln versehen ist.

Interessant ist nun die Frage, wie ein Objekt das ankommende Licht in die Richtung des Betrachters reflektiert. Die gesamte abgegebene Strahlungsleistung einer Lichtquelle wird in Joule pro Sekunde bzw. Watt angegeben und mit Φ bezeichnet. Auf eine Oberfläche im Raum wirkt ein Anteil davon ein, man ermittelt dies als Bestrahlungsstärke E mit dem Zusammenhang:

$$E = \frac{d\Phi}{dA} \tag{2.1}$$

Die Einheit ist Watt pro Quadratmeter und bezieht somit die aufgenommene Strahlungsleistung auf eine Einheitsfläche. Für die abstrahlende Fläche gibt es eine dazu analoge Meßgröße, die spezifische Strahlungsemission, ebenfalls in Watt pro Quadratmeter. In der Computergrafik betrachtet man Lichtquelle, Objektoberfläche (= Pixel) und Betrachter aber als Punkt, so daß dort die Strahlungsdichte L auf den Raumwinkel, angegeben in sr (Steradiant), bezogen wird. Die Formel für den Zusammenhang lautet:

$$L = \frac{d^2\Phi}{dA(d\omega cos\theta)} \tag{2.2}$$

dω steht hier für den differentiellen Raumwinkel. Wenn dieser Winkel gegen Null strebt, so wird aus dem Raumwinkel gedanklich ein einziger Lichtstrahl. Das θ in der Gleichung

[1] es können nicht alle Farben repräsentiert werden, weil der Anteil der Phosphoren teilweise negativ sein müßte

bezeichnet den Winkel zwischen Oberflächennormalen und der Lichtquelle und ist nötig weil die Lichtmenge bei einer Oberflächenneigung auf eine größere Fläche verteilt wird. Geht man der Einfachheit halber von einem konstanten Winkel von 0^o aus, so wird der Wert zu 1 und die Formel vereinfacht sich zu:

$$L = \frac{d^2\Phi}{dAd\omega} \tag{2.3}$$

Mit dieser Formel kann nun die Strahlungsdichte für einen beliebigen Punkt in der Szene berechnet werden. Sämtliche bis hierhin angegebenen Meßgrößen sind radiometrisch, d.h. sie beziehen nicht die spektrale Empfindlichkeit des Auges ein. Meßgrößen, die dies berücksichtigen (photometrisch) gibt es aber ebenfalls. Um sie zu erhalten wird die entsprechende Strahlungsgröße mit der spektralen Empfindlichkeitskurve des menschlichen Auges gewichtet. Die Strahlungsleistung (Φ) wird dabei zur Lichtleistung und hat die Einheit Lumen, die Strahlungsdichte (L) wird zur Leuchtdichte und besitzt die Einheit Candela pro Quadratmeter. Sie ist direkt proportional zur wahrgenommenen Helligkeit.

2.3. BRDF

Die auf einen Punkt einwirkende Strahlungs- bzw. Leuchtdichte kann nun mit den oben angeführten Formeln berechnet werden, es fehlt aber noch das von der Objektoberfläche wieder abgestrahlte Spektrum in Richtung des Betrachters. Zur besseren Aufschlüsselung bedient man sich eines Modells, welches in Abbildung 2.2 dargestellt ist. $\vec{s}$ und $\vec{t}$ sind die Oberflächentangenten (sie bilden mit der Normalen $\vec{N}$ ein Koordinatensystem), θ_i ist der Winkel, den die Lichtquelle mit der Normalen einschließt, ϕ_i der dazugehörige Azimuth. Entsprechend beschreiben die mit "o" indizierten Werte die Winkel für den Betrachter. $\vec{l}$ und $\vec{e}$ bezeichnen die normalisierten Licht- und Betrachtervektoren und ergeben sich aus den entsprechenden Winkeln zu:

$$\vec{l} = \begin{pmatrix} sin\theta_i sin\phi_i \\ cos\theta_i \\ sin\theta_i cos\phi_i \end{pmatrix}, \vec{e} = \begin{pmatrix} sin\theta_o sin\phi_o \\ cos\theta_o \\ sin\theta_o cos\phi_o \end{pmatrix} \tag{2.4}$$

Die Funktion

$$f(\theta_o, \phi_o, \theta_i, \phi_i) \tag{2.5}$$

wird als BRDF (*Bidirectional Reflectance Distribution Function*) bezeichnet, zu deutsch Bidirektionale Strahldichte-Verteilungsfunktion. Hinter diesem langen Begriff verbirgt sich ein recht einfacher Zusammenhang. Setzt man die vier Winkel ein, so liefert die Funktion einen Wert zurück, der den in die Betrachterrichtung reflektierten Anteil der Strahldichte der Lichtquelle beschreibt.

Da nun eine Funktion zur Verfügung steht, welche das Verhältnis zwischen einem gegebenen Ein- und Ausfallvektor angibt, kann man einen verallgemeinerten Zusammenhang für alle Richtungen herleiten. Man denkt sich über den Punkt auf einer Oberfläche eine Halbkugel und integriert über diese das gesamte einfallende Licht.

$$L(\theta_o, \phi_o) = \int \int_\Omega f(\theta_o, \phi_o, \theta_i, \phi_i) L(\theta_i, \phi_i) cos(\theta_i) d\sigma(\theta_i, \phi_i) \tag{2.6}$$

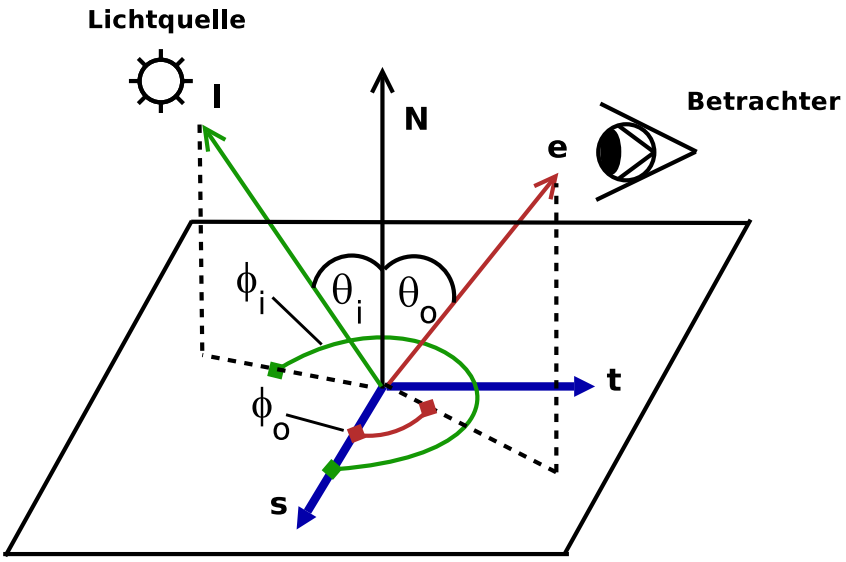

Abbildung 2.2.: Winkel bei der BRDF

Auch hier drückt die Formel einen einfachen Zusammenhang aus. Sie ermittelt die Strahldichte für den gegebenen Ausfallwinkel, indem sie über alle möglichen Einfallwinkel der Halbkugel Ω integriert. Für jeden Vektor wird der BRDF-Wert ermittelt und mit dem Kosinus des einfallenden Winkels gewichtet (wir erinnern uns: je steiler der Winkel, desto größer die Verteilung des Lichtes über die gesamte Fläche). Das Ergebnis ist die Summe der Strahldichten aller Lichtquellen. Um den Farbwert eines Punktes zu erhalten wird dies jeweils für die drei Farbkanäle durchgeführt.

Eine Vereinfachung der Formel ist möglich, wenn man nur eine einzige Punktlichtquelle betrachtet und die Winkel durch die entsprechenden Vektoren ersetzt:

$$L(\vec{e}) = f(\vec{e}, \vec{l})L(\vec{l})cos(\theta_i) \tag{2.7}$$

Der Kosinusausdruck kann schließlich noch durch das Vektorprodukt zwischen der Oberflächennormale und dem Lichtvektor ersetzt werden[2]:

$$L(\vec{e}) = f(\vec{e}, \vec{l})L(\vec{l})(\vec{N} \cdot \vec{l}) \tag{2.8}$$

Falls sich mehrere Lichter in der Szene befinden, wird dies für jede Lichtquelle und für jeden Farbkanals wiederholt und die Ergebnisse werden aufsummiert.

Neben der hier vorgestellten BRDF-Funktion gibt es noch weiter verallgemeinernde Varianten, welche Eigenschaften wie Abhängigkeit von der Position innerhalb des Materials, Polarisation des Lichtes, Transmission, Lichtverteilung usw. mit einbeziehen. Eine Übersicht sowie eine detailiertere Darstellung der hier vorgestellten Zusammenhänge verschafft [AMH99].

[2]Voraussetzung ist, daß beide normalisiert sind

2.4. Ermittlung der BRDF

Fassen wir zusammen: Die Helligkeit eines Punktes in Abhängigkeit von der Strahldichte einer Lichtquelle und dem Beobachterstandort läßt sich mit Hilfe der BRDF ermitteln. Die Frage ist nun, wie die BRDF selber ermittelt wird. Eine Idee ist es, das abgestrahlte Spektrum eines zu erfassenden Materials unter verschiedenen Winkeln zu messen. Die Meßwerte geben dann den genauen spektralen Zusammenhang wieder. Die Anzahl der erfaßten Meßwerte würde dabei allerdings sehr groß sein. Die Reduktion auf einen Punkt auf der Oberfläche erfaßt darüber hinaus bei einigen Materialien nicht das wahre Reflektionsverhalten, da sich bei ihnen das Licht im Material verteilt und eine Reflektion nicht nur an der gemessenen Stelle auftritt. Bei anderen Materialien ist die Reflektion stark winkelabhängig und eine Messung in endlichen Abständen ist möglicherweise nicht in der Lage, dieses Phänomen zu erfassen.

Kurzum, die rein punktuelle Vermessung allein kann noch keine generell zufriedenstellende Lösung liefern. Es gibt daher eine Vielzahl von Publikationen und Ansätzen zu diesem Thema, eine Zusammenfassung ist zum Beispiel in [AMH99] zu finden. In [Ros04] stellt R.Rost ein Verfahren unter Verwendung von polynomialen Texturen vor und setzt dies mit Hilfe von GLSL um. In [Fer04] wird ein Verfahren für SBRDF (spatial BRDF, es handelt sich um eine ortsabhängige BRDF) vorgestellt, welches ebenfalls mit Shadern umgesetzt wird. Ein wichtiger Faktor bei der Berechnung von BRD-Funktionen spielt die Fresnel-Reflexionsformel. Sie besagt, daß die Reflektivität eines Materials zunimmt, je steiler der Blickwinkel auf dieses Material ist (also für den Fall, daß θ_i gegen 90^o tendiert). Eine schöne Darstellung an Beispiel einer real gemessenen Größe findet sich in [WW99].

Für die vorliegende Arbeit wurde keine aufwendige BRD-Funktion verwendet, da das Hauptaugenmerk auf die Realisierung des Beleuchtungsverfahrens gerichtet war. Außerdem sollte auch mit der Hardware, die dem Autor zur Verfügung stand, Echtzeitgeschwindigkeit erreichen werden.

2.5. Polygone

In der Realität sind Oberflächen kontinuierlich, sie sind an jedem Punkt definiert und die Meßfläche läßt sich beliebig[3] verkleinern. In der Computergrafik gibt es mathematische Verfahren, die Oberflächen ebenfalls kontinuierlich beschreiben können. Die existierende Hardware ist aber nicht auf solche mathematischen Parameter optimiert[4]. Sie erwartet die Oberfläche als diskrete planare Flächen, welche durch die linearen Verbindungen von Eckpunkten (engl. *Vertex*, Plural *Vertices*) definiert werden. Eine solche Fläche wird auch als Polygon (zu deutsch eigentlich "aus vielen Winkeln bestehend") bezeichnet. Grafik-APIs wie OpenGL erlauben es auch, nur ein oder zwei Vertices als Definition einer Grundform anzugeben, folgerichtig dann Punkt und Linie genannt. Flächen werden durch 3 oder mehr koplanare Eckpunkte gebildet. Diese werden dann zu einer Maschenstruktur (engl. *mesh*) zusammengefaßt, so daß sie einen Verbund bilden und keine Übergänge zwischen ihnen sichtbar werden. Die Aufteilung eines Polygons wird sogar zwingend nötig wenn

[3] ab einer gewissen Größenordnung sind natürlich quantenmechanische Effekte und damit Erfassungsprobleme zu erwarten

[4] die Hardware bietet allerdings die Erzeugung von Polygonen mit Hilfe von Evaluatoren an

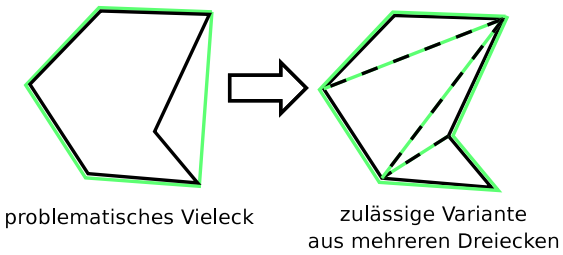

Abbildung 2.3.: unzulässige Polygon-Form

Abbildung 2.4.: Teekanne mit einer Farbe pro Fläche

eine Grundform nicht konvex ist. Abbildung 2.3 zeigt ein Beispiel. Hinzuzufügen ist noch, daß die übliche Grafikkartenhardware auf unterster Ebene Dreiecke rasterisiert, so daß es einen Geschwindigkeitsvorteil bieten kann wenn die Polygondaten bereits vorher in Dreiecke zerlegt werden (engl. *triangulation*).

Für jeden Eckpunkt können neben seiner Position auch noch weitere Informationen wie Farbe oder Texturkoordinaten vorgegeben werden. Für das Füllen der kompletten Polygonfläche (im Englischen als *face* bezeichnet) müssen daraus dann die Werte für jeden dargestellten Pixel errechnet werden. Dafür gibt es verschiedene Ansätze und Möglichkeiten.

Das Einfachste ist es, eine gemeinsame Farbe pro Fläche festzulegen. Je nach Unterteilung der Fläche führt das natürlich zu deutlichen Abstufungen. In Abbildung 2.4 ist dieses Problem gut zu erkennen.

Da sich wohl nur wenige natürliche Objekte auf diese Weise realistisch darstellen lassen eignet sich diese Methode nur für die schnelle Vorschau. Um eine realistischere Oberfläche zu bekommen, wird daher die Farbinformation für jeden Pixel durch Interpolation aus den Eckpunkten gewonnen. Die zwei dabei üblicherweise verwendeten Methoden sind nach ihren Erfindern als Gouraud und Phong-Schattierungsverfahren[5] (engl. *shading*) bekannt.

[5]gemeint ist die Schattierung der Pixelfarbe (= Änderung des Helligkeitswertes bei Beibehaltung des Farbwertes)

Abbildung 2.5.: Pixelberechnung nach Gouraud (li.) und Phong (re.) im Vergleich

Beim Gouraud-Verfahren (beschrieben in [Gou71]) berechnet man die Farbwerte für jeden Eckpunkt und interpoliert daraus die anderen Pixel. Dieser Ansatz ist einfach zu implementieren und zeigt bereits ein realistischeres Ergebnis. Da die Interpolation aber linear innerhalb jeder Fläche erfolgt und Übergänge zu benachbarten Flächen unberücksichtigt bleiben, erkennt man unter ungünstigen Bedingungen so wieder die Maschenstruktur der zugrundeliegenden Polygone (siehe Abbildung 2.5 linke Seite). Das in der gleichen Abbildung rechts dargestellte Bild zeigt das Phong-Verfahren ([Pho75]). Dieses Bild wirkt wesentlich realistischer, man hat hierbei das Gefühl, daß die Oberfläche der Teekanne eine durchgehende Fläche ohne Ecken und Kanten ist.

Die Idee war es, die Berechnung der Farbinformation nicht nur an den Eckpunkten selber, sondern für jeden Punkt der Fläche neu durchzuführen. Aus Kapitel 2.3 ist bereits bekannt, daß zur Berechnung der Farbinformation die Oberflächennormale verwendet wird. Phong schlug daher vor, jedem Eckpunkt eine eigene Normale zuzuweisen und zwischen diesen zu interpolieren. Der dadurch ermittelte Wert wird dann zur Berechnung des Pixels benutzt. Dies führt zu einem scheinbar kontinuierlichen Verlauf der Oberfläche, obwohl die Position der Punkte selber nicht berechnet wird.

Da die Eckpunktnormalen durch die Software vorgegeben werden, kann für ihre Berechnung die umliegenden Flächen herangezogen werden, damit die Interpolation zwischen allen Normalen der Oberfläche ohne größere Diskontinuitäten[6] erfolgt. Das Verfahren kann nur bei entsprechend vorberechneten Normalen seine volle Wirkung entfalten, daher ist hier die Vorarbeit durch die Modellierungssoftware von kritischer Bedeutung.

2.6. Phong-Verfahren

Nachdem nun bekannt ist, wie die zur Berechnung der jeweiligen Pixel nötige Information gewonnen wird, fehlt noch die Berechnungsvorschrift einer BRDF selber. Wie schon angeführt, nahm der Autor für die vorliegende Arbeit eines der einfacheren aber etablierten Verfahren als Grundlage. Es ist als Phong-Beleuchtungsmodell bekannt und stammt vom selben Autor wie das Phong-Verfahren in Kapitel 2.5. Es handelt sich um ein rein em-

[6] dies kann aber natürlich auch beabsichtigt sein, man denke an eine Kante beim Würfel

pirisches Modell, da die zugrundeliegenden Ideen keinem physikalischen Zusammenhang entsprechen. Phong setzt das Licht, welches von dem Punkt an der Oberfläche zurückgeworfen wird, gedanklich aus zwei Anteilen zusammen. Einmal ein diffuser Anteil (engl. *diffuse term*) und ein gespiegelter Anteil (engl. *specular term*). Die verwendete BRDF ist dabei:

$$f_{phong}(\vec{e},\vec{l}) = [k_d(\vec{N}\cdot\vec{l}) + k_s(\vec{r}\cdot\vec{e})^n]\frac{1}{(\vec{N}\cdot\vec{l})} \tag{2.9}$$

Setzt man diese Funktion in die allgemeine Formel 2.8 aus Kapitel 2.3 ein, so ergibt sich:

$$L_{phong}(\vec{e}) = [k_d(\vec{N}\cdot\vec{l}) + k_s(\vec{r}\cdot\vec{e})^n]L(\vec{l}) \tag{2.10}$$

Der Ausdruck $\frac{1}{(\vec{N}\cdot\vec{l})}$ am Ende von Formel 2.9 wird nur benötigt, damit er beim Einsetzen herausfällt, Phong bezieht die Winkelabhängigkeit bereits in den diffusen Anteil mit ein. Dieser diffuse Anteil wird von einer Oberfläche in alle Richtungen gleichermaßen reflektiert. Er ist unabhängig vom Betrachterstandort. Der gespiegelte Anteil hingegen ist abhängig vom Betrachterwinkel. Der bislang noch nicht erwähnte Vektor $\vec{r}$ stellt den Reflektionsvektor der Lichtquelle dar. Da Einfallswinkel gleich dem reflektierten Ausfallswinkel ist, gilt:

$$\vec{r} = \vec{l} - 2(\vec{N}\cdot\vec{l})\vec{N} \tag{2.11}$$

$\vec{l}$ und $\vec{N}$ müssen dafür normalisiert sein. Die beiden k's in Formel 2.9 und 2.10 sind Parameter, um die Auswirkung der beiden Anteile zu variieren. k_d wirkt auf den diffusen, k_s auf den gespiegelten Lichtanteil. Der Exponent n kontrolliert den Abfall des gespiegelten Lichtanteils sobald der Betrachterwinkel sich aus der Hauptreflektionsrichtung hinausbewegt (das dadurch auf einen Bereich konzentrierte Spiegellicht wird englisch als *highlight* bezeichnet).

Mit Hilfe des BRDF-Browsers "bv" von Szymon Rusinkiewicz [Rus01] läßt sich die Auswirkung der Parameter anschaulich darstellen. Er kann neben Phong auch noch eine Vielzahl weiterer BRD-Funktionen visualisieren.

Abbildung 2.6 zeigt zwei verschiedene Einstellungen. In jeder ist auf der rechten Seite eine von der Lichtquelle beschienene Kugel zu sehen und links ist die schematische Darstellung der beiden Vektoren $\vec{l}$ (grün) und $\vec{e}$ (hier nicht sichtbar, da innerhalb der purpurnen Hülle) zu erkennen. Der Parameter n steht einmal auf 200 (links) und einmal auf 10 (rechts). Die purpurne Hülle stellt den reflektierten Anteil in die vom Ursprung ausgehende Richtung dar. Man erkennt, daß für größere n der Bereich des "Phong-Highlights" kleiner wird.

Da die Berechnung des Reflektionsvektors durch die Vektormultiplikation einen gewissen Rechenaufwand erfordert, vereinfachte Blinn die Formel, indem er statt des Reflektionsvektors einen sogenannten Halbvektor (engl. *half vector*) einführte, der sich wie folgt errechnet:

$$\vec{h} = \frac{\vec{l}+\vec{e}}{2} \tag{2.12}$$

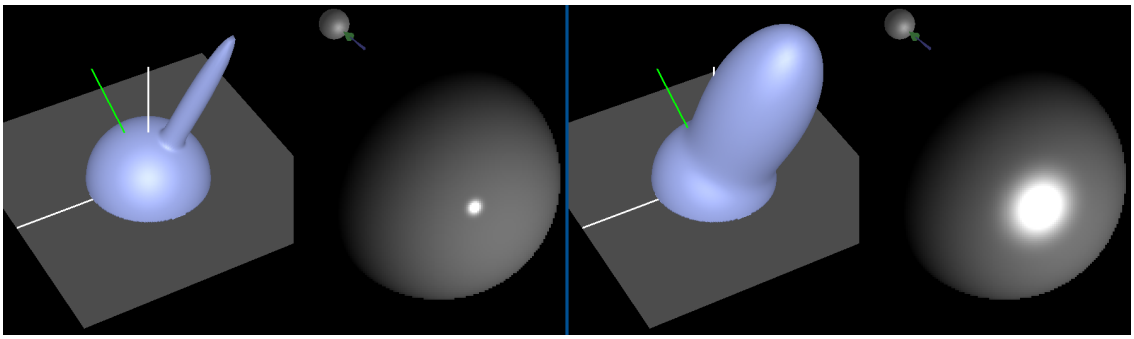

Abbildung 2.6.: Phong-BRDF (Bilder erzeugt mit BV [Rus01])

Der Halbvektor stellt die Normale dar, unter der die Lichtquelle perfekt in Richtung des Betrachters spiegeln würde. Je größer der Winkel zur Lichtquelle, desto kleiner wird also auch der gespiegelte Anteil. Man darf nicht vergessen, daß der diffuse Anteil nach wie vor mit der Normalen der Oberfläche berechnet wird, die Änderung betrifft nur den gespiegelten Anteil. Damit wird die Formel 2.10 zu:

$$L_{blinn}(\vec{e}) = [k_d(\vec{N} \cdot \vec{l}) + k_s(\vec{h} \cdot \vec{e})^n]L(\vec{l}) \tag{2.13}$$

Diese Formeln bilden die Basis für die Lichtberechnung in den heutzutage üblichen Grafik-APIs für Echtzeitanwendungen.

2.7. Grafik-API Lichtquellen

Da nun die theoretischen Grundlagen der Berechnung der empfangenen und abgestrahlten Leuchtdichte ermittelt wurden, folgt die Betrachtung, wie Lichtquellen in der Computergrafik-Praxis bislang repräsentiert werden.

In den üblichen Grafik-APIs wird eine Lichtquelle über den Leuchtdichteanteil ihrer drei Primärfarben (Rot, Grün und Blau) beschrieben. Die Anteile werden normiert dargestellt, so daß ein Wert von 1.0 in allen 3 Anteilen eine weiße Lichtquelle und gleichzeitig die größtmögliche Helligkeit ergibt. Darüber hinaus können zum Beispiel bei OpenGL die Anteile der Primärfarben für den diffusen und gespiegelten Anteil getrennt definiert werden, so daß man für den Spiegellichtanteil eine andere Farbe angeben kann als für die diffuse Reflektion. Dies entspricht der Beobachtung, daß bei Kunststoff der gespiegelte Anteil nicht die Farbe des Kunststoffs annimmt, während dies bei metallischen Oberflächen der Fall ist. Da in einer natürlichen Umgebung immer ein gewisses Grundlicht vorhanden ist, erlaubt es OpenGL darüber hinaus, unabhängig von diffusem und gespiegeltem Lichtanteil einen Umgebungslichtanteil anzugeben, welcher in seiner farblichen Zusammensetzung ebenfalls frei definierbar ist.

Die Aufgabe dieser Lichtquellen läßt sich mit den Scheinwerfern im klassischen Theater oder im Film vergleichen. Sie sollen nicht die natürlichen Verhältnisse widerspiegeln, sondern möglichst viele Möglichkeiten und Parameter anbieten, um flexibel jede gewünschte Lichtsituation erzeugen zu können.

Abbildung 2.7.: Kugelumgebungstextur (Quelle: Paul Haeberli [Hae04])

Wenn man sich aber ein Objekt in einer natürlichen Umgebung vorstellt, so stammt nur ein geringer Teil des einwirkenden Lichtes direkt von der Lichtquelle. Ein großer Anteil stammt von Licht, welches von der Umgebung auf dieses Objekt reflektiert wurde. Diesen globalen Beleuchtungsanteil zu berechnen ist die Aufgabe von Techniken wie Ray Tracing und Radiosity. Sie sind aber noch zu rechenintensiv, um befriedigend in Echtzeit ablaufen zu können auch wenn es bereits interessante Entwicklungen in diesem Bereich gibt wie den Saarcor [Saa04]. Die gegenwärtige Forschung für den Echtzeitbereich verfolgt daher für diesen Lichtanteil die Möglichkeit, Informationen über die Beleuchtung aus Bildern der Umgebung zu gewinnen, in die das Objekt integriert werden soll.

2.8. Bildbasierte Lichtquellen

Die Environment Map (zu deutsch: Umgebungstextur) kann als Vorläufer der bildbasierten Lichtquellen gesehen werden. Sie wurde ursprünglich für voll spiegelnde Oberflächen benutzt. Da die Verfahren zur Verfolgung der Lichtstrahlen (Ray Tracing) sehr rechenzeitaufwendig sind, sind sie für Echtzeit-Bedingungen ungeeignet. Um trotzdem die Illusion einer Spiegelung zu erzeugen, projiziert man eine Abbildung der Umgebung auf das Objekt. Ein beliebtes Verfahren ist dabei das *Cube Mapping*, bei dem eine 360^o-Ansicht durch 6 Aufnahmen gewonnen wird. Aus den Vektoren $\vec{e}$ und $\vec{N}$ der Oberfläche (siehe Abbildung 2.2) wird der Reflektionsvektor ermittelt und als Nachschlageinformation für die Cube Map benutzt. Da es kein echter Spiegel ist, können Objekte, die in geringer Entfernung zum berechneten Objekt stehen, nicht realistisch erfaßt werden[7].

Die Idee lag nahe, die Environment Map als Information über die auf das Objekt treffende Beleuchtung zu verwenden, da sie die Umgebung in alle Richtungen erfaßt. Bei

[7] es gibt allerdings shaderbasierte Ansätze, Kapitel 19, “Image-based lighting” in [Fer04].

einer (perfekten) Spiegeloberfläche wird ein auf die Oberfläche auftreffender Lichtstrahl in genau eine Richtung reflektiert. Bei einer diffusen Objektoberfläche verteilt sich die Leuchtdichte dieses Lichtstrahls nun in mehrere Richtungen. Der Zusammenhang dieser Verteilung wird durch die BRDF beschrieben und die Environment Map kann, mit dieser gefiltert, ausgewertet werden. W. Heidrich und P. Seidel beschrieben bereits 1999 ein solches Verfahren in [HS99]. Da diese Filterung in Echtzeit erfolgen muß, liegt es nahe, sie bereits im Voraus durchzuführen. Dabei gibt es jedoch einige Schwierigkeiten, die in Kapitel 5.9 noch näher erläutert werden.

In den letzten Jahren sind Verfahren entwickelt worden, die die Leuchtdichteinformation mit Hilfe von Kugelfunktionen (engl. *spherical harmonics*) erfassen und ein Abbild der Lichtsituation mit Hilfe weniger Koeffizienten erlauben. Dies kann in Analogie zur Fouriertransformation gesehen werden, die ebenfalls die Approximation einer Wellenform durch Koeffizienten durchführt. Nähere Informationen zu diesen Kugelfunktionen und ihrer Anwendung finden sich in [KSS02]. Da die so erzeugten Koeffizienten weniger Platz einnehmen als eine Textur, wird somit Speicher und Übertragungsbandbreite eingespart. Das Problem des zusätzlichen Rechenaufwandes tritt durch die zunehmende Geschwindigkeit der GPUs zurück.

3. Übersicht Grafikhardware

3.1. Aufbau

Das Grundschema einer üblichen Grafikkarte zeigt Abbildung 3.1. Sie war früher nur als Steckmodul in das PC-Gesamtsystem integriert, heutzutage ist sie aber oft schon direkter Bestandteil des Mainboards. Die Anbindung erfolgt in den meisten Fällen noch über den AGP-Bus, in Zukunft wird es aber zunehmend nur noch Systeme mit PCI-Express geben, welcher eine logische Weiterenwicklung der PCI/AGP-Busse darstellt (größere Bandbreite und Übertragungsgeschwindigkeit). Der als "Bus Bridge" bezeichnete Teil kennzeichnet den Chipsatz, der die Anpassung der Signale vom Bus zur GPU herstellt. Der DAC (engl. *digital/analogue converter*) ist der Digital/Analog-Wandler für den VGA-Anschluß. Bei Verwendung des DVI-D (engl. *digital video interface*) würde er nicht mehr unbedingt benötigt, alle bislang üblichen Karten bieten aber vorerst auch noch analoge Ausgänge an.

Die zwei RAM-Bänke stehen stellvertretend für unterschiedliche Organisation und Aufteilung der Speicherbausteine. Es gibt eine Vielzahl von Speichertypen, am populärsten sind zur Zeit DDR und DDR-II SDRAM (engl. Double Data Rate Synchronous Dynamic Random Access Memory. Die Anbindung dieses Speichers an die GPU erfolgt über einen sehr breiten Bus[1] (128/256 Bit). Zum Vergleich: der AGP-Bus hat nur maximal 32 Bit (4 Byte) zur Verfügung.

Zentraler Punkt dieses Kapitels ist jedoch der letzte verbleibende Abschnitt, die GPU (engl. *graphics processing unit*, Grafikprozessoreinheit). Sie übernimmt die Verarbeitung und Aufbereitung der vom PC kommenden Daten und legt die berechneten Grafikdaten anschließend in einem RAM-Bereich ab. In vielen Fällen ist dies direkt der Bildausgabe-

[1] die physikalische Anbindung ist allerdings wesentlich komplexer, da der Speicherbus in mehrere kleine Busse aufgeteilt wird, Stichwort *Crossbar*-Architektur

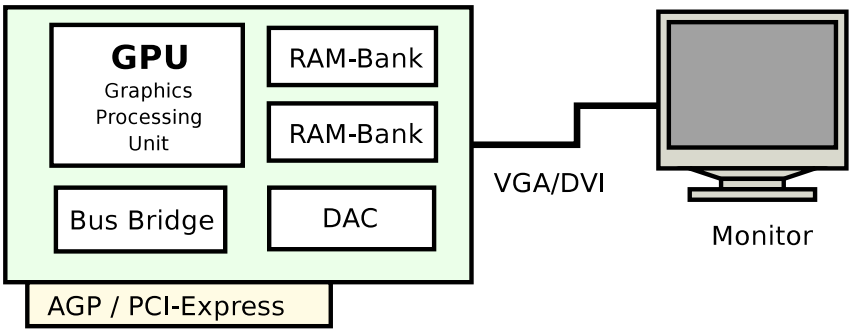

Abbildung 3.1.: Grafikkarte Schema

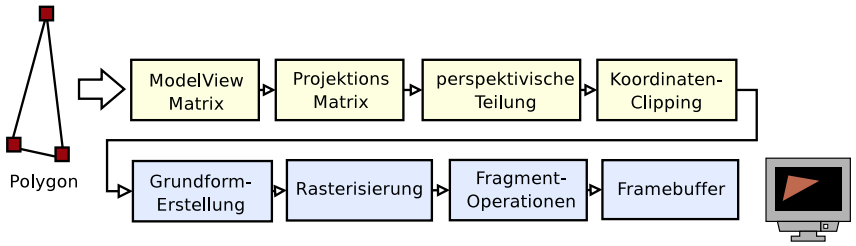

Abbildung 3.2.: klassische Stufen der Polygonberechnung

puffer (engl. *framebuffer*), teilweise werden die Daten auch zwischengespeichert, ohne daß eine direkte Darstellung erfolgt (zum Beispiel, um sie als Textur zu verwenden oder um mehrere Berechnungsdurchläufe zu akkumulieren).

Es folgt nun ein Überblick der einzelnen Aufgaben des Prozessors. Leider ist die Informationspolitik der Grafikkartenhersteller zu diesem Thema sehr beschränkend[2], so daß der tatsächliche Aufbau der Prozessoren sich nur indirekt über abstrakte Modelle wie OpenGL oder die Shadersprache erschließt, welche nicht der wirklichen Implementierung entsprechen müssen.

3.2. Klassische GPU-Aufgaben

Bei der GPU handelt es sich um sogenannte Datenstromprozessoren (engl. *stream processor*). Es steht nicht wie beim PC die Abarbeitung eines Programmteils durch eine Prozessoreinheit im Vordergrund, sondern die parallele Verarbeitung von Datenströmen durch viele gleichzeitig operierende kleinere Prozessoreinheiten. Um diese Darstellung zu präzisieren, betrachten wir in Abbildung 3.2 die klassische Abarbeitung eines Polygons. Als "klassisch" bezeichnet der Autor hier den Weg der sogenannten festen Funktionalität im Gegensatz zur erst später hinzugekommenen Programmierbarkeit.

Das Polygon besteht im Beispiel aus 3 Eckpunkten, welche als vierdimensionale Vektoren (homogenes Koordinatensystem) definiert sind. In den meisten Fällen werden die Eckpunkte im lokalen Koordinatensystem des Modellierungsprogramms definiert sein, mit einem frei gewählten Ursprung und einer 1.0 als vierter Koordinate. Die ModelView-Matrix ist dabei bereits das Produkt zweier Matrizen, die erste (Model) überführt die Koordinaten vom lokalen in das globale Weltkoordinatensystem der Szene. Dadurch wird ein Bezug zu den anderen Elementen der Szene hergestellt. Da homogene Matrizen verwendet werden, sind Skalierungen, Rotationen und auch Transformationen möglich.

Der andere Operand, die View-Matrix, verschiebt dann die gesamte Szene in die gewünschte Kameraposition. Die Ausgangslage ist der Ursprung mit Blick in die negative Z-Achsenrichtung.

Bis zu diesem Punkt ist die Szene noch vollständig dreidimensional vorhanden. Damit sie für die Monitorwiedergabe auf zwei Dimensionen und den für den (virtuellen)

[2]Die Furcht vor Spionage durch Mitbewerber scheint so groß zu sein, daß die meisten technischen Details geheimgehalten werden, so daß nur eine grobe Übersicht gegeben werden kann.

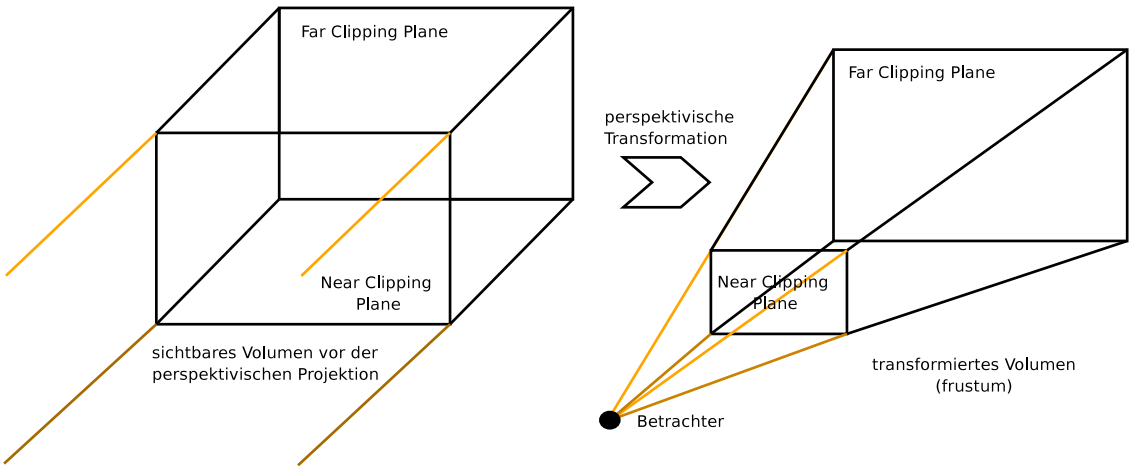

Abbildung 3.3.: perspektivische Projektion

Beobachter sichtbaren Bereich reduziert wird, muß nun eine Projektion erfolgen. Es gibt eine Reihe von Projektionsarten, für virtuelle Umgebungen sind dies häufig perspektivische Projektionen, da sie die Sichtweise des menschlichen Auges nachbilden (je weiter ein Objekt sich entfernt, desto kleiner erscheint es).

Um die Projektion zu erreichen, wird dabei eine spezielle Projektionsmatrix erstellt, welche die Koordinaten der Eckpunkte so transformiert, daß das sichtbare Volumen (engl. *view volume*) eine abgestumpfte Pyramidenform besitzt (engl. *frustum*). Die Auswirkung ist in Abbildung 3.3 schematisch dargestellt. Nach der perspektivischen Teilung (die drei ersten Koordinaten der sich ergebenden Eckpunkte werden durch die vierte Koordinate geteilt) werden die Koordinaten als sogenannte NDC (engl. *normalized device coordinates*) bezeichnet. Diese Normalisierung bedeutet, daß alle Koordinaten, die sich innerhalb des Bereichs von [-1...1] befinden, sichtbar sind und alle darüber hinausragenden sich außerhalb des sichtbaren Volumens befinden. Die Wahl der nahen und fernen Begrenzungsebenen (engl. *near/far clipping plane*) legt dabei den Tiefenbereich fest, innerhalb dessen Objekte erfaßt werden.

Es folgt noch eine Betrachtung für den Sonderfall, daß sich ein Polygon nur teilweise außerhalb des sichtbaren Bereichs befindet. In diesem Fall wird eine neue Koordinate an der begrenzenden Ebene erzeugt, so daß das Polygon später an dieser Stelle abgeschnitten werden kann, damit es nur innerhalb des sichtbaren Bereichs dargestellt wird. Dies ist im Schaubild 3.2 als “Koordinaten-Clipping” bezeichnet.

An dieser Stelle besitzen die Polygone noch 3 Koordinaten (die vierte ist nun überall zu 1.0 geworden und kann damit entfallen). Die Tiefeninformation (Z, dritte Koordinate) wird noch benötigt, um nach der Rasterisierung zu entscheiden, ob ein Objekt sichtbar ist oder durch andere verdeckt wird.

Es folgt nun die Grundformerstellung (engl. *primitive assembly*). Ziel der Rasterisierung wird es sein, die Fläche zwischen den Eckpunkten zu füllen. In der Grundformerstellung werden daher alle zu einem Polygon gehörigen Eckpunkte zusammengefaßt. Für die Berechnung der Farbeigenschaften der einzelnen Pixel werden die in Kapitel 2 beschriebenen

Formeln und Angaben verwendet. Darüber hinaus kann die Oberfläche noch mit Texturen versehen werden, von denen auch mehrere untereinander verknüpft werden können.

Der Unterschied zwischen Pixeln und Fragmenten ist nicht vollständig definiert, teilweise wird der Begriff auch gleichgesetzt. Man könnte als Unterschied festlegen, daß ein Fragment die Einheit ist, die als Pixel in den Bildspeicher geschrieben wird. Da mehrere Objekte hintereinander angeordnet sein können, kann es passieren, daß mehrere Fragmente in den gleichen Pixel geschrieben werden. Ob dies geschieht legen die erwähnten Fragmentoperationen fest. Im Normalfall ersetzt ein Fragment mit einer geringeren Tiefe eines mit einer größeren (es liegt weiter hinten). Für besondere Anwendungen läßt sich dieses Verhalten aber auch manipulieren und ganz ausschalten (wichtig für transparente Objekte).

Da auf einer Desktopoberfläche ein Fenster ein anderes verdecken kann, ist es möglich, daß ein Teil des berechneten Bildes nicht dargestellt werden soll, daher wird für jeden Pixel festgestellt, ob er innerhalb des erlaubten Bereiches liegt. Das Ausschneiden eines Bildinhaltes gehört auch zu den Fragmentoperationen und wird bei OpenGL als "Scherentest" (engl. *scissortest*) bezeichnet. Ebenfalls kann die Verknüpfung mit den schon im Bildspeicher vorhandenen Pixeldaten festgelegt werden (addieren, multiplizieren, ersetzen, ...). Weitere Operationen betreffen den Alphakanal und den Stencilpuffer und sollen hier nicht näher betrachtet werden. Ihnen allen gemeinsam ist, daß sie über das Schreiben des Fragments in den Bildspeicher entscheiden.

Genauere Details darüber lassen sich in [AMH99] nachlesen, Grundlagen zu den Matrizen und Projektionen in [FvDFH90].

3.3. Pipelines

Die in Kapitel 3.2 beschriebenen Funktionen werden für jedes Polygon durchgeführt. Festzuhalten ist, daß die Multiplikation mit der Transformations- und Projektionsmatrix für jeden Eckpunkt des Polygons unabhängig voneinander stattfindet und in Zusammenhang mit Kapitel 2 wird deutlich, daß die Berechnung der Lichtinformation wiederum unabhängig[3] für jeden einzelnen Pixel durchgeführt werden kann.

Es ist also möglich, einen Großteil der Berechnung parallel durchführen zu können. Aus diesem Grundgedanken heraus entstand die Aufteilung in parallel operierende Pipelines. Unter einer Pipeline wird dabei ein System verstanden, in welches man Daten eingibt, die nacheinander in verschiedenen Stufen abgearbeitet werden. Es ist dabei nicht möglich, eine Stufe zu überspringen. Wenn also eine Stufe zu lange benötigt, so stauen sich die Nachfolgedaten. Dieses "Steckenbleiben" wird im Englischen als *pipeline stall* bezeichnet und sollte unter normalen Umständen vermieden werden.

Man kann in der GPU die anfallenden Aufgaben in zwei Teilbereiche aufteilen, in Abbildung 3.2 sind sie farblich getrennt. Beim ersten Teilbereich handelt es sich um die Berechnung, die pro Eckpunkt anfällt und beim zweiten um die Berechnung, welche pro Pixel anfällt. Man kann dies als große Pipeline sehen, die sich wiederum in zwei kleinere Bereiche aufteilen läßt, die Eckpunkt- und die Fragmentpipeline. Jeder dieser Bereiche hat eine bestimmte, klar definierte Aufgabe zu erfüllen. Der Gedanke war naheliegend,

[3] dies gilt natürlich nur für lokale Beleuchtungsmodelle, auf welche man sich aber aus diesem Grunde auch beschränkt

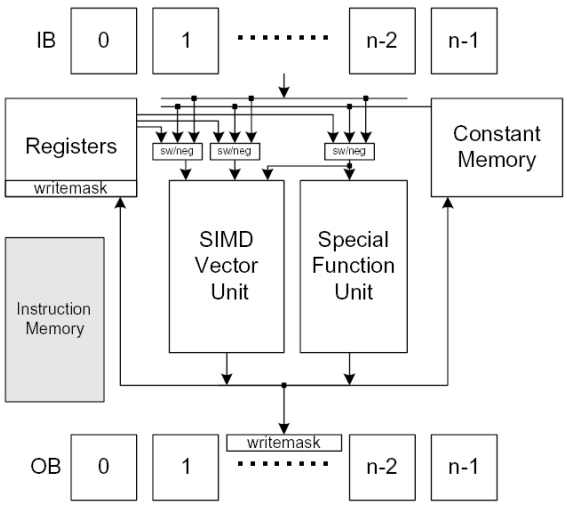

Abbildung 3.4.: Vertexprozessor FPU Blockdiagramm (Quelle: [LKM01])

die Berechnungen innerhalb dieser Pipelines von der fest verdrahteten Funktionalität hin zu einer frei programmierbaren zu erweitern. Dies führte zum Konzept des Vertex[4]-/ bzw. Fragmentprozessors.

3.4. Vertex-/Fragmentprozessor

Wie bereits erwähnt, kann an dieser Stelle nicht allzu detailliert auf die Operation der Vertex- und Fragmentprozessoren eingegangen werden. Außerdem ist dieser Bereich noch mit großem Wandel verbunden, sowohl in der Anzahl der Pipelines als auch - laut den groben Herstellerangaben - in ihrem Aufbau. Die FX 5600 des Autors besitzt nur 4 Fragment Pipelines, die GeForce 6800 bereits 16. Diese Zahl wird sich auf absehbare Zeit sicherlich noch vergrößern.

Die Pipelines haben wie beschrieben eine definierte Ein- und Ausgabe. Der Datenstrom kann also kontinuierlich stattfinden, jeder Vertex und jedes Fragment wird nacheinander abgearbeitet. Zugriff auf die Texturen erfolgt dabei über einen Cache, so daß örtlich zusammenliegende Texturdaten beschleunigt gelesen werden können. Es sind auch Programmverzweigungen möglich, allerdings ist die Definition der Schleifenvariable unter Umständen mit Einschränkungen verbunden (es muß teilweise ein konstanter Wert sein). Auch der Zugriff auf die Texturen ist nur bei der allerneuesten Generation direkt aus dem Vertex-Programm möglich, ältere GPUs erlauben dies nur vom Fragment-Programm aus.

Die Länge der Programme ist von anfangs nur wenigen Anweisungen auf 65535 angestiegen und wird somit von den Herstellern gerne als “unendlich” angegeben, auch wenn dies natürlich nicht den Tatsachen entspricht. Anders als bei einer CPU arbeiten die GPU-Prozessoren nicht bevorzugt mit Fest-, sondern mit Gleitkommazahlen (engl. *floating-point*

[4]Vertex ist der englische Begriff für “Eckpunkt” und soll für den Prozessor als Bezeichnung beibehalten werden, da es sich wie bei der “CPU“ um einen etablierten Begriff handelt, während ein “Eckpunktprozessor” wenig bekannt sein dürfte.

number). Die Länge der entsprechenden Datentypen hat im Laufe der Zeit von 12 auf 32 Bit zugenommen. Wie bereits erwähnt, ist der technologische Sprung in diesem Bereich noch sehr groß und die aktuelle Generation wird nicht die letzte sein. Eine schematische Darstellung der FPU (Gleitkomma-Recheneinheit) des Vertexprozessors ist in Abbildung 3.4 zu sehen, nähere Informationen darüber finden sich in [LKM01].

Da die Pixelpipelines die Pixeldaten für den Bereich zwischen den Eckpunkten liefern sollen, müssen sie auch Informationen über die interpolierten Werte bekommen. Wie in Kapitel 2.6 erläutert, betrifft dies zum Beispiel die Normalen welche für jeden Eckpunkt angegeben werden. Da die Interpolation perspektivisch korrekt durchgeführt werden muß, ist es nötig, daß alle Informationen über die umliegenden Eckpunkte vorliegen. Dies verdeutlicht den Ansatz der Unterteilung in Vertex- und Fragmentpipeline als zwei Teile einer großen GPU-Pipeline. Die Berechnung der Pixel erfolgt mit großer Wahrscheinlichkeit[5] auch in einem gewissen Muster, so daß benachbarte Pixel in einem Polygon auch benachbarte Pipeline-Einheiten beanspruchen, da die Texturdaten in diesem Fall normalerweise ebenfalls örtlich nahe beinander liegen. Auf diese Weise könnten sie zum Beispiel auf einen gemeinsamen Texturcache zugreifen und der gefürchtete *Cache Miss* (Information ist nicht im Cache enthalten und muß neu angefordert werden) ließe sich minimieren.

[5] hier schweigen sich die Hersteller leider über Details aus

4. Shader-Programmierung

4.1. Abstraktion

In Kapitel 3.4 wurde bereits erläutert, daß die GPU aus vielen parallel ausgeführten Pipelines besteht, von der jede unabhängig von der anderen arbeitet. Für die Programmierung wurde diese reale GPU-Implementierung auf ein abstraktes Modell reduziert, welches dem Shader-Entwickler eine Konzentration auf die wesentlichen Aufgaben erlaubt.

Die Verteilung der eingehenden Daten auf die einzelnen Pipelines ist dabei genausowenig beeinflußbar wie die Reihenfolge der Eckpunktabarbeitung innerhalb eines Polygons[1]. Es wird immer nur der Durchlauf für einen Eckpunkt oder einen Pixel betrachtet, die Speicherung von Informationen für einen späteren Abruf während eines anderen Fragment-Programm-Durchlaufs ist nicht möglich, da es die Möglichkeit einer unabhängigen parallelen Ausführung verhindern würde.

Die Anwendung ist aber in der Lage, über Variablen mit dem Shader zu kommunizieren. Es können der Pipeline auch zusätzliche, frei wählbare Eckpunkt-Eigenschaften hinzugefügt werden. Auf diese Weise kann zum Beispiel eine Information wie "Druck" oder "Temperatur" für jeden Eckpunkt vergeben werden. Eine gezielte Kommunikation mit dem Shaderdurchlauf pro Pixel ist allerdings nicht möglich, der Shader kann jedoch die Position des Pixels abfragen und hat so einen gewissen Spielraum.

4.2. Renderman

Wenn man die Entwicklung der Shadersprachen darstellen will, so ist auch die Renderman Interface Spezifikation zu erwähnen. Ursprünglich von Pixar [Pix04] 1988 nach sechsjähriger Entwicklung veröffentlicht, legte sie ein Protokoll zwischen Modellierungssoftware und der Grafikberechnungssoftware fest. Sie war ursprünglich darauf ausgelegt, in Hardware realisiert zu werden, dieses wurde allerdings nie in die Tat umgesetzt, sondern es blieb bei einem rein auf Softwareebene operierendem Protokoll.

Die Spezifikation deckt nicht nur eine Shadersprache ab, sondern bietet eine komplette Schnittstelle mit ähnlichen Aufgaben wie OpenGL. Der für die Shaderhochsprachen relevante Teilbereich nennt sich Renderman Shading Language. Sie ist ähnlich der in den Folgeabschnitten erwähnten Hochsprachen C-basiert und bietet die Möglichkeit, die Farbeigenschaften einer Objektoberfläche frei programmierbar zu erzeugen. Da sie nicht für die Merkmale der heutigen Grafikkarten ausgelegt war, ist die Art und Weise, in der Shader definiert werden (5 verschiedene Typen, zum Beispiel Lichtshader, Verschiebungsshader, Oberflächenshader...) nicht direkt in Übereinstimmung mit dem getrennten Modell der Vertex- und Fragmentprozessoren zu bringen.

[1] man kann experimentell Rückschlüsse auf die Implementierung ziehen, aber dies wäre nur für ein Modell wenn nicht sogar nur für einen Treiber gültig

Shadermodel	ps_2_0	ps_3_0	vs_2_0	vs_3_0
Programmlänge (Instruktionen)	96	65535	256	65535
interpolierende Register	10	10	10	10
temporäre Register	12	32	12	32
konstante Register (float)	32	224	>=256	>=256
konstante Register (bool)	16	16	16	16
konstante Register (int)	16	16	16	16

Tabelle 4.1.: Vergleich Shadermodel 2.0/3.0 (Quelle: Microsoft [MSD04])

Die Renderman Shading Language war nicht für die Erfüllung von Echtzeitbedingungen gedacht, sondern um hochqualitative Bilder zu erzeugen. Es wurden für die Entwicklung der späteren "Echtzeit"-Shadingsprachen allerdings Anleihen genommen (zum Beispiel die Typqualifizierer uniform und varying), so daß sie teilweise als Vorlage für diese gedient hat. Die Spezifikation kann von Pixar's Webseite [Pix04] bezogen werden.

4.3. Shadermodel

Bevor auf die einzelnen Hochsprachen eingegangen werden soll, muß noch der Begriff "Shadermodel" erläutert werden, welcher sich als Maßstab für die Beurteilung der GPU-Merkmale etabliert hat und in vielen Publikationen als bekannt vorausgesetzt wird. Diese Festlegung wurde ursprünglich von Microsoft für die DirectX-Grafik-API als gemeinsame Basis definiert. Im Gegensatz zur OpenGL-API, welche versucht, die Möglichkeiten der aktuellen Hardware widerzuspiegeln, ist das Shadermodel und dabei speziell die Version 3.0 auch auf zukünftige Entwicklungen ausgelegt und es gab bei der ursprünglichen Festlegung noch keine Hardware, welche vollständig 3.0-kompatibel war. Erst der NV40-Prozessor erfüllte die Anforderungen.

Microsoft unterscheidet zwischen Pixel- und Vertexshader, abgekürzt ps und vs. Dabei handelt es sich um Definitionen auf Assemblerebene. Die Tabelle 4.1 verschafft einen kleinen Überblick über so definierte Eigenschaften von Version 2.0 und 3.0.

Man erkennt, daß der größte Zuwachs bei der Programmlänge stattgefunden hat. Die "interpolierenden" Register verbinden dabei Vertex- und Pixelshader. Sie erlauben die perspektivisch korrekte Berechnung des Zwischenwertes für einen Pixel. Die Interpolation selber wird intern zwischen Vertex- und Pixelpipeline durchgeführt, eine Einflußnahme ist nicht vorgesehen. Das 3.0-Modell fügt ebenfalls Erweiterungen hinzu wie die Abfragemöglichkeit der Seite eines Polygons (Vorder-/Rückseite), dynamische und statische Flußkontrolle und die Möglichkeit, aus dem Vertexshader heraus eine Textur anzusprechen.

Da es sich um eine Definition des kleinsten gemeinsamen Vielfachen handelt, sind natürlich auch größere Werte als die in der Tabelle angeführten erlaubt. Zu der Definition der Assemblerprogramme gehören noch einzelnen Opcodes wie beispielsweise MOV (move register), MAD (multiply and add) oder DP3 (3-component dot product), die wie bei

einer CPU nacheinander[2] abgearbeitet werden und das eigentliche Assemblerprogramm ausmachen.

Die Schwierigkeit für einen Entwickler liegt darin, daß zwar eine gemeinsame Basis existiert, aber die Hardware unterschiedliche Eigenarten bei der Ausführung der Instruktionen hat, so daß es zum Teil nötig ist, die Assemblerinstruktionen auf eine bestimmte Plattform zu optimieren. Darüber hinaus müssen für jedes Shadermodel neue Assemblerprogramme erstellt werden, was mühsam und fehleranfällig ist. Man bedenke, daß eine Anwendung für eine Reihe von Plattformen getestet sein muß.

Diese Probleme sind nicht neu und führten bei den PC-CPUs zu der Entwicklung von Hochsprachen mit optimierenden Compilern, welche den Assemblercode für eine bestimmte Plattform erzeugen, ohne daß der Entwickler die genauen Details kennen muß.

4.4. HLSL/CG

Die beiden Shader-Hochsprachen HLSL und CG [MGAK03] können in einem genannt werden, da die verwendeten Sprachelemente bei beiden gleich angelegt ist. Die Ähnlichkeit ist natürlich kein Zufall, sondern liegt in der Zusammenarbeit von NVidia (für CG) und Microsoft (für HLSL) begründet. HLSL kann dabei auch als DirectX-High-Level Shadingsprache bezeichnet werden, da der Compiler Bestandteil der DirectX-Hilfsbibliothek ist, welche für die Verwendung fest in die Anwendung einkompiliert werden muß. Bei CG hingegen befindet sich der Compiler in einer gemeinsam genutzen Bibliothek.

HLSL setzt Win32/DirectX voraus, während CG auch in OpenGL (Win32/Linux) genutzt werden kann. Die Verarbeitung erfolgt dabei wie in Bild 4.1 dargestellt.

Die Anwendung übergibt dem Compiler den Quellcode als Zeichenkette. Der Compiler erzeugt daraus den Assemberquelltext unter Angabe eines Profils. Bei HLSL ist das die zu verwendende Vertex-/Pixelshaderversion, bei CG entspricht es der GPU (z.B. fp30/fp40 für NV30/NV40-GPUs). Bei CG gibt man darüber auch die Ziel-API an (DirectX oder OpenGL).

Die jeweilige API wiederum reicht den Assemblerquelltext an den Grafikkartentreiber weiter. Dieser besitzt nun einen Assembler, welcher den endgültigen ausführbaren Code erzeugt. Dies muß im Treiber geschehen, da dieser Code GPU-spezifisch ist. Der Treiber kümmert sich auch um die Aktivierung und Aufteilung des Codes auf die einzelnen Pipelines.

4.5. GLSL

Für OpenGL existierte durch CG bereits die Möglichkeit einer Shader-Hochsprache. Viele Firmen wollten sich jedoch nicht der Kontrolle durch NVidia aussetzen, da die Rechte an CG bei Nvidia liegen. Es existiert ein freier Parser, welcher allerdings keinen Einblick in die Internas der Profile erlaubt. Ein nicht-NVidia-spezifisches Profil hätte daher die Zustimmung NVidias finden müssen was sicherlich nicht im Interesse der Mitbewerber lag (Veröffentlichung von Implementationsdetails).

[2] ähnlich einer CPU im PC wird aber auch hier versucht, aufeinanderfolgende Instruktionen parallel in der Recheneinheit auszuführen wenn sie sich nicht gegenseitig beeinflussen.

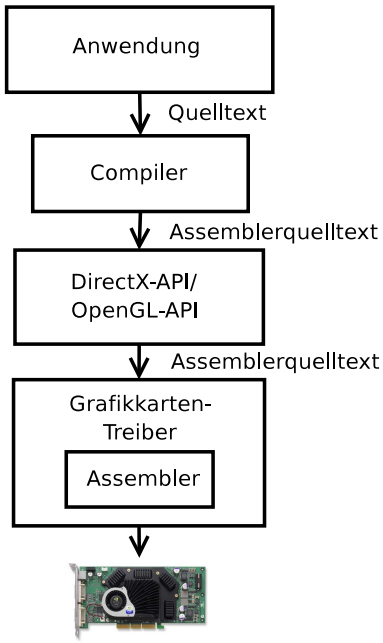

Abbildung 4.1.: Schema Verarbeitung HLSL/CG

Das OpenGL Architecture Review Board (abgekürzt *ARB*, dabei handelt es sich um eine Gruppe von Vertretern der Computergrafikindustrie) entschied sich daher für ein eigenes unabhängiges System mit einer direkten Anbindung an den OpenGL-Standard. Man nannte es OpenGL-Shading Language oder kurz GLSL. Ein dazugehöriger Standard [Ros04] spezifiziert die Sprache.

Lange Zeit nur als ARB-Erweiterung vorhanden, wurde sie mit der Spezifikation von OpenGL Version 2.0 mit in die Kern-API aufgenommen. Treiber auf OpenGL 2.0-Stand sind aber zum aktuellen Zeitpunkt (Ende 2004) noch nicht in breitem Maße verfügbar, es wird aber damit gerechnet, daß sich dies in den nächsten Monaten ändert. Da die Erweiterung aber bereits von allen relevanten Firmen unterstützt wird, ist das Abwarten auch nicht zwingend nötig.

Im Vergleich zu der Abbildung für HLSL/CG ist in Abbildung 4.2 die Umsetzung für GLSL zu sehen.

Der Compiler ist dabei Bestandteil des OpenGL-Treibers geworden. Eine Zwischenausgabe in einen Assemblerquelltext ist nicht vorgesehen[3]. Dies hat den Vorteil, daß der einmal geschriebene Quelltext direkt von neuen Merkmalen der GPU profitieren kann, ohne daß eine Änderung der Anwendung nötig ist. Der Nachteil (für ein Unternehmen) ist allerdings, daß der Quelltext des Shaderprogramms (aber nicht der Anwendung!) dabei immer in einer gewissen Form beigelegt werden muß. Eine binäre Variante ist bislang nicht vorgesehen.

[3]OpenGL definiert zwar den Aufbau von Vertex/Fragment-Assemblerprogrammen, allerdings sind diese nicht Bestandteil der Kern-Bibliothek, sondern existieren in Form von OpenGL-API-Erweiterungen, welche vom ARB standardisiert wurden.

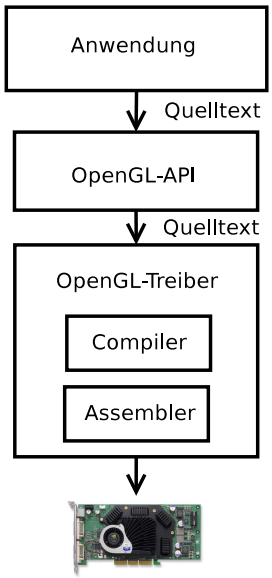

Abbildung 4.2.: Schema Verarbeitung GLSL

Da Vertex- und Fragment-Programm immer paarweise zusammenarbeiten, werden sie in GLSL als ein gemeinsames Programmobjekt behandelt, welches aus dem Quelltext kompiliert und zusammengelinkt wird. Diese Sichtweise ist aber nur eine Abstraktion und entspricht nicht notwendigerweise dem tatsächlichen Ablauf innerhalb des Treibers, so daß im Schaubild ein "Linker" nicht als eigenständiges Element auftritt.

4.6. Shadersprache

Da eine Beschreibung des kompletten Sprachstandards den Rahmen sprengen würde, soll an dieser Stelle nur eine kurze Einführung folgen. Die Sprache ist sehr stark an C/C++ angelehnt und daher für einen C/C++-Programmierer schnell zu erlernen. Der Anteil von C++ ist allerdings zum jetzigen Stand gering, das Überladen von Funktionen ist möglich und auch der bool-Typ wurde von C++ entnommen. Eine Klassendefinition gibt es nicht, allerdings ist das class-Schlüsselwort reserviert. Es gibt Arrays und Strukturen und die Definition und Deklaration einer Variablen ist ebenfalls an C angelehnt. Man verfügt auch über die bekannten Vergleichsoperatoren (==, <, >, !, ...) und Schleifenkonstrukte (*for*, *while*, *do-while*). Funktionen können Parameter und einen Rückgabewert haben. Zeiger hingegen existieren nicht.

Einige Elemente wurden hinzugefügt, um den speziellen Anforderungen eines GPU-Programms gerecht zu werden. So gibt es neben den skalaren Datentypen (*float*, *int*, *bool*) spezielle Vektordatentypen (*vec2*, *vec3*, *vec4*,...), 2x2, 3x3 und 4x4-Matrizen können ebenfalls direkt als Variable deklariert werden (*mat2*, *mat3*, *mat4*).

Um den Umgang mit Texturen zu erleichtern, wurden spezielle Sampler-Datentypen eingeführt *(sampler1D*, *sampler2D*, *sampler3D*, *samplerCube*, *sampler1DShadow, sampler2DShadow*). Auf diese Weise kann die Anwendung die Textureinheiten dynamisch ver-

walten. Die "*Shadow*"-Sampler sind speziell für Schattentexturen ausgelegt. Diese Texturen werden aus Sicht der Lichtquelle berechnet und enthalten nur die Tiefeninformation der Objekte. Auf diese Weise kann dann über die Entfernung[4] des Punktes von der Lichtquelle die Aussage getroffen werden, ob eine direkte Beleuchtung durch diese Lichtquelle vorliegt oder nicht.

Zu den üblichen mathematischen Operatoren (+, -, /, *) kommt der "Swizzle"-Operator. Mit diesem lassen sich die Vektordatentypen komponentenweise ansprechen (*color.r*, *color.g*, *color.b*). Die Ähnlichkeit zum Punktoperator einer Struktur ist sicherlich nicht zufällig. Allerdings lassen sich die Komponenten beliebig kombinieren, *"color.rgb"* zum Beispiel stellt direkt einen Vektor mit drei Komponenten dar und auch *"color.bgr"* ist möglich, genauso wie *"color.rrr"*. Für den leichteren Umgang mit Vektoren wirken sich die mathematischen Operatoren auch auf alle Komponenten aus, ein *"w=u+v"* mit vec3-Variablen bedeutet ausgeschrieben *"w.x = u.x + v.x; w.y= u.y + v.y; w.z = u.z + v.z"*. Die Multiplikation zweier Matrizen bildet das mathematisch korrekte Matrixprodukt, das gleiche gilt für die Multiplikation eines Vektors mit einer Matrix (je nach Reihenfolge als Zeilen- oder Spaltenvektor).

Der von C bekannte Präprozessordurchlauf existiert ebenfalls, so daß Bereiche mit "#if" und "#endif" ausgeklammert werden können und Makros per "#define" möglich sind.

Ein wichtiger Punkt sind die Variablenqualifizierer *const*, *attribute*, *uniform* und *varying*. *const* ist bereits aus dem C-Sprachstandard bekannt und bezeichnet eine konstante Variable (sie kann nur gelesen werden). *Attribute*-Variablen werden für Informationen verwendet, welche sich für jeden Eckpunkt eines Polygons ändern können. *Uniform*-Variablen hingegen für Informationen, die für ein Polygon oder sogar den gesamten Durchlauf hinweg konstant bleiben (das Schreiben dieser Variablen kann mehr GPU-Zeit kosten als das Schreiben einer *attribute*-Variablen). Aufgrund der höheren Anforderungen sind *attribute*-Variablen nur für Gleitkommadatentypen möglich. Sowohl *attribute* als auch *uniform* sind für die Kommunikation zwischen Anwendung und Shader vorgesehen und können daher innerhalb des Shaders nicht mit einem anderen Wert überschrieben werden. Auf *attribute*-Variablen kann darüber hinaus nur innerhalb eines Vertex-Programms zugegriffen werden.

Varying-Variablen dienen der Kommunikation zwischen Vertex- und Fragment-Programm. Da ein Fragment üblicherweise das Ergebnis der Interpolation zwischen den Angaben mehrerer Eckpunkte ist, werden die Variablen ebenfalls entsprechend perspektivisch korrekt interpoliert.

Neben der Möglichkeit, eigene Funktionen zu definieren gibt es eine Reihe von Standard-Funktionen, die einen engen Bezug zur Shader-Berechnung haben. So gibt es geometrische Funktionen (*dot, cross*, *length*, *normalize*, ...), mathematische Funktionen (*sin*, *cos*, *exp2*, ...), Vektorvergleichsoperationen (*lessThan*, *lessThanEqual*, *greaterThan*...), Texturzugriffsfunktionen (*texture2D*, *textureCube*, ...) und noch einige mehr.

Diese kurze Zusammenfassung soll für das Verständnis der Programme dienen, kann aber natürlich keine vollständige Abhandlung des Themas darstellen. Der Standard ist in [JK03] beschrieben, eine detailliertere Einführung anhand von Beispielen in [Ros04], welches auch als das sogenannte "Orange Book" bezeichnet wird[5].

[4] in der Praxis geschieht dies über eine Transformation der Objektkoordinaten in das Koordinatensystem der Schattentextur

[5] die dazugehörigen Bücher sind: Blue Book [BS04] (OpenGL-Sprachreferenz) und Red Book [BS03] (OpenGL-Einführung und Beispiele)

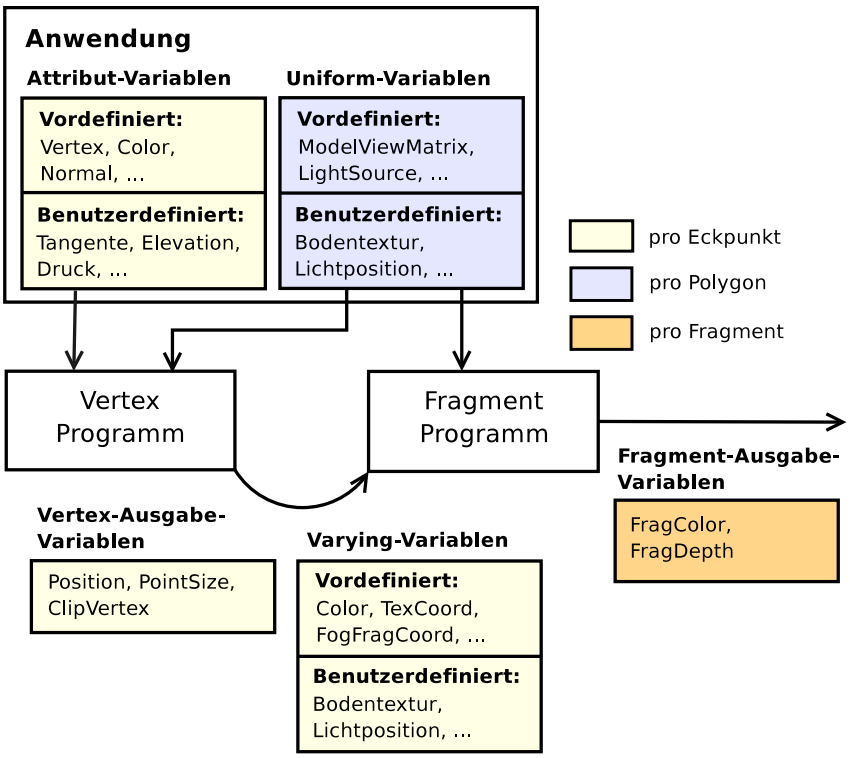

Abbildung 4.3.: Zusammenspiel Anwendung, Vertex- und Fragmentshader

4.7. Vertex-Programm

Die Abbildung 4.3 zeigt das Schema des Zusammenspiels von Anwendung, Vertex- und Fragmentshader.

Der Vertexprozessor ruft eine festgelegte Startfunktion im Vertex-Programm auf. Ihr Prototyp heißt wie von C zu erwarten “void main()”. Diese Funktion wird dann komplett bis zum Ende durchlaufen. Es gibt nur eine festgelegte Variable, die vom Vertex-Programm geschrieben werden muß (“gl_Position”). In dieser werden die transformierten Eckpunktkoordinaten erwartet. Das kürzeste Vertex-Programm lautet daher:

```
void main(){
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Für die Kommunikation mit der Anwendung können innerhalb des Vertex-Programms Variablen sowohl als *attribute* wie auch als *uniform* qualifiziert werden. Es gibt eine Reihe von vordefinierten Variablen, die impliziert deklariert sind, so daß sie direkt verwendet werden können. Beispiele wären die Vertex-Position selber (als “vec4 gl_Vertex”), der dazugehörige Normalenvektor (“vec3 gl_Normal”) oder die Transformationsmatrizen (“mat4 gl_ModelViewMatrix”).

Die Anwendung übergibt diese vordefinierten Variablen mit Hilfe der normalen OpenGL-Funktionen, so daß sie nahtlos mit fester Funktionalität verwendbar sind.

Für die *attribute*/*uniform*-Variablen wurde der OpenGL-Standard um entsprechende Funktionen (*glVertexAttrib*/*glUniform*) erweitert, die sich in die vorhandenen Funktionen für die vordefinierten Variablen eingliedern. Auf diese Weise stellen sie eine natürliche Erweiterung der vorhandenen API dar.

Der Versuch, eine *uniform*-Variable für jeden Eckpunkt zu ändern, könnte je nach Implementierung einen erheblichen Performanzeinbruch mit sich bringen und sollte vermieden werden.

Zu den Aufgaben des Vertex-Programms können gehören:

- Transformation der Eckpunkte und Normalen
- Erzeugung und Transformation der Texturkoordinaten
- Transformation und Berechnung von Beleuchtungswerten pro Eckpunkt

"Können", weil es dem Shaderprogrammierer frei steht, die Abfolge nach seinen Wünschen zu gestalten und gewisse Punkte wegzulassen oder durch alternative Berechnungen zu ersetzen.

Die erwähnten *varying*-Variablen werden deklariert und benutzt wie jede andere Variable auch, die Interpolation erfolgt vollkommen transparent und liegt nicht im Einflußbereich des Vertex-Programms. Es ist auch nicht möglich, einen Eckpunkt wegzuwerfen oder neue, zusätzliche Eckpunkte zu generieren. Es wird immer exakt ein Eckpunkt als Eingabe übergeben und (transformiert) zurückerwartet.

4.8. Fragment-Programm

Nachdem alle Eckpunkte eines Polygons abgearbeitet sind, werden die ermittelten Koordinaten verwendet, um die dazugehörige Polygonfläche zu berechnen. Durch das Ergebnis des Clippings oder der Ermittlung der Polygonausrichtung (zum Betrachter hin oder von ihm abgewendet) kann es passieren, daß eine Fläche nicht oder nur teilweise gezeichnet wird, obwohl alle Eckpunkte berechnet wurden.

Im Gegensatz zum Vertex-Programm ist es einem Fragment-Programm möglich, ein Fragment wegzuwerfen, indem "discard" aufgerufen wird. Die Kommunikation mit der Anwendung findet in gleicher Form wie mit dem Vertex-Programm über *uniform*-Variablen statt. Da ein Vertex- und ein Fragmentprogramm für einen Materialdurchlauf eine Einheit bilden, können sie eine solche Variable unter gleichem Namen deklarieren, so daß zum Beispiel die Lichtposition nur einmal von der Anwendung geschrieben werden muß.

Das Fragment-Programm kann nun auf die *varying*-Variablen, welche vom Vertexprozessor geschrieben und von der nachfolgenden Stufe interpoliert wurde, zugreifen. Da die perspektivischen Informationen bereits vollständig bekannt sind (alle benötigten Eckpunkte wurden ausgewertet) kann auch auf die Ableitungen dieser Variablen zugegriffen werden, um zum Beispiel die Richtung der Normalenänderung für das Bump- oder Normalmapping zu bestimmen oder um bei zu großen Texturkoordinatenänderungen Antialiasing-Maßnahmen zu ergreifen.

Das kürzestmögliche Fragment-Programm ist:

```
void main(){
    gl_FragColor = vec(1.0,0.0,0.0,1.0);
}
```

Die Einstiegsfunktion ist wie beim Vertex-Programm “void main()”. Da die Programme beim Laden durch die OpenGL-API eindeutig mit ihrem Typ angegeben werden müssen, ist die Zuordnung problemlos möglich. Das Beispielfragment-Programm liefert für jedes Fragment den gleichen Farbwert (100% Rot) zurück, so daß es in dieser Form wohl selten verwendet werden wird.

Die Aufgaben eines Fragment-Programms bestehen im:

- Auslesen und Verknüpfen von Texturinformationen
- Berechnung des Farb- und Alphawertes für einen Pixel
- evtl. Schreiben des Tiefenwertes (wird aber üblicherweise von der Hardware übernommen)

Es bleibt noch zu erwähnen, daß der Fragmentshader bei einem Zugriff auf die Textur über die entsprechende Funktion immer auf die zum aktuellen Fragment gehörende Detailstufe zugreift, falls eine entprechende Mipmapstufe definiert wurde. Man kann allerdings zu dem berechneten Detailwert einen konstanten Wert aufaddieren, so daß man auch auf jede andere Mipmap Zugriff bekommt. Der Zugriff auf die Textur erfolgt nicht in absoluten Pixelwerten sondern normalisiert, die Werte gehen von 0.0 bis 1.0. Auf diese Art kann der Zugriff auf jede Textur unabhängig von Größe und Detailstufe in gleicher Weise erfolgen.

5. Implementierung

5.1. Konzept

Nachdem die theoretischen Grundlagen vom Autor erarbeitet wurden, folgt nun die Beschreibung der eigentlichen Implementierung. Die Beschreibung gliedert sich in 2 Teile, einmal die Implementierung des VideoTextur-Plugins und zum anderen die Verwendung der Texturen bei der Beleuchtung.

Vor der Implementierung wurde ein Konzept entwickelt und die benötigte Software wurde ausgewählt. Die Punkte, die von der Anwendung umgesetzt werden sollten, wurden wie folgt festgelegt

- Darstellung des Zuginnenraums mit möglichst großer Bewegungsfreiheit.
- Laden von Bildsequenzen (evtl. auch komprimiert) als Videotextur mit möglichst flexibler Parametrisierung.
- Verwendung einer Sequenz zur Beleuchtung des Zuginnenraums.
- Verwendung von Sequenzen innerhalb des Zuges als Szenenelement, die Beleuchtung sollte sich auch auf diese Elemente auswirken.

Um diese Punkte umzusetzen war ein Grafiksystem nötig, welches Szenendaten und Shader verwalten konnte. Die Objekte lagen als Maya-Dateien vor, hier galt es, einen geeigneten Exporter zu finden/schreiben. Da die GLSL-Shadersprache noch nicht lange in die Treiber integriert worden war (NVidia-Treiber: Juli 2004) boten kommerzielle Produkte wie die OpenInventor-Umgebung diese erst in einer Betaversion an.

Da die Anwendung von ihrer Komplexität überschaubar war, wurden einige Rahmenwerke vom Autor erstellt. Diese boten allerdings nur geringe Erweiterungsmöglichkeiten. Die Entscheidung fiel daher für die OpenSource Grafikberechnungs-Bibliothek Ogre nachdem diese die GLSL-Unterstützung in einer stabilen Version (0.15.0 bzw. 0.15.1) anbot.

Nach Sichtung der Ogre-API wurde deutlich, daß die Integration der Videotexturen in diese Bibliothek am sinnvollsten in Form eines Plugins durchzuführen war.

Es existierte bereits ein freies Exporter-Plugin für Maya, so daß der Austausch hier problemlos ohne eigene Erweiterung möglich war.

5.2. Materialkonfiguration

Die Konfiguration von Ogre erfolgt zum Teil durch Methodenaufrufe aus der Anwendung heraus, zum anderen Teil durch Textdateien im Anwendungsverzeichnis. Dazu gehört auch die Definition der verwendeten Materialien. Ein Material ist dabei eine vollständige Beschreibung der für die Oberflächenberechnung notwendigen Parameter.

Es setzt sich aus diversen Einträgen zusammen, die den folgenden Grundaufbau besitzen:

```
material Materialname
{
    technique
    {
        pass
        {
            [...]
        }
    }
}
```

Die Zuordnung von Material zu Shader erfolgt bereits in Maya über den Namen des dort verwendeten Shadermaterials. Es ist auch möglich, jedem Mitglied der Gruppenhierarchie ein eigenes Material zuzuweisen.

Der Export erfolgt in ein XML-Format, welches für den endgültigen Gebrauch dann in eine binäre "mesh"-Datei umgewandelt wird und von Ogre über API-Aufrufe geladen wird.

Innerhalb des Materials tauchen *technique* und *pass* auf. Es können mehrere *technique*-Ebenen existieren (zum Beispiel eine shaderlose Berechnung für einfachere Hardware) und diese wiederum können mehrere *passes* besitzen. Ein *Pass* ist dabei ein kompletter Zeichendurchlauf für alle darzustellenden Objekte. Mehrere Durchläufe sind zum Beispiel nötig, wenn die Anzahl der gleichzeitig aktiven Textureinheiten nicht ausreicht, um alle gewünschten Elemente einzubinden. Ein anderes Beispiel ist die Anwendung mehrerer Shader auf eine Oberfläche (pro Durchlauf kann immer nur ein Shaderpaar pro Material aktiv sein) oder bei einer Schattenberechnung mit Hilfe einer Tiefentextur. Die Kombinationsart des jeweiligen Durchlauf-Ergebnisses wird ebenfalls angegeben (zum Beispiel: addieren, multiplizieren, alphablending, ...).

Die Materialdefinition auf diese Weise (*technique*/*pass*) durchzuführen ist eine anerkannte Methode, eine detailierte Beschreibung findet sich in Kapitel 36 in [Fer04].

Innerhalb eines Durchlaufs lassen sich umfangreiche Einstellungen für die feste Funktionalität vornehmen (Lichteinstellungen, Verknüpfung von Textureinheiten, ...). Shader-Quelltext wird ebenfalls per Material-Eintrag eingebunden und die in 4.6 beschriebenen *uniform*-Variablen können an dieser Stelle auch statisch bestimmt werden (aber natürlich auch per Programmcode später dynamisch geändert werden).

5.3. Videotextur-Plugin

Das angestrebte Ziel war die Darstellung einer gespeicherten Videosequenz innerhalb der 3D-Szene. In der Vorüberlegungsphase wurden dafür die folgenden Bedingungen definiert:

1. Das Material sollte in unterschiedlichen Auflösungen vorliegen können. Dabei sollte auch die Möglichkeit gegeben sein, einen eventuell existierenden Alphakanal zu benutzen.

2. Die Wiedergabe sollte möglichst mit der bei Aufzeichnung verwendeten Bildwiederholrate erfolgen (im Normalfall 25 Vollbilder/sek)

3. Das Material sollte durch einen parallel ablaufenden Thread im RAM zwischengespeichert werden, um unabhängig von der Festplattenladezeit zu werden.

4. Die Video-Textur sollte von Shader-Programmen ausgelesen und als Beleuchtung verwendet werden können.

Der erste Punkt konnte nur mit Einschränkungen realisiert werden, da bei OpenGL unterhalb von Version 2.0 eine Textur in jede Dimension einer Zweierpotenz entsprechen muß. Die Übernahme einer 720x576 großen Videotextur war also nicht möglich, es wurde daher als Kompromiß eine Auflösung von 512x512 gewählt. Der Alphakanal konnte dagegen problemlos übernommen werden, die Hardware unterstützt dies sowohl beim BGRA-Format[1] als auch bei den später beschriebenen komprimierten Formaten.

Der zweite Punkt läßt sich nur erfüllen, wenn die produzierte Anzahl Bilder größer oder gleich der Vorgabe der Bildsequenz ist. Ist dies der Fall, so sorgt eine Zeitabstandsmessung für die Umsetzung. Dies wird später noch als Flußdiagramm näher beschrieben. Ist die Bilderrate hingegen zu klein, so wird die Videotextur zu langsam aber dafür komplett wiedergegeben (es kommt nicht zu Sprüngen).

Punkt 3 konnte umgesetzt werden, bedingte aber die Duplizierung von eigentlich bereits vorhandenen Ogre-Methoden, welche für das Laden und Parsen von Bildern verantwortlich sind, da sie leider noch nicht threadsicher[2] angelegt waren. Aus diesem Grund muß auch der Pfad zu den Bildsequenzen innerhalb des Materialskriptes vollständig angegeben werden, normalerweise ist man bei Ogre in der Lage, dies zentral über einen Ressourcemanager zu erledigen. Andererseits führt diese Trennung bei den komprimierten DDS-Bildern zu einem Geschwindigkeitsvorteil, da die Routinen rein auf das Laden der Daten optimiert wurden und nicht auf eine eventuell durchzuführende Dekompression (innerhalb des PC-RAMs).

Der letzte Punkt - die Beleuchtung durch die Shader - stellt den zentralen Teil der vorliegenden Arbeit dar und wird ausführlich in Kapitel 5.8ff. dargestellt.

5.4. Textur-Kompression

Das BGRA-Format könnte wie erwähnt zwar direkt verwendet werden, allerdings ergibt sich bei einer 512x512 Textur rechnerisch pro Bild eine Datenmenge von 1 MByte. Dazu kommt noch eine gewisse Anzahl Bytes für die Kodierung der Bildinformation. Vernachlässigt man letzteres, so ergeben sich bei 25 Bildern pro Sekunde eine Menge von 25 MByte pro Sekunde. Eine kontinuierliche Übertragung würde eine heutige Festplatte bereits auslasten.

[1] BGRA/RGBA sind zwei verschiedene Arten die einzelnen Farbkanäle und den Alphakanal anzuordnen, die Information bleibt die gleiche. Allerdings ist laut Support BGRA das native Format zumindest für die verwendete NVidia-Hardware.

[2] Threadsicherheit bedeutet zum Beispiel, daß eine Variable, die in einem Thread noch verwendet wird nicht gleichzeitig in einem anderen Thread geändert werden kann, da dies möglicherweise zu undefinierten Zuständen führt.

Name	Alpha	Kompression	vormultipliziert
DXT1	0 oder 2 Stufen	8:1	nein
DXT2	16 Stufen	4:1	ja
DXT3	16 Stufen	4:1	nein
DXT4	8 interpolierte Stufen	4:1	ja
DXT5	8 interpolierte Stufen	4:1	nein

Tabelle 5.1.: DXT-Formateigenschaften

Eine alternative Methode wäre es, die Daten nur in der Auflösung anzufordern, die auch benötigt wird. Diese Information müßte aber von der Anwendung ermittelt und an den Videotextur-Cache übermittelt werden.

Es wurde daher als Alternative die Verwendungsmöglichkeit von Kompressionsverfahren geprüft. Die für Abspielmedien wie DVD oder Internet-Daten üblichen Verfahren wie MPEG2 oder MPEG4 bieten eine hohe Kompressionsrate, werden aber vom 3D-Grafiksystem nicht unterstützt, so daß die Dekodierung in Software erfolgen müßte.

Um den Aufwand für die Dekodierung gering zu halten, wurde daher das DXT-Verfahren verwendet, welches schon seit einigen Jahren von fast allen Grafikkarten in Hardware dekomprimiert werden kann. Dieses Verfahren, welches ursprünglich von der Firma S3 entwickelt wurde (und daher auch als S3TC-Kompression bekannt ist) ist in 5 Varianten verfügbar (genannt DXT1-DXT5). Microsoft hatte es für die hauseigene DirectX-Grafikschnittstelle lizensiert und für diese Schnittstelle auch ein Dateiformat namens DDS (*DirectDrawSurface*) spezifiziert, welches dafür gedacht war, direkt in die DirectX-API geladen zu werden. Es kann daher neben unkomprimierten Pixelformaten auch die DXT-Formate speichern und erlaubt darüber hinaus die Speicherung von mehreren Mipmapstufen in einer Datei. Dieses Format wurde auch für die vorliegende Anwendung als Speicherformat für komprimierte Bildsequenzen verwendet. Für unkomprimierte Bildsequenzen wurde hingegen das TGA-Format gewählt, da es sich schnell parsen läßt und die Daten direkt verwendet werden können.

Die Eigenschaften der DXT-Formate sind in Tabelle 5.1 dargestellt. Die vormultiplizierten Formate werden von keiner dem Autor bekannten Hardware/Grafik-API unterstützt und sind nur der Vollständigkeit halber aufgezählt. Das DXT-Verfahren teilt die Pixel in einem Bild in 4x4 große Elemente, so genannte Texels *(texture elements)* auf. Aus diesen 16 Farbwerten werden nun 2 Referenzwerte ermittelt und der Bereich dazwischen in 4 Stufen unterteilt, so daß die 16 Farbwerte des Originaltexels mit je 2 Bit kodiert werden können. Die vollständige Information für einen Texel besteht dann aus 2 Farbwerten (RGB565-kodiert, also 2*16 Bit) und 16*2 Bit für jeden Pixel. Daraus ergibt sich eine Kodierung von 4 Bits pro Pixel. Bei DXT4/5 wird dieses Verfahren auch für den Alphakanal angewendet, allerdings in der Kombination 2*8 Bit Referenz und 16*3 Transparenzbit für einen Texel. Für die vorliegende Anwendung kam dennoch das DXT3-Format zum Einsatz, da die Übergänge zwischen den Alphawerten bei der Szene der Zugpassagiere sehr steil ist und es daher besser ist, einen gleichmäßig in 16 Stufen unterteilten Bereich in der ganzen Szene zu haben als nur 8.

Es liegt auf der Hand, daß dieses Verfahren nur zufriedenstellend funktioniert, wenn

Abbildung 5.1.: Vergleich RGB (von DV-Material) und DXT3

alle 16 Pixel sich durch Interpolation der beiden Referenzfarbwerte ihrer ursprünglichen Einfärbung entsprechend darstellen lassen. Um ein Gefühl für die Kompressionseffekte zu bekommen, wurde in Abbildung 5.1 ein Vergleich zwischen RGB mit 8 Bit pro Farbkanal und dem daraus konvertierten DXT3 dargestellt. Allerdings muß berücksichtigt werden, daß das Quellmaterial ursprünglich von DV stammt, welches eine 4:2:0-Abtastung im YUV-Farbraum vornimmt. Dadurch ist die Auflösung der Farbinformation in vertikaler und horizontaler Richtung um die Hälfte reduziert. Diese Reduktion bleibt bei der Transformation nach RGB natürlich erhalten.

Im direkten Vergleich der beiden Bilder ist der Unterschied kaum wahrnehmbar. Erst die starke Vergrößerung zeigt die Änderungen in der Pixelstruktur, die aus der Reduktion auf 4 mögliche Farbwerte pro Texel resultiert.

In Tabelle 5.2 ist ein grober Vergleich zwischen den Ladezeiten der einzelnen Texturformate dargestellt. Der Eintrag “ohne Mipmap” heißt, daß nur das Basisbild in den Speicher geladen wurde, “Automipmap” bedeutet, daß beim Laden zusätzlich von der Grafikkarte selber Mipmaps erzeugt wurden.

Die Auflösung der Bilder ist als 2er-Potenz in Klammern angegeben. Bei “Automip-

Texturformat	Nvidia FX5600-128MB 325/550 MHz
B8G8R8 (2^9) ohne Mipmap	14-15 ms
B8G8R8 (2^9) Automipmap	17-19 ms
B8G8R8 (2^8) ohne Mipmap	3.5 ms
B8G8R8 (2^8) Automipmap	4 ms
DXT3 (2^9) ohne Mipmap	1-2 ms
DXT3 (2^9) Automipmap	52-55 ms
DXT3 (2^8) ohne Mipmap	0.2-0.3 ms
DXT3 (2^8) Automipmap	12-14 ms

Tabelle 5.2.: Vergleich Ladevorgang Texturdaten

map" wurde immer die höchste Anzahl von gefilterten Bildern erstellt (im Falle von 2^9 (Bildauflösung 512x512) also beispielsweise 9).

Der Test zeigte, daß es - zumindest für die getestete Hardware/Treiber-Kombination - von Nachteil ist, die Mipmaps für das DXT3-Bild über die Hardware zu erzeugen und man diese besser mit in die Bilddatei abspeichert. Die Zunahme an Bilddaten stehen in keinem Verhältnis zur Zunahme der Ladezeit. Die große Verzögerung beruht eventuell darauf, daß die Kompression von der Hardware vorgenommen wird und der Treiber zwischen Grafikkarten-RAM und PC-RAM hin- und herkopiert und das Laden einer Textur aus der Grafikkarte heraus über den AGP-Bus mit großen Leistungseinbußen verbunden ist.

5.5. Videotextur-Implementierung

Eine Schnittstellenklasse für externe Texturquellen ist bereits in der Ogre-API verankert, diese implementiert die Funktionen, die der Einbindung eines externen Plugins (zu deutsch in etwa "Steckmodul") dienen.

Auf Programmierebene erfolgt die Einbindung des Plugins durch die Ableitung von den benötigten internen Klassen und der Kompilierung als dynamische Bibliothek (Dateierweiterung .so unter Linux, .dll unter Windows). Diese dynamische Bibliothek wird dann von der Ogre-Engine bei Programmstart geladen, insofern der "plugins.cfg" ein entsprechender Eintrag hinzugefügt wurde.

Das Konzept für das VideoTexture-Plugin sah keine direkte Kommunikation der Anwendung mit dem Plugin vor, sondern alle Angaben sollten über die Shadermaterial-Definition erfolgen. Dies erwies sich als flexible Lösung, da die Videotexturen somit ohne Neukompilierung der Anwendung austauschbar und parametrisierbar sind.

Abbildung 5.2 zeigt eine grobe schematische Übersicht der Plugin-Einbindung. Die von der Ogre::ExternalTextureSource-Klasse abgeleitete VideoTextureSource-Klasse liegt wie beschrieben in Form einer dynamischen Bibliothek vor. Sie wird bei Programmstart vom ExternalTextureSource-Manager geladen und initialisiert, dabei registriert sie sich als "vidtex"-Texturquelle und fügt dem Ogre-Materialparser eigene Parameter ("imagebase", "imagepath" etc.) mit entsprechenden Setzen/Abfrage-Methoden hinzu.

In der Abbildung rechts ist nur der relevante Ausschnitt des Skripts zu sehen, der

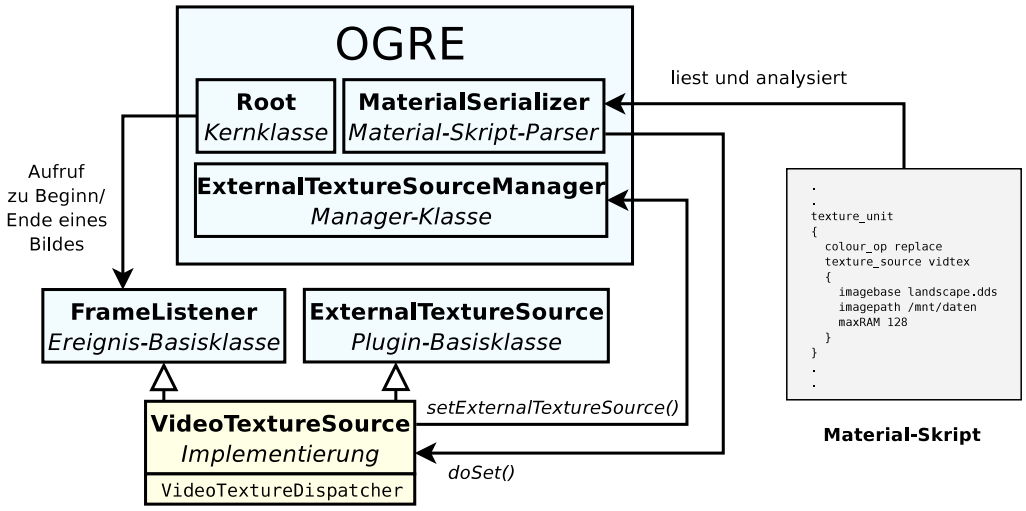

Abbildung 5.2.: VideoTexture-Plugin (Einbindung)

vollständige Aufbau des Material-Skripts wurde bereits in 5.2 beschrieben. Falls sich wie hier gezeigt innerhalb einer “texture_unit” ein “texture_source”-Bereich befindet, so holt sich die MaterialSerializer-Klasse über den ExternalTextureSourceManager einen Zeiger auf die entsprechende texture_source-Quelle und ruft dort die zu den jeweiligen Parametern gehörende doSet()-Methode auf. Auf diese Weise werden die Benutzerparameter zum Plugin kommuniziert und können dort ausgewertet werden.

Außerdem erbt VideoTextureSource von der FrameListener-Klasse die Eigenschaft, zu Beginn/Ende eines Bildes benachrichtigt zu werden.

Alle Parameter, die im Material-Skript verwendet werden können, sind in Anhang A aufgeführt.

5.6. Texturladevorgang aus dem Cache

Um sowohl, wie im Konzept vorgesehen, unabhängig von den Schwankungen der Festplattenladezeit zu sein, als auch - bei entsprechender RAM-Ausstattung - die Möglichkeit zu haben, die komplette Bildsequenz im Speicher zu halten, wurde ein Cache implementiert. Dieser läuft innerhalb eines eigenen Threads, in Abbildung 5.3 ist seine Ausführung in blauer Farbe von der Ausführung des Hauptthreads (schwarz) abgehoben.

Abbildung 5.4 zeigt das Schema der internen Arbeitsweise. Das Füllen des PC-RAM Caches erfolgt anhand der vorgegebenen Parameter. Man gibt sowohl die Größe des RAM-Cache als auch den Bereich der zu ladenden Bilder an. Ist der Bereich von der Anzahl her zu groß für die reservierte RAM-Größe, so lädt der Thread zunächst so viele Bilder wie möglich. Ist der Cache voll, wird der Thread schlafen gelegt und wartet anschließend auf den “Verbrauch” eines Bildes. Sobald dies der Fall ist, wird er geweckt und überschreibt den freien Platz mit einem neuen Bild. Auf diese Weise wird immer ein Ausschnitt der gesamten Bildsequenz im Cache gehalten. Abbildung 5.5 verdeutlicht dies.

Da die Bilder als Bildobjekte sequentiell in einem STL-Container gespeichert werden,

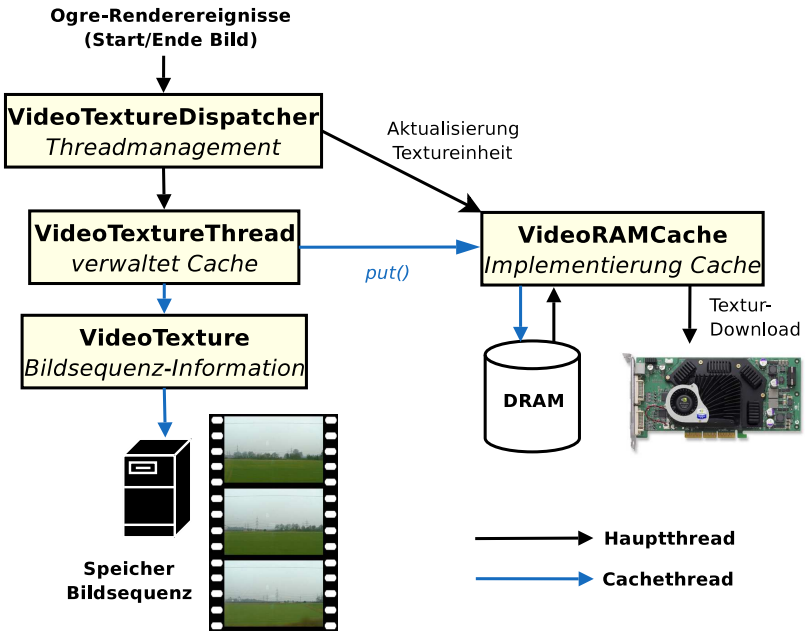

Abbildung 5.3.: VideoTexture-Plugin (interner Aufbau)

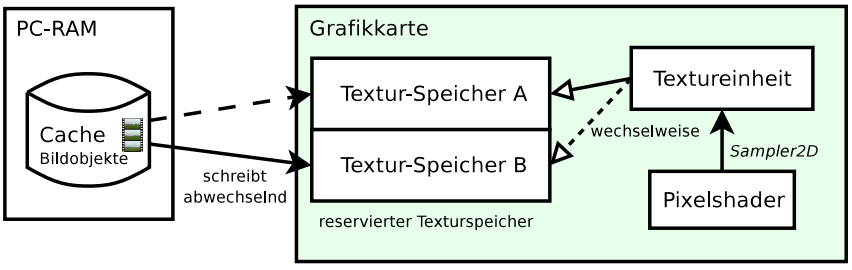

Abbildung 5.4.: Arbeitsweise VideoTexture-Cache

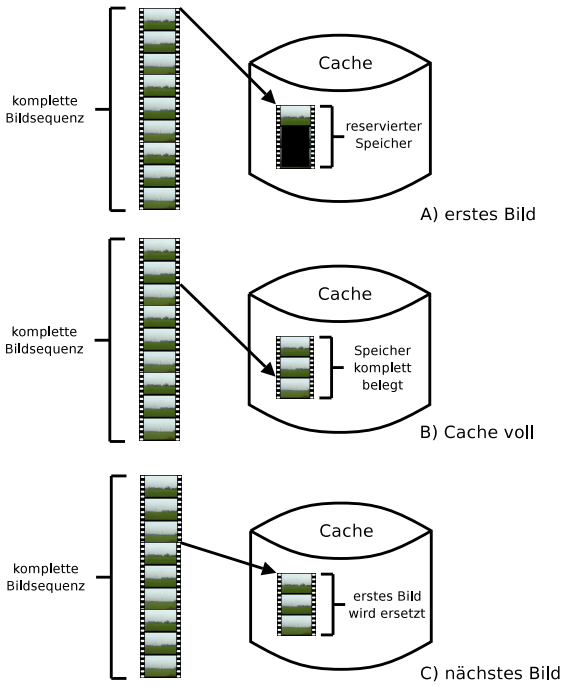

Abbildung 5.5.: Speicherung bei nicht ausreichend großem Videotextur-Cache

ist dies mit Hilfe von zwei Indices realisiert. Ein Index zeigt auf das zu (über)schreibende Objekt, der andere auf das zu lesende. Ein Schreiben ist nur möglich wenn der Cache nicht voll ist. Das Kopieren eines Bildes in das RAM der Grafikkarte erhöht dabei den Zähler für die Anzahl der freien Bilder im Cache um eins falls von nun an das Nachfolgebild gezeigt werden soll. Anschließend wird wie beschrieben der Thread geweckt. Dieser Vorgang wiederholt sich nun, so daß ein ständiges Nachpuffern und Überschreiben erfolgt.

Ist die Anzahl der Bilder hingegen kleiner als das angegebene Cache-RAM, so wäre ein ständiges Nachpuffern nicht sinnvoll. In diesem Fall lädt der Thread bei Programmstart alle Bilder in den Cache und legt sich anschließend bis zum Beenden der Anwendung schlafen, so daß er keine Rechenzeit mehr beansprucht. Die Anwendung kopiert nun der Reihe nach alle Bilder im RAM-Cache abwechselnd in die beiden Texturspeicher. Ist das letzte Bild erreicht, so wird der Index auf das erste Bild des Caches zurückgesetzt und somit läuft die Sequenz fortlaufend ab.

In beiden Fällen wird der Zeitpunkt, an dem das Nachfolgebild gezeigt werden soll über den zeitlichen Abstand zum letzten Bildwechsel ermittelt. Dies ist nötig, da die Grafikkarte zum Beispiel 50 Bilder pro Sekunde zeichnen kann, die Sequenz aber nur 25 Bilder pro Sekunde liefert. Hierbei muß jedes Bild im Cache zweimal gezeigt werden bevor es (für die zu kleine Cache-Variante) entfernt und durch ein neues Bild überschrieben werden kann. Ein grobes Flußdiagramm für die angewendete Logik ist in Abbildung 5.6 ersichtlich. Es handelt sich um das wesentliche Schema, die tatsächliche Implementierung ist komplexer und gliedert sich in mehrere Klassenmethoden auf, da nicht nur eine, sondern mehrere Texturen verwaltet werden müssen.

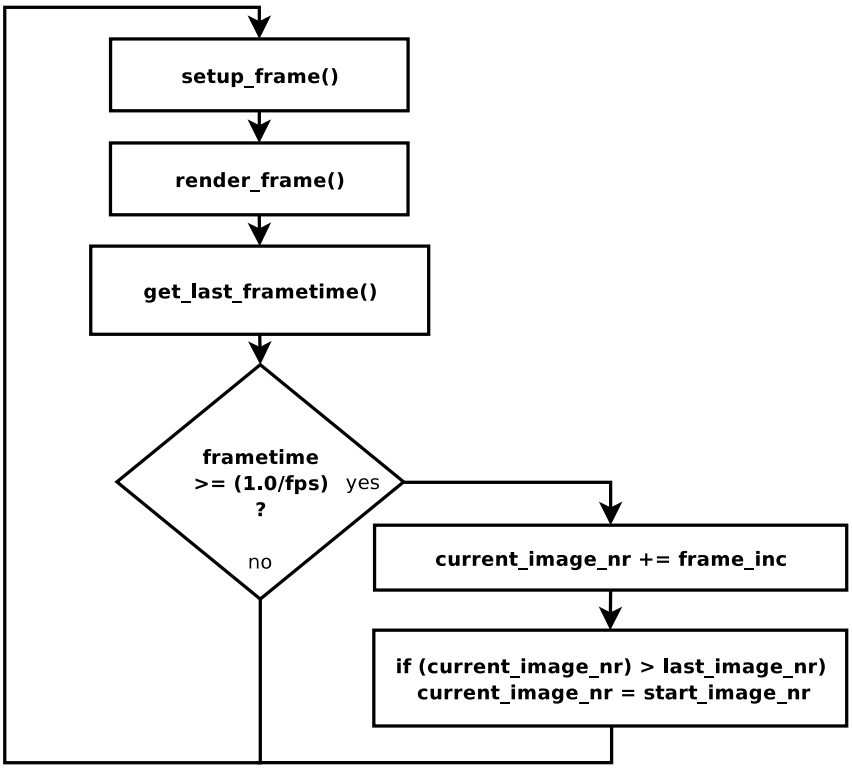

Abbildung 5.6.: Flußdiagramm Erhöhung der Bildnummer im TexturCache

Zusätzlich kann der "frame_inc"-Materialskript-Parameter verwendet werden, um eine gegebene Anzahl Bilder innerhalb einer Sequenz zu überspringen. Auf diese Weise kann eine Sequenz, die zur Wiedergabe mit 25 Bildern pro Sekunde gedacht ist, in nur 12 Bildern (bei einem frameinc-Wert von "2"[3]) pro Sekunde wiedergegeben werden. Natürlich leidet dabei die Wiedergabequalität (es wird "ruckelig").

Abschließend ist noch ein Punkt in Abbildung 5.4 erläuterungsbedürftig. In der Grafikkarte läßt sich der Texturspeicher unabhängig von der Textureinheit reservieren. Dies ist wichtig, da die Anzahl der Texureinheiten begrenzt ist (Minimum bei OpenGL sind 2, im Normalfall werden heutzutage 4 angeboten). Die Zuordnung erfolgt dabei über einen Zeiger auf den jeweiligen Texturspeicher und kann für jeden Durchlauf geändert werden. Innerhalb der in Abbildung 5.6 sichtbaren "render_frame()"-Funktion werden alle Durchläufe für ein Bild durchgeführt. Da die GPU innerhalb der Grafikkarte keine endlich kleinen Rechenzeiten besitzt, ist damit zu rechnen, daß die Berechnung der Daten und Pixelwerte über die "render_frame()"-Funktion hinaus noch parallel zu allen Folgefunktionen abläuft.

Hätte man nun nur einen Texturspeicher reserviert und würde damit beginnen, das nächste Bild zu laden, so müßte sich der Grafikkartentreiber mit der GPU synchronisieren. Im Klartext hieße dies: die Anwendung müßte warten bis alle Berechnungen, die auf diese Textureinheit zugreifen, abgeschlossen sind. Da trotz der GPU-Unterstützung vor

[3]Theoretisch natürlich 12,5. Der Parameter ist aber zur Zeit als Ganzzahl-Wert realisiert.

Abbildung 5.7.: Landschaftstextur

der Berechnung noch CPU-intensive Aufgaben anfallen (Szenenhierarchieauswertung, Ermittlung der Objektreihenfolge) ist dies nicht im Sinne einer effektiven Szenendarstellung. Legt man nun einen zweiten Texturspeicher an, so kann auf diesen unabhängig von allen Berechnungen innerhalb der GPU zugegriffen werden und die Ladezeit wird so klein wie möglich gehalten.

Nachdem nun die Anwendung alle benötigten Texturen in die Grafikkarte geladen hat werden sie innerhalb des Durchlaufs für jedes verwendete Material mit der entsprechenden Textureinheit verknüpft und können aus dem Shader heraus über eine Variable vom Typ *sampler2D* angesprochen werden.

5.7. Verwendete Videotexturen

Innerhalb der Szene wurde für die Beleuchtung eine Landschaftstextursequenz mit einer Auflösung von 512x512 verwendet, welche aus 450 Bildern bestand. Die Wiedergabegeschwindigkeit war dabei auf 25 Bilder pro Sekunde festgelegt. Die Aufnahmen waren bereits als DV-Material vorhanden und gehörten zu den Daten, die vom PCM-Projekt ("People Cargo Mover", ein neuartiges Hochschienenbahnkonzept, erläutert in [PCM04]) vorlagen. Genauere Vorgaben konnten für den Inhalt daher nicht gegeben werden. Um eine in sich geschlossene Sequenz zu erzeugen und den Beleuchtungseffekt zu demonstrieren wurde mit After Effects eine Tunneldurchfahrt hinzugefügt, welche mit einem durch das Bild wandernden Streifen den Effekt einer im Tunnel angebrachten Lampe ergeben sollte. Abbildung 5.7 zeigt eine Aufnahme aus der Landschaftssequenz.

Um die zwei Passagiere im Inneren des Zuges darzustellen, wurde eine knapp 31 Sekunden (773 Bilder) lange Aufnahme im virtuellen Studio der Fachhochschule durchgeführt. Anschließend wurde der blaue Hintergrund in dieser Aufnahme mit einer Compositing Software ausmaskiert und mit Hilfe des Alphakanals als transparent markiert. Um hier

Abbildung 5.8.: Ein Bild der Passagierszene

ebenfalls eine in sich geschlossene Szene zu erhalten wurde der Anschluß mit Hilfe eines Morphing-Programms - nicht perfekt, aber akzeptabel - hergestellt. Zum Vergleich ist in Abbildung 5.8 links ein Bild der Originalsequenz zu sehen und rechts das ausgestanzte Bild, eingesetzt in einen neuen Hintergrund.

Die Landschaftsaufnahme wurde in der Szene auf eine große leinwandähnliche Fläche projiziert, die aus einem Zylinder ausgeschnitten war, um eine möglichst großflächige Überdeckung der Fensterfläche zu erreichen.

5.8. Beleuchtung

Die Aufgabe der Shader war es nun, aus der Textur (hier speziell der Landschaftstextur) die entsprechende Beleuchtungs-Information zu entnehmen und auf die Pixel anzuwenden. Dabei wurden zunächst die Vorgaben des Konzeptes noch genauer spezifiziert.

1. Die Beleuchtung sollte in Echtzeit erfolgen können
2. Sie sollte auf alle Oberflächen, also auch auf die Passagier-Videotextur wirken können.
3. Die Einwirkung sollte winkelabhängig sein, so daß der realistische Eindruck einer Beleuchtung von außen durch die Fenster entstünde.

Diese drei Punkte gaben den Rahmen für die Entwicklung der Shader vor.

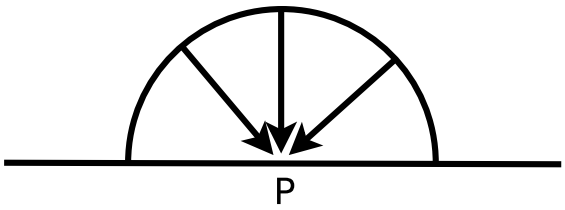

Abbildung 5.9.: Beleuchtungserfassung über Hemisphäre

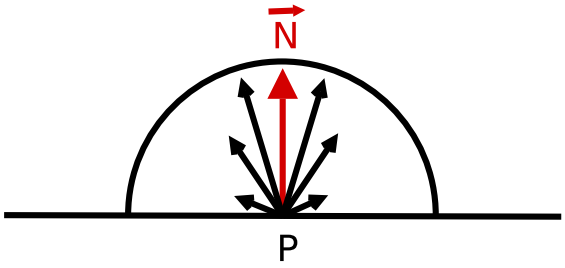

Abbildung 5.10.: diffuser Anteil für eine Normale

5.9. Theoretischer Ansatz für die Shader

In der Designphase ging es darum, den genauen Aufbau der Shader zu entwerfen. Die Eingabevariablen mußten festgelegt und der verwendete Algorithmus skizziert werden. Bei der Entwicklung des Verfahrens wurden existierende bildbasierte Beleuchtungsverfahren auf ihre Verwendbarkeit betrachtet. Diese Verfahren sollen in den wesentlichen Teilen kurz im Zusammenhang mit dem vorliegenden speziellen Problemfall betrachtet werden.

Betrachten wir zunächst einen Punkt P, welcher sich auf einer Oberfläche befindet. Man kann sich um den Punkt herum eine Halbkugel denken, über die alle einwirkenden Leuchtdichten integriert werden wie bereits in Kapitel 2 definiert. Abbildung 5.9 zeigt diese Hemisphäre um den Punkt P herum.

Die drei eingezeichneten Lichtvektoren sollen hier nur als Beispiel dienen, eine vollständige Erfassung würde mathematisch das Integral über die gesamte Halbkugel erfordern wie es auch die schon angesprochene Gleichung 2.6 vorsieht. Für die betrachteten Echtzeitverfahren kann die Halbkugel nicht unendlich klein unterteilt werden, da sie mit Hilfe einer Umgebungstextur ermittelt wird, deren kleinste Einheit die Größe der einzelnen Texel darstellt. Jedes Texel kann dann als einzelne Lichtquelle betrachtet werden. Bei einer entsprechenden Auflösung der Umgebungstextur nimmt die Summenbildung dieser einzelnen Elemente eine große Anzahl von Rechenschritten ein, daher ist es naheliegend, die Summe vorauszuberechnen. Betrachtet man nur den diffusen Anteil, so ändert sich für einen kleiner werdende Elevationswinkel über P nur der Faktor des $cos\phi$-Anteils. Dies ist schematisch in 5.10 für einen Punkt P und seiner Normalen N dargestellt. Dabei wurde der Lichtvektor gleich N angenommen.

Die Länge der Vektoren gibt den Faktor an, mit dem ein in dieser Richtung liegender

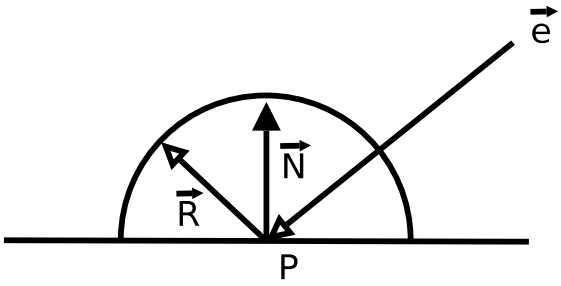

Abbildung 5.11.: verwendete Vektoren für die Berechnung des Spiegelanteils

Texel der Umgebungstextur gewichtet wird. Dies wird wie erwähnt im Voraus durchgeführt, indem man den Punkt P in der Originaltextur durch die gewichtete Summe der auf ihn einwirkenden Anteile der gesamten Texel auf der Kugeloberfläche ersetzt. Es entsteht eine neue diffuse Umgebungstextur. Führt man dies für alle Normalen im dreidimensionalen Raum durch, so erhält man für jede Richtung einen Wert, der für die Berechnung des diffusen Oberflächenleuchtdichteanteils verwendet werden kann, da dieser Anteil unabhängig vom Betrachter ist. Da eine Textur im Normalfall nur Werte von 0.0 bis 1.0 speichert, muß die Summe der Texelgewichtung dabei noch normalisiert werden, damit die Werte in diesem Bereich liegen.

Für den Spiegelanteil ließe sich dieses Verfahren in gleicher Weise durchführen, allerdings muß hierbei der Betrachterstandort mit einbezogen werden. Abbildung 5.11 zeigt die Zusammenhänge. Für jeden Betrachterstandort läßt sich ein dazugehöriger Betrachtervektor $\vec{e}$ finden, der zusammen mit der Normalen $\vec{N}$ den Reflektionsvektor $\vec{R}$ ergibt. Man kann nun ähnlich wie bei dem diffusen Anteil eine Gewichtung des Texels, auf das $\vec{R}$ zeigt, mit seinen Nachbartexeln vornehmen. Die Form der Gewichtung kann in diesem Fall in Anlehnung an das Phong-Modell zum Beispiel durch $(\vec{R} \cdot \vec{N})^s$ beschrieben werden. $\vec{R}$ und $\vec{N}$ müssen dafür normalisiert sein und s ist der Spiegelexponent, der über eine berechnete Textur konstant gehalten wird. Dies kann ebenfalls für alle Richtungen durchgeführt werden.

Das Endergebnis des Farbwertes der Pixeloberfläche wäre dann die Summe dieser beiden Anteile. Leider läßt sich der gespiegelte Anteil nicht befriedigend für alle Situationen auf diese Weise beschreiben. Die Berechnung bezieht nämlich nicht den Horizont der Oberfläche in die Betrachtung ein. Stellt man sich den Beobachter e mit einem sehr kleinen Elevationswinkel vor, so liegt ein Teil der zu gewichtenden Texel bereits unter dem Horizont von P und würde somit nicht in die Berechnung einfließen. Die Problematik ist in Abbildung 5.12 dargestellt.

Neben dem eigentlichen Reflektionsvektor $\vec{R}$ sind dort noch zwei kleinere Vektoren eingezeichnet, die wie bei der Abbildung des diffusen Anteils stellvertetend sind für die Gewichtung der Texel in diese Richtung. Man sieht hierbei deutlich, daß der Anteil des unteren Vektors in diesem Fall nicht berücksichtigt werden darf. Die Textur ist aber normalenunabhängig angelegt und bezieht sich nur auf den Reflektionsvektor $\vec{R}$. Die zusätzliche Einbeziehung des Normalenvektors würde eine höherdimensionale Textur verlangen, was den benötigten Speicher um das Vielfache erhöhen würde und zudem nicht direkt von

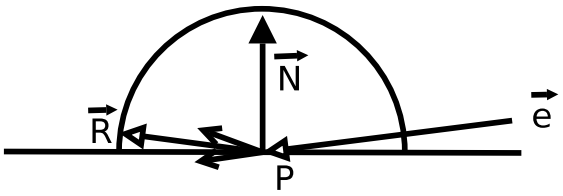

Abbildung 5.12.: Problem bei der Berechnung des Spiegelanteils

Abbildung 5.13.: Mipmaps Landschaftstextur

der Hardware unterstützt wird (maximale Anzahl der Textureinheiten liegt momentan üblicherweise bei 4 und die höchste Texturdimension liegt bei 3).

Selbst wenn man für unsere Anwendung diesen Fehler mit in das Ergebnis einkalkulieren würde, wäre für den Zugriff auf die Spiegeltextur eine weitere Textureinheit nötig und somit auch ein weiterer Texturzugriff. Um den Aufwand möglichst gering zu halten wurde daher schließlich nur der rein diffuse Anteil betrachtet. Zudem wurde eine Vereinfachung des Filters vorgenommen. Bei der Betrachtung der Landschaftstextur fiel auf, daß sie keine prägnanten Zonen mit großer Helligkeit besitzt, sondern relativ gleichmäßig verläuft.

Die Idee war daher, statt einer *cos*-Gewichtung einen simplen linearen Filter zu verwenden. Dies war aus dem Gedanken heraus entstanden, daß die Hardware selber bereits eine automatische Filterung für die Mipmap-Stufen anbot und wir somit die Fähigkeiten der Hardware ausnutzen könnten. Wie sich allerdings später bei Messungen (Tabelle 5.2) zeigte, ist die automatische Generierung durch die Hardware keine günstige Kombination in Verbindung mit komprimierten Texturen, so daß am Ende die Texturdaten doch extern vorgefiltert wurden.

Zur Veranschaulichung dieser Vorfilterung sind alle Mipmap-Stufen für ein Bild der Landschaftstextur in Abbildung 5.13 dargestellt. Links ist die Original Textur zu erkennen und jede rechts anschließende Textur ist gefiltert und in ihrer Auflösung in beide Dimensionen halbiert worden.

Da der Zugriff auf diese Texturen aus dem Fragmentshader heraus über normalisierte

Texturkoordinaten erfolgt, kann auch auf eine 4x4 Textur noch in beliebig kleinen Abständen zugegriffen werden. Die Hardware bietet für diese texelunabhängigen Zugriffe eine Anzahl von Filteralgorithmen an, neben dem Punktfilter existieren auch bilineare, trilineare und anisotropische Filter. Für unseren Zweck wurde ein bilinearer Filter verwendet, da wir nur die Information einer Mipmap-Stufe verwenden wollten. Der Zugriff auf einen Punkt dieser Mipmap lieferte also einen gewichteten Wert für eine große Fläche der Originaltextur.

Wenn diese Originaltextur zum Beispiel aus 512x512 Texel besteht, so ist bei der 4x4 Mipmap ein Texel der Mipmap stellvertetend für 128 Texel der Originaltextur und würde somit in Relation zu dem beschriebenen Ansatz von Zeichnung 5.10 die Länge der Gewichtungsvektoren bis zu einem gewissen Winkel in gleicher Größe bestehen lassen falls ein Box Filter verwendet würde. Die genaue Implementierung der Filter, die von der Hardware bei der automatischen Mipmap-Generierung verwendet wird ist allerdings herstellerabhängig und damit nicht näher spezifiziert. Auch aus diesem Grund erscheint die externe Generierung der Mipmap-Stufen sinnvoller, da so die Filterform definiert vorgegeben werden kann und auch die beschriebene *cos*-Gewichtung möglich wäre. Wolfgang Heidrich beschreibt ein OpenGL-basiertes Verfahren in [Hei00], die Implementierung dieses Ansatzes hätte aber die Erstellung einer weiteren Komponente für die Anwendung verlangt und konnte aus Zeitgründen leider nicht verwirklicht werden.

Nachdem der Ansatz für das Gewichtungsverfahren der Textur damit definiert ist fehlen noch 2 wichtige Punkte. Einmal müssen die Texturkoordinaten ermittelt werden und zum anderen muß die Generierung des Umgebungslichtanteils für den Innenraum festgelegt werden.

Zunächst folgt die Beschreibung für die Texturkoordinaten. Bei einer Umgebungstextur, die eine 360^o-Ansicht erlaubt, ist die Generierung der Texturkoordinaten direkt aus Elevation und Azimuth des Normalen- bzw. Beobachtervektors möglich. In den meisten Fällen wird die Abbildung als 6 einzelne Würfeltexturen realisiert, die anhand der Winkelvorzeichen und -bereiche ausgewählt werden. Dies ist sogar direkt über die Hardware möglich. Bei unserer Anwendung haben wir nur einen begrenzten Ausschnitt, den man mit einer Würfeltexturansicht vergleichen kann.

Abbildung 5.14 zeigt eine grobes Schema der Zuggrundfläche. Im Inneren des Zuges befinden sich hauptsächlich die Sitzreihen als komplexe Gegenstände. Wand und Boden stellen hingegen annähernd Ebenen dar.

Als Beispiel ist ein Betrachtervektor $\vec{e}$ eingezeichnet. Da wir nur den diffusen Anteil betrachten, ist die Oberflächenleuchtdichte des Punktes P nur abhängig von der Normalen an dieser Position. Würden wir die Landschaftstextur als eine Seite eines Würfels betrachten, so hätten wir die in Abbildung 5.15 erkennbare Problematik. Ein Punkt P wird bei der Würfelumgebungstextur als im Ursprung eines (hier nur zweidimensional gezeichneten) Würfels angenommen. Von den 6 Flächen des Würfels sind hier 4 sichtbar und als +x, -x und +z und -z angegeben. In unserem Fall hätten wir nur eine davon ausgefüllt, nämlich +z (eine freie Entscheidung, die sich mit der im 3D-Bereich üblichen Vereinbarung des Koordinatensystems deckt, daß die y-Koordinate nach “oben” zeigt.).

Vektor $\vec{a}$ zeigt wie gewünscht auf einen Texel der Landschaftstextur, Vektor $\vec{b}$ hingegen würde auf die Textur +x zeigen, welche in unserem Modell leer wäre. Daher eignet sich diese Art der Texturkoordinatengenerierung nicht für unseren Zweck.

Besser geeignet ist eine Abbildung auf eine Halbkugel, dargestellt in Abbildung 5.16.

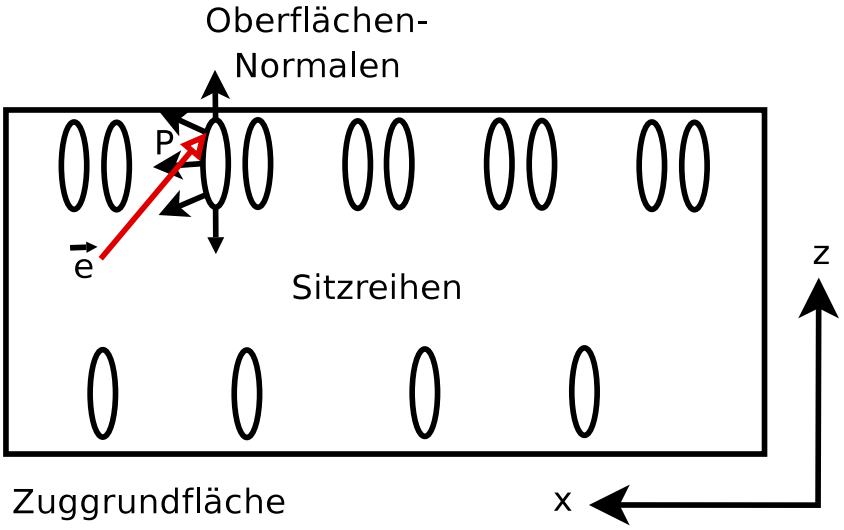

Abbildung 5.14.: Schema der Anordnung von Zug und Textur

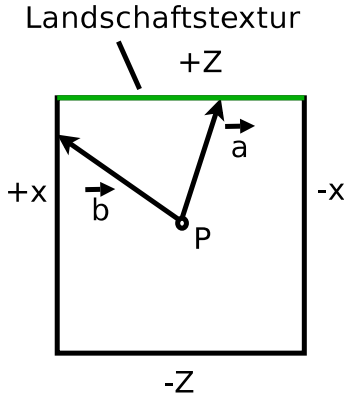

Abbildung 5.15.: Probleme bei Verwendung von Würfeltexturkoordinaten

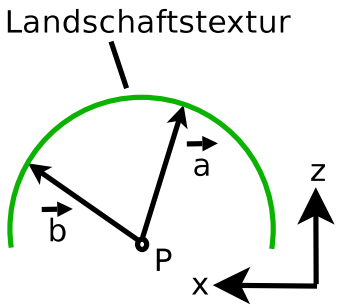

Abbildung 5.16.: Bessere Lösung mit Abbildung auf eine Halbkugel

Die Vektoren treffen nun auf einen Texel der Textur, dies ist für alle Winkel innerhalb von 180^o in positiver Z-Richtung möglich.

Da die Abbildung der Landschaft von ihrer Perspektive her keinen Öffnungswinkel von 180^o erfaßt, ergibt diese Abbildung einen inkorrekten Winkel. Das Problem wird aber durch die Tatsache gemildert, daß die Landschaftstextur ein relativ gleichmäßiges Muster darstellt. Die Aufteilung in Himmel und Wiese/Felder bleibt über die gesamte Dauer großflächig erhalten, so daß die Verzerrung durch die Abbildung akzeptabel ist und durch die Filterung in ihrer Abweichung noch stärker reduziert wird.

Bei der genaueren Prüfung des Halbkugelmodells fiel noch ein weiterer Effekt auf, der analysiert werden mußte. Die in Abbildung 5.16 gezeigte Halbkugel gilt in dieser Form nur für Punkte, die im Bereich der Fensterscheibenebene des Zuges liegen. Die Problematik wird in Abbildung 5.17 dargestellt. Wie bereits angedeutet, wird bei den Umgebungstexturverfahren der Punkt an der Oberfläche üblicherweise in der Mitte der Kugel/des Würfels gedacht, da auf diese Weise eine einheitliche Berechnung erfolgen kann. Für die Abbildung bedeutet dies, daß die auf der Umgebungstextur abgebildete Textur sich quasi in unendlicher Entfernung vom Punkt befindet, es spielt also keine Rolle, wo sich der Punkt innerhalb der Weltkoordinaten befindet. Dies führt in anderen Bereichen (Innenräume) zu Problemen, in unserem Fall ist es akzeptabel.

Inakzeptabel scheint hingegen, daß der Normalenvektor, auf dessen Basis der Texel in der Textur ermittelt wird, auf die Zugwand im Inneren des Zuges zeigt. Somit würde er von dem im Zuginneren herschenden Umgebungslichtanteil beleuchtet und nicht direkt von der Textur. Wir müssen dies also berücksichtigen, indem wir den Winkel der Normalen für Punkte mit abnehmender z-Koordinate einschränken

An dieser Stelle erschien eine kurze Analyse des verwendeten Maya-Zugmodells nützlich, um die beim tatsächlichen Modell auftretenden Winkel zu ermitteln. Da der Grundriß des Zuges nicht rechteckförmig ist, wurden die äußersten Fenstergrenzen ermittelt und 3 Winkel eingezeichnet. α (80^o) und β (86^o) sind die extremsten Stellen, an denen sich ein Objekt befindet, ω (155^o) ist hingegen mittig und stellt den größtmöglichen Öffnungswinkel dar. Die Winkel sind zusammen mit einem Drahtgittermodell des Zuges (von oben betrachtet) in Abbildung 5.18 dargestellt.

Um diesen Öffnungswinkel auf möglichst einfache Weise nachzubilden, wurden verschiedene Möglichkeiten mit unterschiedlichem Ansatz und Komplexitätsgrad überlegt.

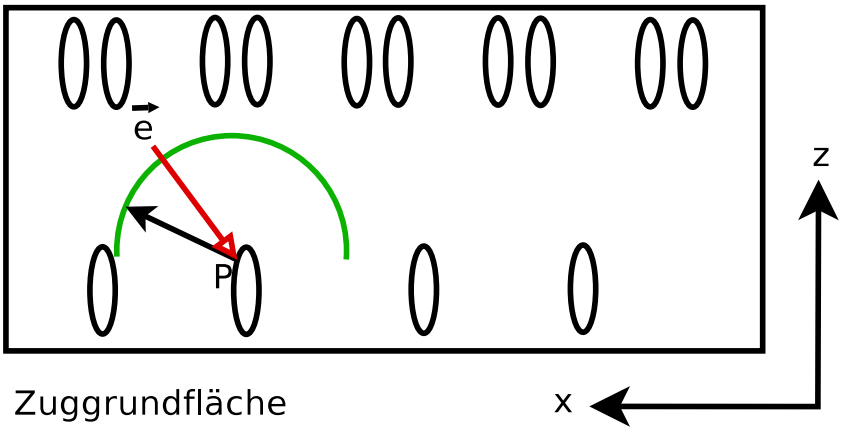

Abbildung 5.17.: Problem bei Punkten im Zuginneren

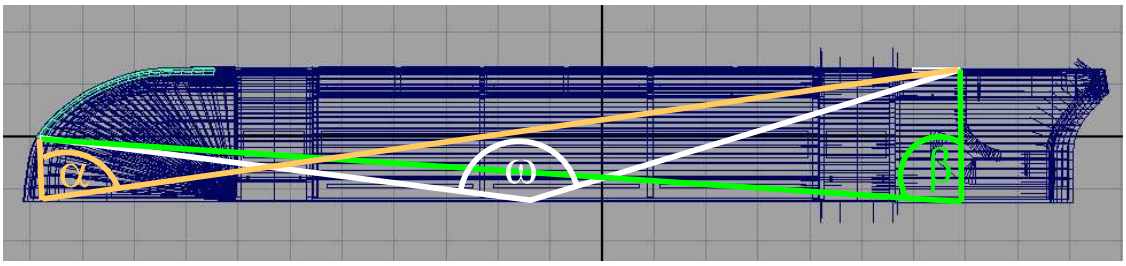

Abbildung 5.18.: Verdeckungswinkel im Zug

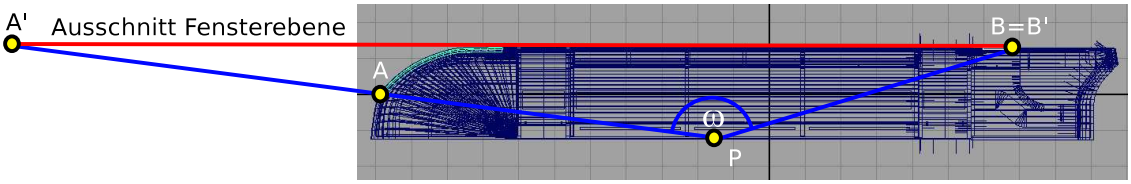

Abbildung 5.19.: Sichtbarer Bereich für Punkt P

1. Berücksichtigung der realen Geometrie des Zuges. Der in Richtung der Normalen abgeschickte Strahl wird auf einen Schnittpunkt mit der Zuggeometrie überprüft.

2. Projektion des Öffnungsspaltes auf die xy-Ebene und Ermittlung der Normalen, die diese Ebene schneiden.

3. Ermittlung eines kreisförmigen Ausschnitts zwischen der Begrenzung durch den Fensterbereich.

Die erste Variante kann nicht innerhalb des Shaders durchgeführt werden, da sie Informationen über die gesamte Geometrie voraussetzt. Dieser Ansatz wurde daher verworfen.

Die zweite Variante ist hingegen im Shader realisierbar, die Normale ist zusammen mit dem Eckpunkt als Strahl zu formulieren, der Schnittpunkt t mit der Ebene durch Einsetzen in eine Gleichung ermittelbar.

Abbildung 5.19 zeigt die Winkel und verwendeten Punkte. Der von Punkt P aus sichtbare Bereich in der horizontalen Richtung wird durch den Öffnungswinkel ω beschrieben[4]. A und B stellen die äußersten Enden der Zugfenster dar. Die Fensterebene ist nun die Projektion über A und B auf eine gedachte xy-Ebene. A' und B' auf dieser Ebene stellen die horizontale Begrenzung des auf diesen Punkt direkt einfallenden Außenlichtes dar. Bild 5.20 zeigt dies für die frontale Ansicht. h ist hierbei die Höhe der Fenster, $\vec{q}$ und $\vec{r}$ stehen stellvertretend für zwei verschiedene Normalen. Q und R sind die dazugehörigen Punkte auf der xy-Ebene. P ist die Weltposition des Eckpunktes. Aus dieser Angabe und der Normalen läßt sich ein Strahl formulieren, hier für den Vektor q:

$$f(t) = P + \vec{q}t \tag{5.1}$$

Die allgemeine Ebenengleichung lautet:

$$(X - S) \cdot \vec{N} = 0 \tag{5.2}$$

$\vec{N}$ ist der Normalenvektor der Ebene, S ist ein frei wählbarer Punkt auf der Ebene und X der zu untersuchende Punkt. Nach Einsetzen von f(t) als Punkt X, auflösen nach t und Einsetzen als Parameter in die Strahlgleichung ergibt sich für den resultierenden Punkt:

$$f(\vec{q}) = P + \frac{(S - P) \cdot \vec{N}}{\vec{q} \cdot \vec{N}} \vec{q} \tag{5.3}$$

[4]bei all diesen Ansätzen wird die Unterteilung der Fenster nicht berücksichtigt

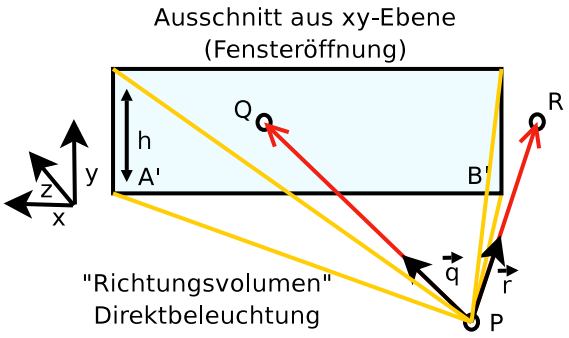

Abbildung 5.20.: Sichtbarer Bereich, perspektivische Ansicht

In unserem Fall ist $\vec{N}$ durch die xy-Ebene gegeben und bleibt konstant, S kann ebenfalls konstant gewählt werden. Mit dieser Formel können A' und Q berechnet werden (B' ist in unserem Fall gleich B). Für alle Punkte, deren Z-Koordinate gleich oder größer der von A ist, gilt dabei, daß sie in Richtung von A' nicht eingeschränkt sind (für die Textur ist nur der Bereich bis 180^o relevant). Nachdem auf diese Weise der Schnittpunkt der Normale mit der Ebene berechnet wurde, muß noch getestet werden, ob der Schnittpunkt Q sich innerhalb der von A', B' und h definierten Grenzen befindet. Ein Problem bei der Berechnung des Punktes ist allerdings, daß auch ein Schnittpunkt gefunden wird falls die Normale in die entgegengesetzte Richtung zeigt. Eine schnelle Erkennung ist aber über das Vorzeichen von t möglich. Für den beschriebenen Fall wäre es negativ.

$$t = \frac{(S - P) \cdot \vec{N}}{\vec{q} \cdot \vec{N}} \tag{5.4}$$

Um noch eine weitere Alternative zum Vergleich zu bekommen wurde das dritte Verfahren ebenfalls realisiert, welches von Ansatz her wesentlich einfacher aufgebaut ist. Bei diesem Verfahren ermittelt man zu jedem Punkt die beiden Vektoren, die auf den linken und rechten Begrenzungspunkt zeigen. Zwischen diesen Punkten wird nun ein Kreis aufgespannt, dessen Ursprung in Höhe der Fenstermitte liegt. Die Benutzung eines Kreises hat den Vorteil, daß für die Information, ob die Normale innerhalb des gewünschten Raumwinkels liegt, nur das Vektorprodukt $\vec{p} \cdot \vec{q}$ nötig ist. Das Vektorprodukt $\vec{h} \cdot \vec{q}$ kann dann für die Ermittlung eines Grenzwertes benutzt.werden. Ist das erste Produkt kleiner als der Grenzwert, so zeigt die Normale außerhalb des Kreises. Der Nachteil ist hierbei allerdings, daß der Öffnungswinkel in horizontaler und vertikaler Richtung gleich ist. Abbildung 5.21 zeigt diesen Ansatz mit der schematisch angedeuteten tatsächlichen rechteckigen Fensteröffnung.

Man erkennt deutlich, daß der Öffnungswinkel in vertikaler Richtung sehr viel größer als die eigentliche Fensteröffnung ist, so daß hierbei die Größe der Öffnung nicht korrekt bestimmt wird, da das Verhältnis zwischen vertikalem und horizontalem Winkel prinzipbedingt bei 1:1 liegt. Eine Analyse der Maya-Szene zeigte, daß das Verhältnis zwischen horizontaler und vertikaler Öffnung in Wirklichkeit etwa bei 14:1 liegt.

Abschließend muß noch erörtert werden, welche Helligkeitsinformation nun für den Fall verwendet werden soll falls die Oberflächennormale außerhalb des ermittelten Öffnungs-

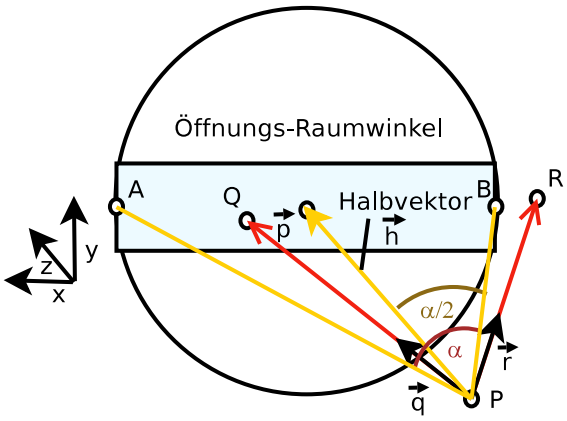

Abbildung 5.21.: Alternatives Verfahren mit Halbvektor

bereiches zeigt. Um es noch einmal zu wiederholen: für das Zugmodell bedeutet dieser Fall, daß dem betrachteten Punkt bzw. der betrachteten Normale eine Innenwand des Zuges gegenüberliegt. Eine Wand ist zwar keine Lichtquelle, in einer realen Szene strahlt sie dennoch genauso wie die Außenlandschaft eine gewisse Lichtmenge ab. Dieses Verhalten kann mit den lokalen Beleuchtungsmodellen, wie sie für die vorliegende Arbeit zur Anwendung kamen, nicht befriedigend realistisch nachgebildet werden.

In der Realität wird Licht von einer Quelle abgestrahlt und ein Teil davon trifft auf eine Oberfläche im Raum. Dann reflektiert diese wiederum das Licht in andere Richtungen (beschrieben durch die BRDF). An dieser Stelle hören die bisher behandelten Modelle auf, da der Empfänger dieser Lichtstrahlen der Beobachter ist. Realistischere Modelle verfolgen alle abgestrahlten Leuchtdichten nicht nur für den Beobachter, sondern für die gesamte Szene. Da eine Verfolgung aller Interaktionen für die gesamten abgestrahlten Lichtteilchen mit gängigen Methoden nicht zu handhaben ist, kann eine Annäherung nur mit Hilfe stochastischer Modelle erreicht werden. Diese sind aber nicht in Echtzeit durchzuführen. Wie aus den theoretischen Grundlagekapiteln hervorgeht, ist die Hardware außerdem nicht auf diese Art der Berechnung ausgelegt, da immer nur ein Eckpunkt bzw. Pixel betrachtet wird.

Um dennoch einen realistischen Eindruck von Helligkeitsschwankungen zu erhalten, wurde für die vorliegende Anwendung für diesen Fall ein gemeinsamer Wert angenommen, der die Leuchtdichte der gesamten Textur auswertet (kleinste Detailstufe). Diese Lösung ist allerdings nicht optimal, da sie die unterschiedliche Reflektionseigenschaften von Wänden nicht berücksichtigt.

Als letzter Punkt muß noch die Grundlage für die Passagiershader erwähnt werden, Da die Passagiertextur auf ein Billboard[5] gerendert wurde und somit keine Geometrieinformation vorliegt, wurde hier ein gemeinsamer Helligkeitswert genommen, der - wie im Fall der indirekten Beleuchtung - aus der kleinsten Mipmapstufe entnommen wurde. Dieser Wert, multipliziert mit der Oberflächentextur (in diesem Fall die Videotextur der Passagiere) ergibt den endgültigen Helligkeitswert. Die Shadertexturfunktion liefert einen

[5]Ein Billboard ist eine Ebene, deren Normale ständig in Richtung des Beobachters ausgerichtet ist.

vierdimensionalen Vektor (R, G, B und Alpha) zurück. Die eigentliche Transparenz wird über das Materialskript von Ogre erreicht, indem das Ergebnis des Passagiermaterialdurchlaufs mit der Hintergrundebene über den Alphawert kombiniert wird. Die Formel dafür lautet:

$$C_{Pixel} = \alpha * C_{Passagiershader} + (1 - \alpha) * C_{Bildspeicher} \tag{5.5}$$

C steht hierbei für die Farbe, α für den Alphawert.

5.10. Realisierung der Shader

Da das Grunddesign des verwendeten Algorithmus feststand, wurden nun die Shader geschrieben. Es sollen an dieser Stelle kurz die wesentlichen Programmzeilen zur Realisierung der Algorithmen dokumentiert werden.

Für alle Objekte im Zuginnenraum wurde als Basis der gleiche Shader verwendet, die individuelle Oberflächengestaltung geschah dann durch eine (statische) Textur. In der ersten Textureinheit wurde die Landschaft als Videotextur geladen, in die zweite Textureinheit dann die zu dem jeweiligen Objekt gehörende Oberflächentextur.

Eine wichtige Entscheidung bei der Umsetzung eines Shaders ist die Frage, ob aufwendige Berechnungen im Vertex- oder im Fragmentprozessor durchgeführt werden sollen. Im Normalfall ist die Anzahl von Eckpunkten wesentlich geringer als die Anzahl der sich daraus ergebenden Bildpunkte, so daß es naheliegend wäre, einen komplexen Algorithmus im Vertexshader umzusetzen und dann dem Fragmentprozessor interpolierte Werte zu übergeben. Das Beispiel des Phong/Gouraud-Shading (siehe Kapitel 2.5f) macht aber bereits deutlich, daß eine Interpolation nicht in jedem Fall ein optimales Ergebnis liefert.

Bei unserem Shader fallen laut Design die folgenden Aufgaben an:

1. Überprüfung mit Hilfe der Oberflächennormale, ob der Eckpunkt direkt von der Landschaftstextur beleuchtet wird

2. Ermittlung eines Gewichtungsfaktors, der die Helligkeit des Pixels bei größer werdendem Winkel reduziert (Vektorprodukt zwischen Z-Achsenvektor und Oberflächennormale)

3. Falls direkte Beleuchtung vorliegt: Ermittlung der genauen UV-Koordinate für das Nachschlagen in die Landschaftstextur, ansonsten kleinste Detailstufe auswählen und gemeinsamen Wert als Grundlage für das Umgebungslicht nehmen.

4. Ermittlung der Oberflächenkoordinaten für die Oberflächentextur (zweite Textureinheit)

Punkt 4 ist trivial, die UV-Koordinaten wurden dem Eckpunkt durch die Modellierungssoftware bereits zugewiesen und müssen nur interpoliert werden. Dafür werden sie einfach in eine (vordefinierte) *varying*-Variable geschrieben.

Punkte 1-3 lassen sich hingegen theoretisch sowohl pro Fragment (mit der interpolierten Normale) oder pro Eckpunkt (mit der Originalnormale) durchführen. Hier sind also beide

Varianten denkbar, daher wurden bei der vorliegenden Anwendung auch beide Möglichkeiten umgesetzt. Die Auswertung der erzielten Geschwindigkeitsunterschiede findet sich im Kapitel 6.2.

Wir beschreiben im folgenden die Realisierung der ersten Shadervariante (Ebenengleichung). Der Kernalgorithmus für die Berechnung des Fensterausschnitts ist dabei für beide Komplexitätsvarianten (pro Fragment/pro Eckpunkt) gleich, nur der Ort des Codes ist unterschiedlich. Die benötigten eckpunktspezifischen Variablen müssen bei der Fragmentvariante vorher vom Vertexprozessor in interpolierende (*varying*-)Variablen geschrieben werden, dies erfolgt durch eine Deklaration innerhalb beider Programme und einer einfachen Zuweisung.

Das Programm beginnt wie erwähnt mit der main()-Funktion des Vertex-Programms. Als erstes transformieren wir den Eckpunkt für die Rasterisierung:

*gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;*

Wir benötigen diesen Wert nicht für unsere Berechnungen, er muß aber vom Vertex-Programm geschrieben werden. Anschließend übergeben wir die Original-UV-Koordinaten des Objektes (um Punkt 4 abzudecken):

gl_TexCoord[0]=gl_MultiTexCoord0;

Die Tatsache, daß wir hierbei zwei eingebaute Variablen mit scheinbar fast gleichem Namen benutzen liegt darin begründet, daß gl_MultiTexCoord0 eine von OpenGL vorgegebene *attribute*-Speichervariable für die UV-Koordinate ist und es sich bei gl_TexCoord[0] um eine *varying*-Variable handelt. Da wir den Wert interpolieren wollen, ist dieser Schritt nötig.

Bei GLSL existiert eine vordefinierte gl_Normal-Variable, in die der von der Anwendung geschriebene Normalenwert abgelegt wird. Da wir die Objekte in der Szene weder skaliert noch rotiert haben ist es nicht notwendig, diesen Wert mit der ebenfalls vorhandenen gl_NormalMatrix (eigentlich die transponierte ModelViewMatrix) zu multiplizieren.

Wir führen alle Berechnungen in Weltkoordinaten durch, damit wir die Vektoren der xyz-Koordinatenachsen direkt verwenden können. Da OpenGL aber nur die bereits multiplizierte gl_ModelViewMatrix anbietet, welche eine Transformation in den Raum des Beobachters (engl. *eye space*) durchführen würde, können wir hierfür nicht auf die OpenGL-Funktionalität zurückgreifen. Ogre erlaubt es aber, diese und noch einige andere Matrizen direkt über das Materialskript anzufordern. Der entsprechende Eintrag heißt "param_named_auto worldmat world_matrix". Hier wird in die Shader-*uniform*-Variable worldmat automatisch die "Welt"-Matrix geschrieben[6].

Mit Hilfe dieser Welt-Matrix transformieren wir den Eckpunkt:

*vec4 world_vertex4 = worldmat * gl_Vertex;*
world_vertex = world_vertex4.xyz;

[6] diese Matrix wird auch als "Model-Matrix" bezeichnet und transformiert vom lokalen Objektraum in den für alle Objekte gemeinsamen Welt-Raum

Der letzte Schritt reduziert vom homogenen Raum in den dreidimensionalen. Dies ist hier durch einfaches Weglassen der 4. Koordinate erfolgt, da wir wissen, daß die Objekt-Eckpunkte allesamt dreidimensional sind, so daß die 4. Koordinate immer 1.0 ist. Eine korrekte Behandlung würde die Division durch die w-Koordinate erfordern. Es handelt sich wie die Annahme über die Normale um eine Optimierung.

Als nächsten Schritt berechnen wir den Vektor vom Eckpunkt auf den Begrenzungspunkt A (siehe Zeichnung 5.19).

```
vec3 A_vector = A - world_vertex;
```

Man erkennt, wie einfach eine Vektorberechnung innerhalb des Shaders durchgeführt werden kann. Wir verwenden diese Variable, um den Schnittpunkt mit der xy-Ebene zu finden und somit den vom Eckpunkt in diese Richtung sichtbaren Bereich zu erfassen.

```
vec3 v = vec3(0.0,0.0,1.0);
vec3 P = vec3(0.0,0.0,B.z);
float t = dot((P-A),v);
t = t/(dot((A_vector),v));
A_plane = A_plane + t * (A_vector);
```

Dies ist die direkte Umsetzung der Formeln aus Kapitel 5.9. v ist die Normale der xy-Ebene, P ist ein Punkt in dieser Ebene (wir nehmen den Punkt B als in der Ebene liegend an, siehe Abbildung 5.19). A_plane enthält nun den Schnittpunkt mit der xy-Ebene. Als nächstes berechnen wir analog dazu den Schnittpunkt der Oberflächennormalen mit der Ebene:

```
t = dot((P-world_vertex),v);
t = t/(dot(gl_Normal,v));
if (t>=0.0)
    intersection = world_vertex + t * gl_Normal;
else
    not_in_plane = true;
```

Die Berechnung von t erfolgt auf gleiche Weise wie zuvor, allerdings werten wir an dieser Stelle das Vorzeichen von t aus. Ein Wert kleiner Null bedeutet, daß die Normale in die negative Richtung verlängert werden muß und besagt somit wie bereits in Kapitel 5.9 erwähnt, daß sie von der Ebene wegzeigt. Für diesen Fall können wir die bool'sche not_in_plane-Variable auf "wahr" setzen.

An dieser Stelle muß allerdings darauf hingewiesen werden, daß wir eine Bedingung noch vernachlässigt haben. Da sich Objekte auch (in Z-Richtung) "vor" dem Punkt A befinden, ist es für diese Eckpunkte nicht möglich, einen Schnittpunkt mit der Ebene zu finden. Dies ist allerdings auch nicht nötig, da wir bereits festgelegt haben, daß eine direkte Beleuchtung grundsätzlich nur dann angenommen wird, wenn die Normale zur positiven Z-Achse einen maximalen Winkel von 90^o einnimmt (also 180^o insgesamt). Wir wissen also für diesen Fall, daß wir in Richtung der positiven X-Achse keine Begrenzung durch die Fenster zu berücksichtigen brauchen. Da wir uns in Weltkoordinaten bewegen, kann diese Bedingung relativ einfach über den Vergleich mit der Z-Koordinate ermittelt werden.

```
if (world_vertex.z >= A.z)
    left_unbounded = true;
```

Um es noch einmal zu verdeutlichen, diese "unbegrenzt"-Bedingung ist nur relevant für Oberflächennormalen, die in Richtung der positiven X-Achse zeigen, für die andere Richtung kann trotzdem eine Begrenzung durch den Punkt B erfolgen.

Wir haben nun alle Informationen für die Aussage, ob der Punkt direkt von der Landschaftstextur beleuchtet wird und können den Vergleich durchführen.

```
float h = 0.65;
direct_light = true;
if ((not_in_plane == true) || (intersection.y >= (B.y+h)) ||
        (intersection.y < (B.y-h)) || (intersection.x < B.x)
        || ((left_unbounded==false) && (intersection.x >= A_plane.x))){
    direct_light = false;
}
```

h gibt hierbei die halbe Höhe des Fensters an (Punkt B liegt genau in der senkrechten Mitte der Fensters). direct_light wird zunächst als "wahr" vorinitialisiert. not_in_plane wurde bereits erläutert. In der folgenden "If"-Bedingung wird überprüft, ob die Normale grundsätzlich von der Ebene wegzeigt, ob der Schnittpunkt oberhalb oder unterhalb des Fensters oder ob er außerhalb des Punktes B (in negativer X-Richtung) liegt. Ist dies alles nicht erfüllt, so wird als letztes noch geprüft, ob Punkt A gültig ist und wenn ja, ob der Schnittpunkt hier eventuell außerhalb der Grenzen liegt. Falls alle diese Bedingungen nicht zutreffen, so ist unsere Annahme der direkten Beleuchtung erfüllt. Im anderen Fall wird direct_light auf "falsch" gesetzt.

An dieser Stelle wissen wir nun, ob der Punkt direkt oder indirekt beleuchtet wird. Für den Fragmentprozessor wird nun die entsprechende Detailstufe festgelegt.

```
if (direct_light == true)
    lod = 9.0;
else
    lod = 1000.0;
```

Es ist an dieser Stelle nur noch Punkt 2 offen. Dieser wird über ein einfaches Vektorprodukt im Fragment-Programm ermittelt. Die Addition von 1.0 und die anschließende Division wird durchgeführt, damit die Gewichtung nicht nur über 90^o, sondern über 180^o erfolgt, da ansonsten eine zu starke Abdunklung erfolgt.

```
float degfac = dot (normalized_normal,vec3(0.0,0.0,1.0));
degfac = (degfac+1.0)/2.0;
```

Für diese Berechnung wird die normalized_normal-Variable aus den interpolierten Normalen ermittelt. Da das Vektorprodukt zwischen zwei normalisierten Vektoren dem Kosinus des eingeschlossenen Winkels entspricht kann dieser Wert nicht mehr im Vertex-Programm ermittelt und interpoliert werden. Dieser Teil des Quelltextes befindet sich also bereits im Fragment-Programm.

Wir berechnen nun den Farbwert des Fragments:

```
vec2 normaltex = -vec2(normalized_normal.x,normalized_nnormal.y);
normaltex = (normaltex+1.0)/2.0;
vec4 col = texture2D(lighttexture,normaltex,lod);
col = vec4(vec3(length(col.xyz)),1.0);
col = vec4(col.rgb*degfac,1.0);
vec4 undertex = texture2D(surfacetexture,gl_TexCoord[0].xy);
gl_FragColor = undertex*col;
```

Zerlegen wir diesen Block in die einzelnen Abschnitte. Zunächst werden aus dem interpolierten Normalenvektor des Fragments die Texturkoordinaten ermittelt. Das Vorzeichen wird umgedreht, da UV-Koordinaten und XY-Koordinaten genau gegeneinander stehen. In Richtung der positiven X/Y-Achse liegt die negative U/V-Achse. Da UV-Koordinaten von 0.0 bis 1.0 reichen muß der Wert noch skaliert und versetzt werden. Der interpolierte lod-Wert wird nun zusammen mit der ermittelten Koordinate verwendet, um den entsprechenden Wert aus der Lichttextur zu erhalten. Dieser Wert wird noch mit degfac gewichtet und zum Schluß mit der Oberflächentextur multipliziert.

6. Ergebnis

6.1. Anwendung

Mit einer entsprechenden Textur als Basis für die Oberfläche versehen präsentiert sich die Anwendung bei Verwendung des Ebenengleichungs-Shaders wie in Abbildung 6.1, bei Verwendung des Halbvektor-Shaders wie in Abbildung 6.2.

Man erkennt deutlich, daß der Boden bei Verwendung des Halbvektors noch von der direkten Beleuchtung beeinflußt wird, während dies bei der Ebenengleichungsvariante auf korrekte Weise durch den indirekten Beleuchtungsanteil geschieht. An den Stühlen erkennt man auch die Richtungswirkung: das direkt von links kommende Licht führt zu einer hellen Kante, während die rechte Seite der Lehne dunkel ist. Da dies auf gleiche Weise in beiden Shadern berechnet wird ist hier auch kein Unterschied zu erkennen.

Als Gegensatz ist in 6.3 eine Aufnahme des Zuges im Tunnel zu erkennen. Die Landschaftstextur wurde dabei auf einen gleichmäßigen Grauwert gesetzt, damit im Zuginneren noch etwas zu erkennen ist. Die hellen Streifen sind hierbei ein experimenteller Shader, der auf die Helligkeit der Beleuchtung reagiert und sich - wie ein Zuglicht - automatisch einschaltet, sobald ein Schwellwert unterschritten wird. Allerdings übt dieser Effekt keinen Einfluß auf den Innenraum aus, dieser wurde allein durch die Landschaftstextur beleuchtet.

Abbildung 6.4 zeigt den Bodenbereich, auch hier ist zu erkennen, daß die Flächen auf der linken Seite nur indirekt beleuchtet werden während Teile der Stühle auf der rechten Seite direkt beleuchtet sind und somit eine größere Helligkeit besitzen. Um diesen Übergang deutlich sichtbar zu machen wurde für Abbildung 6.5 der Shader so erweitert, daß die direkt beleuchteten Flächen rot und die nur indirekt beleuchteten Flächen blau eingefärbt wurden. Die Klappstühle befinden sich am Kopfende des Zuges dem Fenster genau gegenüber, so daß die rechteckige Begrenzungsebene hier sehr gut zu erkennen ist. An den Sitzbezügen erkennt man auch, daß die Wirkung von der Normale der Oberfläche abhängig ist, da die Kissenränder, die gegen die Wände zeigen, nicht direkt beleuchtet und damit blau eingefärbt sind. Abbildung 6.6 zeigt dies noch deutlicher anhand einer Kugel in der Raummitte. Man erkennt den nach rechts und in vertikaler Richtung begrenzten Ausschnitt. Das kleine Bild zeigt die gleiche Kugel unter Verwendung der eckpunktlastigen Variante des Shaders. Man erkennt die Ungenauigkeit durch die Schnittpunktberechnung auf Eckpunktbasis.

6.2. Analyse der Shader

Um einen Überblick über die tatsächliche Leistung der beiden Shadervarianten zu bekommen wurde ein Geschwindigkeitsvergleich durchgeführt. Variante 1 (V1) berechnet die Fensteröffnung mit Hilfe der Ebenengleichung, Variante 2 (V2) berechnet die Öffnung

Abbildung 6.1.: Innenansicht Variante 1 (Ebenengleichung)

Abbildung 6.2.: Innenansicht Variante 2 (Halbvektor)

Abbildung 6.3.: Zug im Tunnel

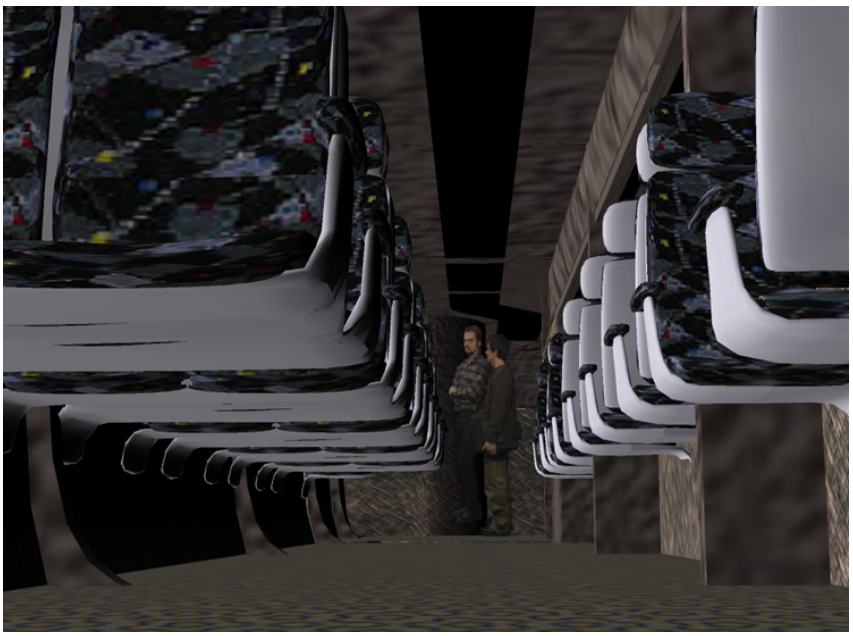

Abbildung 6.4.: Ansicht Bodenbereich

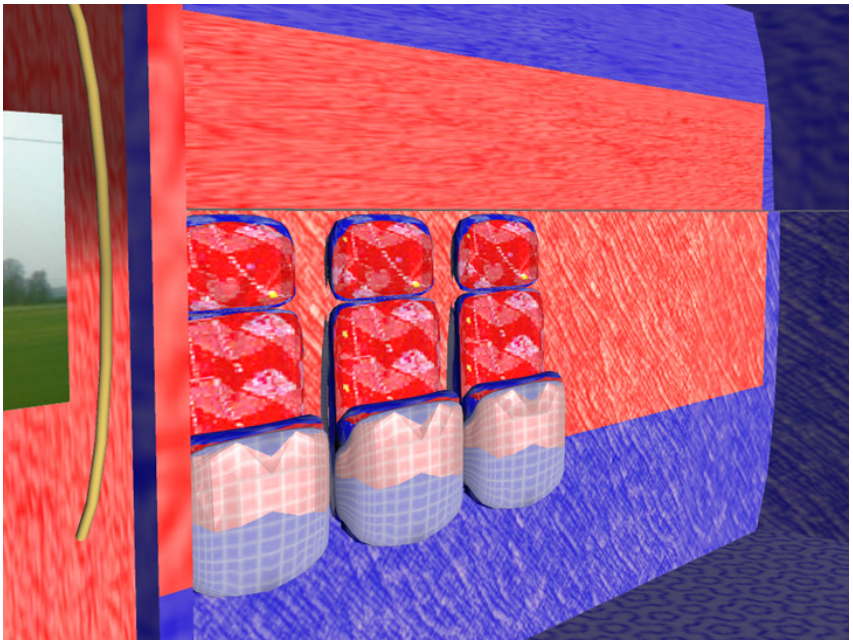

Abbildung 6.5.: Visualisierung des (in)direkt beleuchteten Anteils

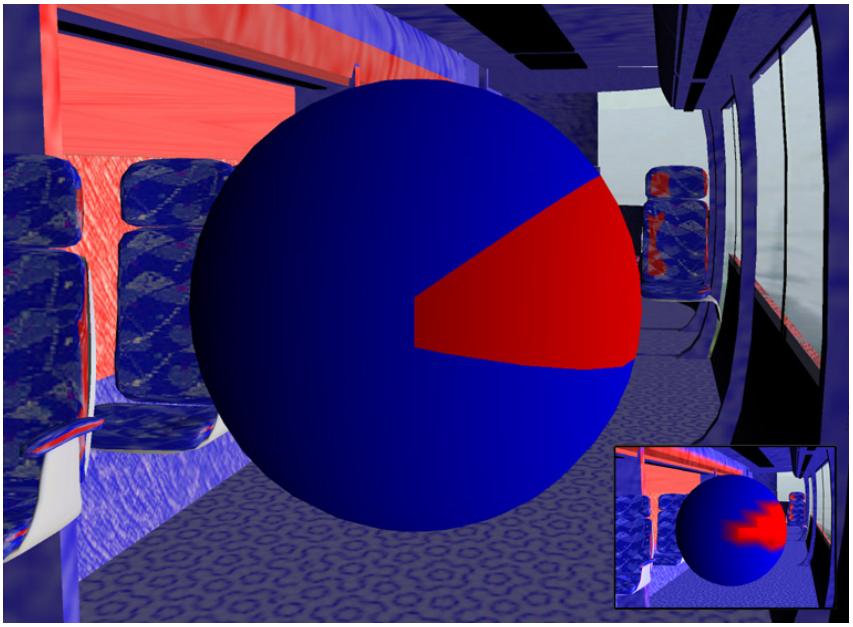

Abbildung 6.6.: (In)direkt beleuchteter Anteil abgebildet auf eine Kugel (der kleine Ausschnitt zeigt die 'Vert'-Variante)

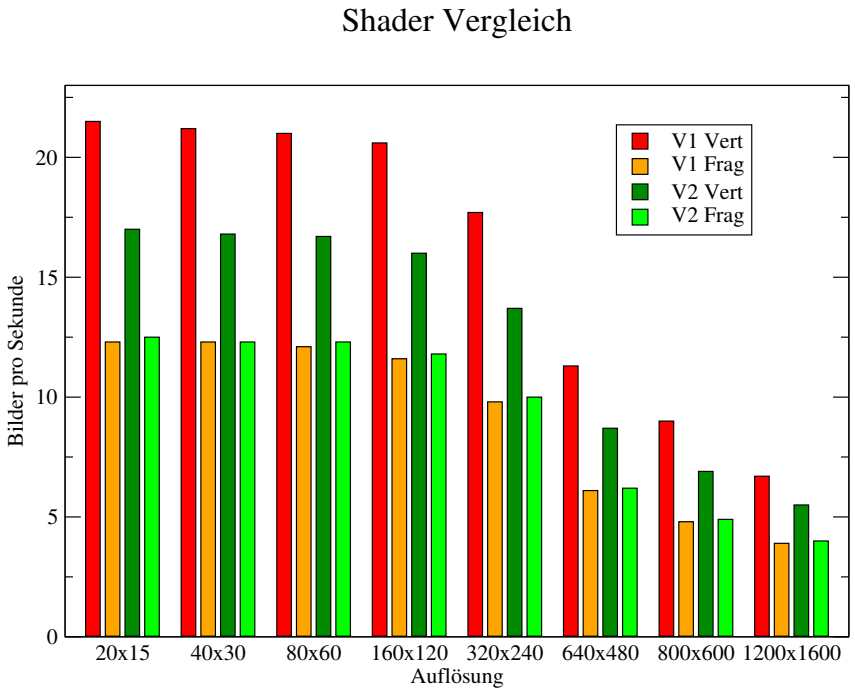

Abbildung 6.7.: Shadergeschwindigkeit bei unterschiedlichen Auflösungen

mit Hilfe des Halbvektors. “Frag” bedeutet dabei, daß die Berechnungen vollständig für jedes Fragment durchgeführt wurden, “Vert” bedeutet, daß - soweit möglich - interpolierte Werte verwendet wurden. Für alle Tests wurde das Anti-Aliasing abgeschaltet und die Daten wurden unter Linux auf einer Athlon XP 1700+-CPU mit 512 MB erfaßt, die verwendete Grafikkarte war eine NVidia GeForce FX 5600 mit 325 MHz GPU- und 550 MHz Speichertakt. Die Anzahl der darzustellenden Eckpunkte in der Szene betrug laut Ogre etwa 110.000 und die Texturen waren dabei vollständig im Speicher gecached, so daß kein Nachladen von der Festplatte die Zeit pro Bild verfälschen konnte. Das Diagramm 6.7 zeigt das Ergebnis der Messung.

Deutlich erkennbar ist der Einbruch der Bilderanzahl für beide Varianten falls der Hauptanteil der Berechnung im Fragmentshader erfolgt. Wertet man die Verhältnisse untereinander für alle Auflösungen aus, so stellt man fest, daß die proportionale Verteilung innerhalb einer Auflösung dabei in etwa gleich bleibt. Dies ist auch zu erwarten, da die Anzahl der darzustellenden Pixel keine Auswirkung auf die jeweilige Laufzeit für einen Eckpunkt bzw. Fragment haben sollte.

In Zahlen bewirkt die pro-Fragment-Berechnung bei Variante 1 einen Einbruch der Bildwiederholrate auf 53%-58%, bei Variante 2 auf 71%-74% der vorherigen Leistung, wobei für diesen Fall beide Varianten in etwa gleichauf liegen.

Die Halbvektor-Methode erweist sich in der Messung also als unnütze Alternative, da sie nicht nur eine ungenauere Erfassung der Öffnung mit sich bringt sondern für die pro-Eckpunkt-Berechnung auch noch langsamer ist.

Betrachtet man für alle Varianten das Verhältnis zwischen Pixelanzahl und Bildwie-

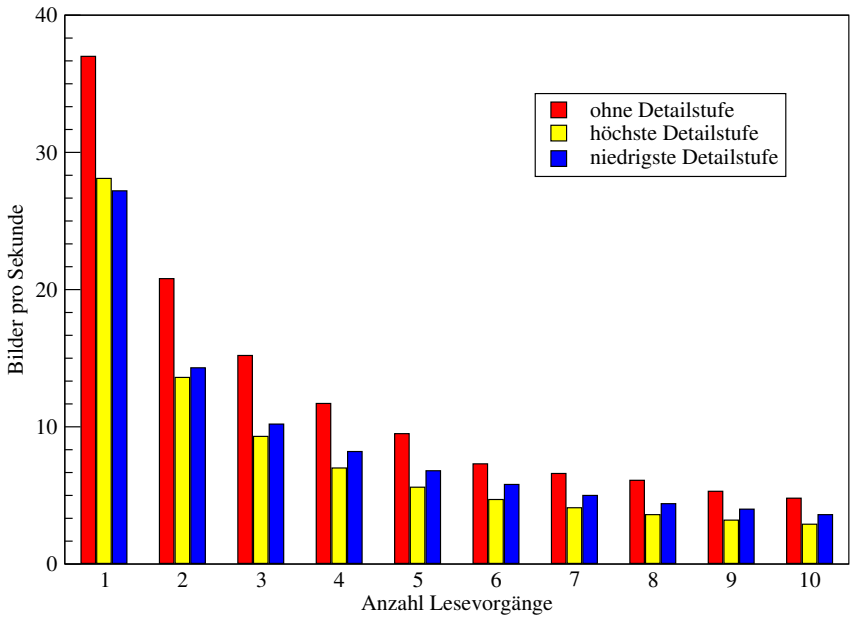

Abbildung 6.8.: Auswirkungen der Texturzugriffe auf die Anzahl der Bilder

derholrate, so ist bei den ersten 4 Auflösungen der Unterschied marginal, so daß hier der Hauptanteil der Zeit für das Einrichten und Schreiben der Szenendaten und die Eckpunkt-Verarbeitung anfällt während das Schreiben der Pixel selber nur geringen Anteil hat. Erst ab 320x240 ist eine deutlichere Abhängigkeit von der Auflösung zu erkennen. Dies ist ein Beispiel für die Schwierigkeit bei der genauen Vorhersage der zu erwartenden Shaderdurchlaufzeiten, da sie sich aus verschiedenen Faktoren ergeben, deren genaue Darstellung den Rahmen des Kapitels sprengen würde. Für eine detailierte Darstellung sei daher an dieser Stelle auf ein NVidia-Paper [NVi04] verwiesen.

Als zweite Analyse wurde die Auswirkung des Texturzugriffs auf die Bildwiederholrate durchgeführt. In der vorliegenden Arbeit werden nur zwei Texturzugriffe verwendet, einmal die Lichttextur und zum zweiten die Oberflächentextur. Für eine aufwendigere BRDF wären aber weitere Zugriffe denkbar (Kombination mehrerer Texturen). Weitere Beispiele sind Normalentexturen und Schattentexturen.

Das Diagramm in Abbildung 6.8 zeigt die Auswertung eines Testshaders. Dieser Testshader führte im Vertex-Programm nur die Eckpunkt-Transformation durch und im Fragment-Programm erfolgten dann nur die Zugriffe auf die Textur. Für dieses Beispiel wurde auf eine Textureinheit zugegriffen und die Zugriffsstelle (Texturkoordinate) wurde in kleinen Schritten erhöht. Dies war nötig, damit der optimierende GLSL-Compiler die Zugriffe nicht zusammenfaßte. Erfaßt wurden zunächst die Werte für das Auslesen der höchsten (volle Auflösung) und niedrigsten (1x1 Mipmap) Detailstufe.

Da in GLSL die Detailstufe für das Auslesen der Textur innerhalb des Fragmentshaders nur in Form eines Versatzwertes angegeben werden kann und der Grenzwert dafür bei ± 1000.0 liegt wurden diese beiden Werte verwendet. Es stellte sich allerdings heraus, daß

der Compiler bei einem Wert von -1000.0 die Berechnung der Detailsstufe wegoptimierte, so daß noch eine dritte Messung mit einem größeren Wert (-150.0) verwendet wurde.

Dies zeigt eine Schwierigkeit im Umgang mit GLSL. Falls eigentlich ein Zugriff auf eine bestimmte Mipmap-Stufe gewünscht ist, so wird dennoch die aktuelle Detailstufe der Szene berechnet, obwohl dieser Wert eigentlich nicht benötigt wird. In einem Assembler-Programm wäre dieses Vorhaben ohne Probleme umzusetzen, in der Hochsprache hingegen existiert keine passende Funktion. Der Vergleich zeigt, daß die Ermittelung der Detailstufe die Bildwiederholrate um den Faktor 0.6-0.7 ausbremst. Bei kritischen Anwendungen wird sich dies bemerkbar machen.

Auch bei unserer Anwendung wäre der Zugriff auf die Lichttextur theoretisch ohne Detailstufe denkbar. Für den Vertexshader existiert eine solche Funktion auch, sie ist aber laut GLSL-Standard nur dort erlaubt und führt zu einer Fehlermeldung falls sie aus dem Fragment-Programm heraus aufgerufen wird. Dem Autor ist nicht ganz ersichtlich welche Gründe dazu geführt haben und ob eine zukünftige Version des Sprachstandards diese Möglichkeit auch im Fragmentshader bieten wird.

Man erkennt im Diagramm, daß der Zugriff auf die niedrigste Detailstufe die Bildwiederholrate etwas weniger reduziert als der Zugriff auf die höchste Stufe. Dies dürfte im Texturcache begründet sein (die niedrigste Mipmap besteht nur aus einem Pixel).

Es fällt ebenfalls auf, daß die Einbuße pro zusätzlichem Texturzugriff nicht konstant ist, sondern bei Faktor 0.5 anfängt und bis 0.9 reicht. Ein eher unrealistischer Gegentest mit reduzierter Geometrie zeigte, daß die Reduktion in diesem Fall sogar anfangs um den Faktor 0.4 erfolgt.

Für das Shaderdesign bedeutet es, daß ein Zugriff auf die Textureinheit eine große Geschwindigkeitseinbuße darstellt und die Anzahl daher möglichst klein gehalten werden sollte. Wenn allerdings bereits ein oder zwei Zugriffe erfolgt sind, fällt die Auswirkung von weiteren Texturzugriffen nicht mehr so stark ins Gewicht.

Zusammenfassend muß natürlich darauf hingewiesen werden, daß die ermittelten Werte nur relevant sind für die NV3x-Generation. Bei der momentan vorherrschenden steten Weiterentwicklung ist damit zu rechnen, daß häufig verwendete Funktionen optimiert werden, dazu gehört sicherlich auch der Zugriff auf die Texturen. Wie in Anhang C zu ersehen ist, wurde bei der Nachfolgegeneration der GPU für Variante 1 die Anzahl der nötigen Vertex-Programminstruktionen bereits reduziert. Dies dürfte an zusätzlichen Opcodes (Assemblerbefehle) liegen. Die NV4x-Generation bietet auch eine Vervierfachung der Fragment-Pipelineanzahl und eine höhere Taktrate. Man kann also davon ausgehen, daß sich die Meßergebnisse nicht unbedingt auf die Nachfolgegeneration übertragen lassen werden.

6.3. Résumé

Die Anwendung wurde den Vorgaben entsprechend realisiert und erreichte auf den zur Verfügung stehenden Testsystemen eine Bildwiederholrate von 6-12 Bildern pro Sekunde. Für eine sinnvolle Echtzeitanwendung sollte dieser Wert allerdings eher bei 30-40 Bildern pro Sekunde liegen, dies wird sich aber erst mit der Nachfolgegeneration der für diese Arbeit verwendeten GPU realisieren lassen.

Die aus Platzgründen nicht im Detail beschriebene Anwendung selber war dank der

vielfältigen Funktionen von Ogre in wenigen Zeilen zu realisieren, der Hauptanteil fiel auf das Laden der Objektdaten und die Kamerasteuerung. Dank eingebauter Keyframes konnte auch eine automatische Kamerafahrt ohne großen Aufwand realisiert werden.

Ein ursprünglicher Ansatz, der nicht bis zum Ende verfolgt wurde war die Realisierung von Schatten. Dies erfolgte aus praktischen Gründen, da eine Analyse der nötigen Schritte zur Erzeugung von Schatten unter Berücksichtigung der Beleuchtung durch die Videotexturen den Umfang der Diplomarbeit gesprengt hätte. Der Autor hält die Schattendarstellung dennoch für sehr wichtig und dies sollte daher einer der ersten Ansätze für eine Erweiterung der vorliegenden Arbeit sein.

Das ursprüngliche Ziel war die Erhöhung des Realitätseindrucks beim Beobachter durch die Videotexturen und die Beleuchtung. Bei den Videotexturen ist dies als gelungen zu bezeichnen, da die Auflösung der dargestellten Texturen für einen realistischen Eindruck ausreicht. Dank des flexiblen Videotextur-Plugins ist eine problemlose Einbindung weiterer Texturen für andere Zwecke möglich. Eine Sache, die dem Plugin allerdings zum jetzigen Zeitpunkt fehlt, ist die Möglichkeit einer Synchronisierung mehrerer Filme. Es wäre zum Beispiel denkbar, 2 verschiedene Ansichten der Landschaftstextur zu verwenden, um die Beleuchtung auch für einen Zug mit zwei Fensterseiten durchzuführen. Dafür müßte aber der Inhalt beider Seiten aufeinander abgestimmt, also synchronisiert werden. Auch eine Interaktion mit dem Benutzer - im Zug durch die Videoterminals angedeutet - ist denkbar, dies würde die Erweiterung um einen über die Anwendung heraus steuerbaren Film erfordern.

Im Falle der Shader ist die Implementierung allerdings noch nicht vollkommen zufriedenstellend. Ein Problem ist die bereits angesprochene niedrige Geschwindigkeit, die durch die aufwendige pro-Fragment-Berechnung bedingt ist. Da die Nachfolgegeneration der GPUs aber Texturzugriffe auch aus dem Vertex-Programm heraus erlauben, wäre es möglich, hier wieder eine Auslagerung zu betreiben.

Ein weiteres Problem ist die nicht vollkommen gelöste Frage nach der indirekten Beleuchtung. In unserer Anwendung wird in diesem Fall einfach ein über die gesamte Textur gemittelter Wert verwendet. Dies entspricht aber nicht der Realität und ergab auch kein so befriedigendes Ergebnis wie ursprünglich vom Autor erhofft. Ein Ansatz für eine Weiterentwicklung wäre möglicherweise die Vorberechnung der Lichtverteilung mit Hilfe eines Radiosity-Algorithmus. Dies ließe sich zwar nicht in Echtzeit durchführen, da die Szene aber statisch ist, kann dies im Voraus durchgeführt werden.

Die vorliegende Anwendung ergibt für die direkt beleuchteten Flächen ein gutes Ergebnis, auch die ändernden Lichtverhältnisse werden glaubhaft simuliert, allerdings erscheint die Gewichtung durch einfache Verwendung der Mipmap-Stufen nicht ausreichend und führt an ungünstigen Flächenübergängen (steile Winkel) zu Sprüngen, die sich nicht mit einer realistischen Erfahrung decken. Für eine Weiterentwicklung wäre hierbei wohl eine bessere Filtervariante wie in [Hei00] beschrieben vorzuziehen.

Als generelle Erfahrung läßt sich festhalten, daß der Großteil der Entwicklungszeit gleichermaßen auf das Debugging der Shader und des Videotextur-Threads entfiel. Bei den Threads lag dies an der anfangs unbekannten Tatsache, daß die Ogre-Bibliothek nicht threadsicher war. Die dadurch auftretenden sporadischen Fehler waren schwer einzugren-

zen. Später hielt durch einen falsch konzipierten Mutex-Block noch ein Deadlock[1] auf, der ebenfalls schwer zu debuggen war, da er nur sporadisch in Erscheinung trat.

Bei den Shadern hingegen hängt der große Debugaufwand zur Zeit noch an der fehlenden Möglichkeit, Shadervariablen und -register zur Laufzeit zu analysieren. Ein Shader kann bislang nur ausgeführt werden, teilweise interessieren aber Zustände von Variablen. Diese lassen sich nur indirekt über die Ausgabe dieser Variable in das Farbregister durchführen. Dabei hat man allerdings das Problem, daß der Wertebereich eingeschränkt ist auf 0.0 bis 1.0. Man ist also gezwungen, den Wert entsprechend aufzubereiten, was zusätzlichen Programmcode kostet. Es ist allerdings zu erwarten, daß zukünftige Implementierungen verstärkt Debugmöglichkeiten anbieten werden.

Abschließend noch ein Wort zu den zukünftigen Verwendungsmöglichkeiten der in der vorliegenden Arbeit zusammengetragenen Informationen. Der Großteil davon betrifft die Grundlagen der Shader und Lichtberechnung. Der Autor hält Informationen in diesem Bereich für sehr wichtig, um die Möglichkeiten, die die Grafikhardware bietet ausnutzen zu können. Er hält es für sehr wahrscheinlich, daß zukünftige Anwendungen es dem Benutzer erleichtern werden, eigene Shader in Skriptform zu integrieren. Ogre kommt diesem Ziel bereits sehr nahe, wie auch die vorliegende Anwendung zeigt. Die Initialisierung der Bibliothek nimmt nur wenige Zeilen in Anspruch, der Großteil liegt in Skripten und Konfigurationsdateien, die problemlos mit einem Texteditor angepaßt werden können.

Die Schritte, die zur Verwendung eines Shaders nötig sind, beziehen also immer weniger die Kenntnis über Strukturen der unteren Ebene mit ein, so daß kein tiefgehendes Wissen über Grafik-APIs wie OpenGL oder DirectX nötig ist. Ogre belegt dies eindrucksvoll, indem es beide APIs in eine gemeinsame API integriert, dabei aber immer noch die Möglichkeiten der Hardware ausreizt.

Die vorliegende Arbeit könnte somit als Einstieg für ein Projekt benutzt werden, welches die Erhöhung des Realitätsgrades einer computergenerierten Szene zum Ziel hat und dessen Teilnehmer einen Überblick über die gängigen Verfahren gewinnen möchten. Bei der vorliegenden Arbeit erfolgt dabei zwar die Einschränkung auf OpenGL, allerdings hält der Autor die Shadersprachen für sehr ähnlich, so daß Grundlagen anhand von GLSL auch einem Cg-Einsteiger nutzen werden.

[1] durch einen Programmierfehler warten zwei Threads auf eine Bedingung, die nie erfüllt wird weil dazu einer von beiden weiterlaufen müßte

7. Ausblick

7.1. Echtzeitbeleuchtung

Die vorliegende Anwendung ist ein erster Schritt in Richtung der Verbessung des Realismus innerhalb einer computergenerierten Szene. Aufbauend auf den Erfahrungen durch die Anfertigung der Anwendung bzw. vorliegenden Arbeit und durch die während der Diplomarbeitsphase natürlich fortgeschrittenen Forschungen auf dem Gebiet lassen sich bereits logische Erweiterungen und Überarbeitungen ankündigen.

Ein wichtiger Teil, der in der vorliegenden Arbeit durch das beschränkte Zeitfenster nicht ausführlich angegangen werden konnte, sind Schatten. Die Meinung vieler, die - allerdings nicht unter Testbedingungen - einen Blick auf die Anwendung warfen, lief auf eine noch zu künstliche Atmosphäre hinaus, obwohl der Beleuchtungseffekt positiv auffiel. Speziell die Beleuchtung der Passagiere wurde interessiert hinterfragt.

Eine realistische Empfindung geht normalerweise einher mit der mehr oder weniger unbewußten Wahrnehmung von Schatten. Besonders prägnant ist das Fehlen bei den Passagieren zu beobachten. Diese scheinen über dem Boden zu schweben. Es seien kurz die wesentlichen Gründe dargelegt, warum die Schattenberechnung trotz der scheinbaren Nähe zur Beleuchtung nicht Bestandteil der vorliegenden Arbeit ist. In der Computergrafik gibt es momentan zwei Grundvarianten der Schattenberechung. Im einen Fall verlängert man die Geometrie der Objekte aus Sicht der Lichtquelle künstlich in die Tiefe (bei passender Unterstützung bis in die Unendlichkeit) und berechnet dann aus Sicht des Beobachters in einem Schattendurchlauf das sogenannte Schattenvolumen (engl. *shadow volume*). Diese Berechnung bedeutet einen erhöhten Rechenaufwand für die Hardware. Die Information, ob sich ein Objekt im Schatten befindet, wird durch den Stencil-Puffer bestimmt. Die genaue Erklärung des Verfahrens würde das Kapitel sprengen, Details finden sich in [AMH99]. Von Vorteil und gleichzeitig auch wieder problematisch sind die scharfen Schattenkanten. In der realen Welt kommen solche Schatten aber nur bei Punktlichtquellen vor und die meisten Lichter sind Flächenlichter, so daß es immer einen Halbschatten gibt.

Bei dem zweiten Verfahren wird die Szene aus Sicht der Lichtquelle berechnet und nur die Tiefenpufferinformationen (Z-Buffer) gespeichert. Es entsteht eine Schattentextur, die transformiert auf die Szene aus Sicht des Beobachters abgebildet wird. Leider ergeben sich durch die begrenzte Auflösung der Textur und perspektivische Verzerrung Aliasingstörungen. Es gibt mehrere Ansätze, diese Problematik zu beheben, eine allgemein zufriedenstellende Lösung wurde aber noch nicht gefunden. Eine kurze Übersicht inklusive einer neuen Variante der perspektivischen Schattenberechnung bietet [WSP04].

Die Ogre-API bietet zwar sowohl Stencil wie auch Schattentexturberechnung an, die Stencil-Variante erwies sich aber als unbrauchbar, da die Komplexität des Zugmodells die Frameraten fast zum Erliegen brachten. Die zweite Textur-Variante ist ohne perspektivische Korrektur, so daß in Tests Aliasstörungen sichtbar wurden.

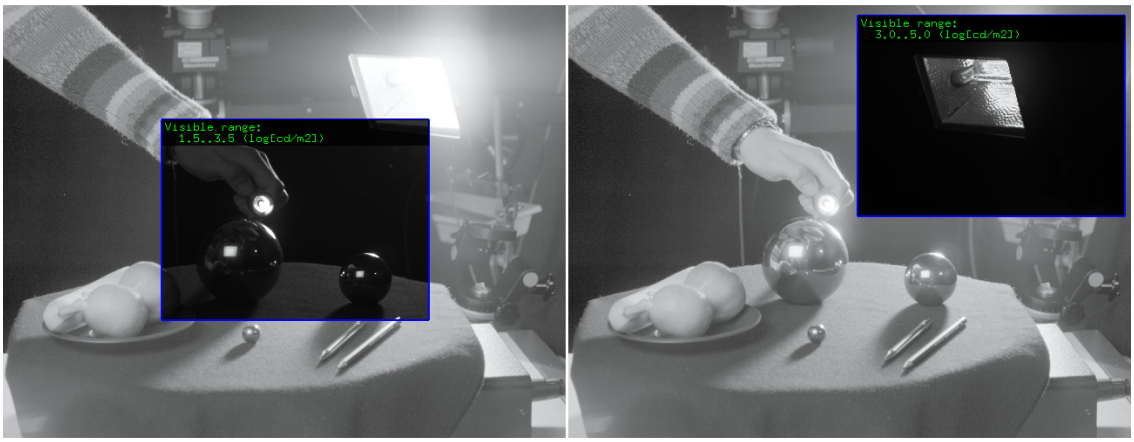

Abbildung 7.1.: Graustufen HDR-Aufzeichnung (Quelle: MPI Informatik [MKMS04])

Ein nächster Schritt wäre also die Erweiterung der Anwendung/Ogre-Bibliothek um einen perspektivischen Schattenalgorithmus mit entsprechender Vorfilterung.

Ein weiterer Ansatz zu Verbesserungen bei der Beleuchtung stellt die bislang sehr niedrige Auflösung der Landschaftstextur dar. Eine Vergrößerung der erfaßten Fläche der Landschaft (also eine echte sphärische Umgebungsabbildung) würde aber vermutlich keine allzugroßen Verbesserungen mit sich bringen. Das größere Potential steckt in High Dynamic Range-Aufnahmen. Momentan gibt es bereits CCD-Kamera-Prototypen, die in der Lage sind, HDR-Bilder direkt zu erfassen [HDR04]. In [MKMS04] wurde darauf aufbauend ein Verfahren vorgestellt, um so erfaßtes Material abgestimmt auf die Wahrnehmung des menschlichen Auges zu kodieren. Abbildung 7.1 zeigt ein Beispiel für den Informationsgehalt einer HDR-Aufnahme. Beide Seiten zeigen das gleiche Bild, der umrandete Bereich stellt unterschiedliche Ausschnitte aus dem gesamten Helligkeitsbereich dar. Obwohl der Scheinwerfer in der linken Aufnahme überbelichtet erscheint, sieht man in der rechten Aufnahme, daß dennoch alle Einzelheiten vorhanden sind.

Für unsere Anwendung würde es die bessere Gewichtung zwischen der Himmelsfläche und dem reflektierten Licht des Bodens erlauben. Da die CCDs einer üblichen DV-Kamera nur einen kleinen Dynamikbereich abdecken können, werden mit Hilfe von Blenden und Filtern die Helligkeitsunterschiede reduziert und der ursprüngliche Szenenkontrast geht verloren. Bei HDR ist diese Information noch im Bildmaterial enthalten und kann angepaßt ausgewertet werden.

Es ist für die Auswertung dieser Information auch nötig, eine Simulation der korrekten Helligkeitsverteilung im Innenraum vorzunehmen. Mit Hilfe der bereits erwähnten Kugelfunktionen lassen sich nicht nur Lichtinformationen, sondern auch Informationen über Selbstabschattung und Verdeckung durch andere Objekte erfassen. Ein Artikel, der sich mit diesem Thema befaßt ist [SKS02]. Für die Zugszene könnte die Vorberechnung mit einem entsprechendem Algorithmus eine sehr viel realistischere Lichtverteilung ermöglichen. Auf diese Weise könnte der in der vorliegenden Arbeit nicht näher differenzierte und von der Gesamttextur abgeleitete Umgebungslichtanteil realistischer dargestellt werden.

Ebenfalls möglich wäre die Erweiterung des bislang rein diffusen Anteils um einen spie-

gelnden Anteil. Bei einer entsprechenden Landschaftsaufnahme mit Sonnenschein könnte auf diese Weise auch versucht werden, die Einstrahlung der Sonne in den Zuginnenraum zu simulieren.

7.2. General Purpose GPU

Da ein wesentlicher Anteil der Szenenberechnung mit Hilfe der GPU vorgenommen wird, ist auch hier ein Ausblick auf die Zukunft sinnvoll. Eine ganze Reihe von Arbeiten beschäftigen sich mit der Umsetzung von existierenden Grafik-Algorithmen auf die Verhältnisse der GPU. Dennoch gibt es auch einen Forschungsbereich, der den eigentlichen Zweck - die Grafikberechnung - zunächst vernachlässigt und sich mit der Möglichkeit einer parallelen Berechnung von allgemeinen Aufgaben durch die GPU beschäftigt. Der dazugehörige Begriff der General Purpose-GPU (deutsch: Allzweck-GPU) findet sich in einer Anzahl von Publikationen wieder und ihm ist sogar eine eigene Website gewidmet [GPG04].

Die Basis dieser Ansätze ist es zum Beispiel, Texturen nicht als Grafikdaten, sondern als allgemeine Daten zu betrachten. Die Hersteller der Grafikchipsätze haben bereits vor einer Weile auf die gestiegenen Anforderungen reagiert und zunehmend die Möglichkeit von Gleitkomma-Grafikspeicher in die Hardware aufgenommen. Die bisher üblichen Grafiktexturen erforderten 24 oder 32-bittige Ganzzahl-Speicher. Auch wenn der Zugriff auf diese Daten intern als Gleitkommazahl gehandhabt wird (0.2, 2.0, ...), so wird im eigentlichen Texturspeicher dabei in Wirklichkeit eine Ganzzahl abgespeichert. Eine 1.0 innerhalb des Shaders bei einem 8 Bit tiefen Speicher entspricht so einem tatsächlichen Wert von 255, bei 16 Bit wären dies bereits 65535. Der Trend zu Aufnahmen mit hoher Dynamik erfordert allerdings die Speicherung als Gleitkommazahl (Exponent und Mantisse). Teilweise bieten die Grafikkarten sogar bereits die Möglichkeit an, Daten als 4*32 Bit Gleitkommazahlen abzuspeichern. Auch wenn die üblichen Monitore und Anzeigegeräte diesen Dynamikumfang nicht wiedergeben können, so besteht dennoch ein Bedarf an dieser hohen Auflösung. Aufwendige Effekte wie Tiefenschärfe oder Bewegungsunschärfe erfordern teilweise mehrere Durchläufe. Eine Akkumulation in einen 8 Bit Puffer führt zu Rundungsfehlern, da die Addition zweier Werte bereits zu einem Überlauf führen kann, so daß hier der im Puffer befindliche Wert zum Beispiel erst gelesen, addiert, halbiert und wieder zurückgeschrieben wird. In einen 128 Bit "Überpuffer" könnten hingegen die Werte einfach addiert werden und die Division würde erst am Ende erfolgen.

Wie bereits erwähnt können natürlich auch allgemeine Daten in solch eine Textur geladen und parallel von mehreren Vertex oder Fragment-Programmen abgearbeitet werden. Das Ergebnis des Fragmentprozessors muß in diesem Fall nicht unbedingt in einen Bildspeicher geschrieben werden, sondern kann ebenfalls als Textur mit großer Bitspeichertiefe ausgegeben und von einem Nachfolgedurchlauf erneut benutzt werden.

Es gibt sogar bereits eine Sprache ("Sh"), die zum alleinigen Zweck der Metaprogrammierung von GPUs entworfen wurde und daher nicht die Priorität wie Cg oder GLSL auf grafische Belange legt, sondern deren Aufgabe die effiziente Einbindung der GPU innerhalb einer Applikation als massiver Parallelrechner ist. Für nähere Informationen sei auf das Buch [MM04] und die dazugehörige Webseite [SHM04] verwiesen.

Ein Anwendungsbeispiel für diese Möglichkeiten, welches sogar einen graphischen Cha-

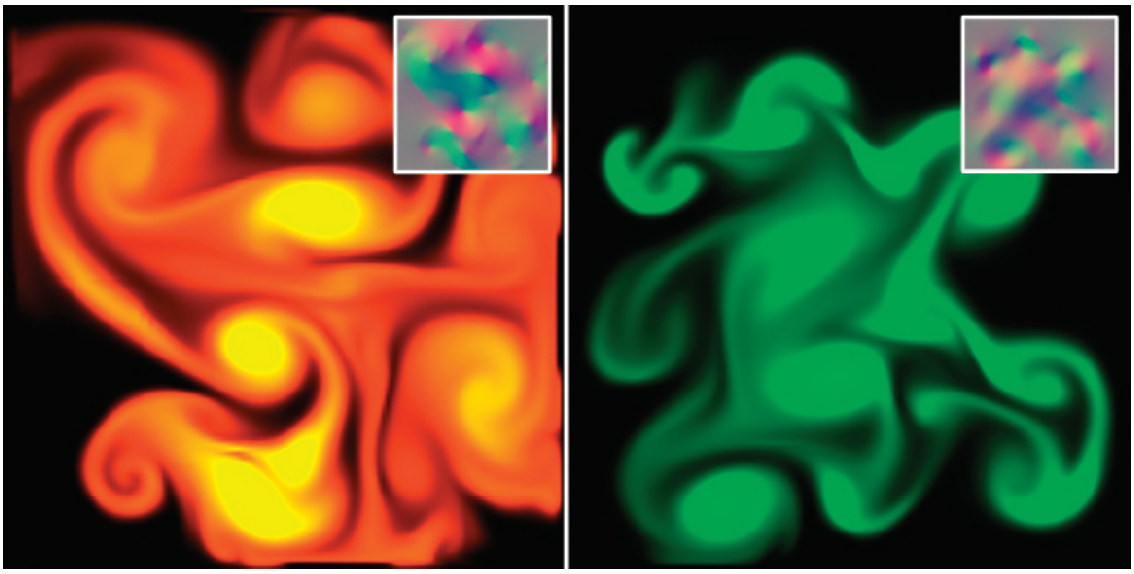

Abbildung 7.2.: Strömungssimulation (Quelle: Mark J. Harris [Fer04])

rakter beibehält (Abbildung 7.2), ist die Berechnung von Strömungen innerhalb einer Flüssigkeit mit Hilfe der Navier-Stokes Gleichung (Kapitel 38 in [Fer04]).

Diese kurze Abhandlung tut dem eigentlichen Thema keine Genüge und die Forschungen in diesem Bereich sind dank der relativ jungen Hardware teilweise noch sehr experimentell. Es ist aber abzusehen, daß zunehmend Bibliotheken in diesem Bereich entstehen und somit Applikationen die Möglichkeit erhalten, bislang auf der CPU ausgeführte Berechnungen transparent in die Grafikkarte zu verlagern. Es ist zu erwarten, daß auf diese Weise Teile der GPU die eigentliche Szenendarstellung durchführen, während andere Bereiche für aufwendigere allgemeine Berechnungen zuständig sind.

Ein Beispiel für eine Verknüpfungsmöglichkeit ist [Jed04], welche auf der GPU das Ray Tracing-Verfahren einmal nicht für Lichtstrahlen, sondern für die Berechnung der Schallausbreitung durchführt. Für die Zugsimulation könnte man auf diese Art und Weise die Beleuchtungsberechnung wie in der vorliegenden Arbeit entwickelt durchführen und gleichzeitig den Schall im Zuginnenraum über eine Allzweck-GPU-Anwendung berechnen.

7.3. Shadersprache

Die Shadersprachen sind wie bereits erwähnt noch in den Anfängen ihrer Entwicklung. Bei GLSL wurde allerdings viel Wert auf eine offene Architektur zwecks zukünftiger Erweiterungen gelegt. Die Sprache bietet teilweise bereits mehr Funktionalität an als die aktuelle Hardware. Ein Flag (*gl_FrontFacing*) welches die Orientierung des aktuell berechneten Polygons angibt existiert zum Beispiel erst in der NV40-Hardware. Dasselbe gilt für die Möglichkeit, auf Texturen aus einem Vertex Programm heraus zuzugreifen. Die "noise"-Funktion[1] wird aber beispielsweise noch von keinem NVidia-Treiber angeboten.

[1] periodische Rauschfunktion, zum Beispiel in Form von Perlin-Rauschen

Es sind bei GLSL auch eine große Anzahl von Schlüsselworten reserviert, so daß einer späteren Eingliederung in den Sprachstandard die Wege geebnet wurden. Die Versionsnummer des unterstützten Sprachstandards bei OpenGL ist wie üblich über eine GL-Erweiterungs-Zeichenkette definiert, so daß Programme zur Laufzeit auf unterschiedlich ausgestattete Hardware reagieren könnnen.

Da Shadersprachen wie alle Sprachen kein festes Produkt sind, sondern der Anpassung an die jeweiligen Möglichkeiten der Hardware unterliegen, wird es interessant zu sehen sein wie sich die Sprachen in den nächsten Jahren verändern werden.

Die Entwicklung bleibt also nach wie vor spannend.

Literaturverzeichnis

[AMH99] Tomas Akenine-Möller and Eric Haines. *Real-time rendering.* A. K. Peters, Ltd., 1999.

[Anj04] Anjuta devstudio. Website, October 2004. `http://anjuta.sourceforge.net`.

[BS03] OpenGL Architecture Review Board and Dave Shreiner. *OpenGL programming Guide (4th ed.): the official guide to learning opengl, version 1.4.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[BS04] OpenGL Architecture Review Board and Dave Shreiner. *OpenGL reference manual (4th ed.): the official reference document to OpenGL, Version 1.4.* Addison-Wesley Longman Publishing Co., Inc., 2004.

[Dia04] Dia a drawing program. Website, August 2004. `http://www.gnome.org/projects/dia`.

[Dis04] Discreet. Website, October 2004. `http://www.discreet.com`.

[Ecc00] Allan Eccles. The diamond monster 3dfx voodoo 1. Website, October 2000. `http://archive.gamespy.com/halloffame/october00/voodoo1`.

[Fer04] Randima Fernando. *GPU Gems.* Addison-Wesley, 2004. `http://developer.nvidia.com/object/gpu_gems_home.html`.

[FvDFH90] James D. Foley, Andries van Dam, Feiner Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice, 2nd ed.* Addison-Wesley, Reading MA, 1990.

[GCC04] Gnu-c++ compiler. Website, October 2004. `http://gcc.gnu.org`.

[Gen04] Gentoo linux. Website, October 2004. `http://www.gentoo.org`.

[GIM04] Gnu image manipulation program. Website, October 2004. `http://www.gimp.org`.

[GKB89] Jack D. Grimes, Les Kohn, and Rajeev Bharadhwaj. The intel i860 64-bit processor: a general-purpose cpu with 3d graphics capabilities. *IEEE Computer Graphics and Applications*, 9(4):85–94, July 1989.

[GNU04] Gnu autotools. Website, July 2004. `http://www.gnu.org`.

[Gou71] Henri Gouraud. Continuous shading of curved surfaces. In *IEEE Transactions on Computers*, volume C-20, pages 623–629, June 1971.

[GPG04] General-purpose computation using graphics hardware. Website, December 2004. http://www.gpgpu.org.

[Gra04] Grace. Website, December 2004. http://plasma-gate.weizmann.ac.il/Grace.

[Hae04] Paul haeberli. Website, December 2004. http://www.sgi.com/misc/grafica/paul.

[HDR04] Hdrc - high dynamic range cmos. Website, December 2004. http://www.hdrc.com.

[Hei00] Wolfgang Heidrich. Environment maps and their applications. *Course Notes, SIGGRAPH 2000*, 2000.

[HS99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In Alyn Rockwood, editor, *Siggraph 1999, Annual Conference Proceedings*, Annual Conference Series, pages 171–178, Los Angeles, 1999. ACM Siggraph, Addison Wesley Longman.

[Jed04] Marcin Jedrzejewski. Computation of room acoustics using programmable video hardware. Master's thesis, Polish-Japanese Institute of Information Technology, 2004.

[JK03] Randi Rost John Kessenich, Dave Baldwin. The opengl shading language, version 1.05. Technical report, February 2003.

[KSS02] Jan Kautz, Peter-Pike Sloan, and John Snyder. Fast, arbitrary brdf shading for low-frequency lighting using spherical harmonics. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 291–296. Eurographics Association, 2002.

[LGP99] Gnu lesser general public license. Website, February 1999. http://www.gnu.org/copyleft/lesser.html.

[LKM01] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158. ACM Press, 2001.

[LyX04] Lyx the document processor. Website, October 2004. http://www.lyx.org.

[May04] Alias maya. Website, October 2004. http://www.alias.com.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[Mic04] Microsoft corporation. Website, October 2004. http://www.microsoft.de.

[min04] Mingw minimalist gnu for windows. Website, September 2004. http://www.mingw.org.

[MKMS04] Rafal Mantiuk, Grzegorz Krawczyk, Karol Myszkowski, and Hans-Peter Seidel. Perception-motivated high dynamic range video encoding. In *Proc. of SIGGRAPH '04*, volume 23, pages 733–741. ACM Press, 2004.

[MM04] Stefanus Du Toit Michael McCool. *Metaprogramming GPU's with SH.* A K Peters Ltd., 2004.

[MSD04] Microsoft developer network. Website, November 2004. `http://msdn.microsoft.com`.

[NAS04] NASA. Electromagnetic spectrum. Website, December 2004. `http://stargazer.gsfc.nasa.gov/epo/resources/resources/background/sun/electromagnetic_spectrum.jsp`.

[NVi04] NVidia. Nvidia gpu programming guide. Website, November 2004. `http://developer.nvidia.com/object/gpu_programming_guide.htm`.

[OGR04] Ogre 3d - object-oriented graphics rendering engine. Website, October 2004. `http://www.ogre3d.org`.

[Ope04a] Opengl - open graphics library application programming interface standard. Website, October 2004. `http://www.opengl.org`.

[Ope04b] The openrt real-time ray-tracing project. Website, March 2004. `http://www.openrt.de`.

[PCM04] People cargo mover. Website, October 2004. `http://www.mwvi.de (Forschungsthemen)`.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. In *Communications of the ACM*, volume 18, pages 311–317, June 1975.

[Pho04] Adobe. Website, October 2004. `http://www.adobe.de`.

[Pix04] Pixar animation studios. Website, November 2004. `http://www.pixar.com`.

[PMWS03] Andreas Pomi, Gerd Marmitt, Ingo Wald, and Philipp Slusallek. Streaming Video Textures for Mixed Reality Applications in Interactive Ray Tracing Environments. In *Virtual Reality, Modelling and Visualization 2003 (VMV) , November 19-21, Munich, Germany*, 2003.

[PTh04] Posix threads for win32. Website, October 2004. `http://sources.redhat.com/pthreads-win32`.

[Pyt04] Python. Website, November 2004. `http://www.python.org`.

[Ren04] Ati rendermonkey. Website, October 2004. `http://www.ati.com/developer/rendermonkey`.

[Ros04] Randi J. Rost. *OpenGL(R) Shading Language.* Addison Wesley Longman Publishing Co., Inc., 2004.

[Rus01] Szymon Rusinkiewicz. Bv - a brdf browser. Website, Jan 2001. http://graphics.stanford.edu/~smr/brdf/bv.

[Saa04] Saarcor - a hardware architecture for real time ray tracing. Website, October 2004. http://www.saarcor.de.

[Sch00] Szeliski, Salesin, Essa, Schödel. Video textures. In *Proceedings SIGGRAPH 2000*, pages 489–498, 2000.

[Sha04] Opengl shader designer. Website, November 2004. http://www.typhoonlabs.com.

[SHM04] Sh: A high-level metaprogramming language for modern gpu's. Website, December 2004. http://libsh.sourceforge.net/docs.html.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 527–536. ACM Press, 2002.

[Ult04] Ultraedit. Website, October 2004. http://www.ultraedit.com.

[WSP04] Michael Wimmer, Dariel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In *Eurographics Symposium on Rendering*, 2004.

[WW99] Alan H. Watt and Allan Watt. *3d Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., 1999.

A. VideoTexturePlugin-Parameter

Die Parameter in Tabelle A.1 werden innerhalb des Ogre-Materialskripts eingefügt. Beispiel für den Aufbau eines Durchlaufs, welcher die Videotextur nutzt:

```
material seatcover
{
    technique
    {
        pass
        {
            vertex_program_ref wood_vertex
            {
            }
            fragment_program_ref wood_frag
            {
                param_named light int 0
                param_named seatcover int 1
            }
            texture_unit
            {
                texture_source vidtex
                {
                    imagebase landscape_tunnel.dds 0 500
                    imagepath /home/pcm/images
                    frames_per_second 25
                    frameinc 1
                    basezeroes 5
                    maxRAM 10
                }
            }
            texture_unit
            {
                texture seatcover.png
            }
        }
    }
}
```

Die Verarbeitung erfolgt durch die Shaderprogramme, die hier als wood_vertex und wood_frag referenziert sind. Der für die Videotextur wichtige Teil ist in der ersten texture_unit (Textureinheit). Die Textureinheiten fangen bei 0 an, in diesem Beispiel werden 2 Textureinheiten verwendet. Im Fragment-Programm kann auf die erste Textureinheit über die “light”-Variable zugegriffen werden und auf die zweite Textureinheit über “seatcover”.

Die erste Textureinheit nutzt als Quelle das vidtex-Plugin und definiert die benötigten Parameter. Zu erwähnen ist noch, daß auf eine bereits definierte vidtex-Quelle in späteren Material-Einträgen nur mit Hilfe der “imagebase”-Angabe (Start/Ende muß ebenfalls weggelassen werden) zugegriffen werden kann. In diesem Fall wird immer die zuerst angegebene Parametrisierung gewählt. Auf diese Weise konnte in der vorliegenden Arbeit die Landschaftstextur von allen Materialien verwendet werden.

Parameter	Bedeutung	Standard
imagebase *Basisname [Start] [Ende]*	Basisname der Bildsequenz. Sequenzen beginnen standardmäßig bei 0 und haben den Aufbau: "*Basisname_Sequenznummer.Erweiterung*". Das verwendete Dateiformat für die Bilder wird aus der Dateierweiterung ermittelt. Start und Endbild kann optional angegeben werden.	-
imagepath *Pfad[;Pfad...]*	Absoluter Suchpfad zu den Bildern. Es können mehrere Pfade angegeben werden, getrennt durch ";". Auf Win32-Systemen müssen die Verzeichnisse durch "\", auf Unix-Systemen mit "/" getrennt werden.	
frameinc *Anzahl*	Zahl, um die der Bildsequenzzähler nach Zugriff auf ein Bild erhöht wird. Nützlich, um zum Beispiel ein Sequenz, welche auf die Wiedergabe mit 24 Bildern pro Sekunde ausgelegt ist mit nur 12 Bildern pro Sekunde zu zeigen (in diesem Fall würde als Anzahl 2 eingetragen).	1
basezeroes *Anzahl*	Maximale Anzahl der führenden Nullstellen für die Sequenznummer.	5
maxRAM *Anzahl*	Maximale Anzahl Bilder, die im RAM zwischengespeichert werden soll. Bei 0 ist der Cache deaktiviert. Falls die Anzahl der Bilder innerhalb einer Sequenz kleiner ist, so wird die gesamte Sequenz in den Speicher gelesen.	25

Tabelle A.1.: Parameter Videotextur-Plugin

Es ist theoretisch möglich, die gleiche Videotextur mit unterschiedlicher Parametrisierung zu laden, allerdings belegt sie in dem Fall genau so viel Speicher wie zwei unterschiedliche Texturen.

B. Vertex/Fragment-Programme

B.1. 'Fragment'-Variante 1 (Ebenengleichung)

B.1.1. Vertex-Programm

```
varying vec3 world_normal, world_vertex;

void main()
{
/* texture coordinates need to be interpolated */
gl_TexCoord[0]=gl_MultiTexCoord0;
/* compute the vertex position */
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
/* we need to interpolate the normal (left in world space ) */
world_normal = gl_Normal;
/* get vertex in world coordinates */
vec4 world_vertex4 = worldmat * gl_Vertex;
world_vertex = world_vertex4.xyz;
}
```

B.1.2. Fragment-Programm

```
uniform sampler2D tex1, tex2;
uniform vec3 A,B;
varying vec3 world_normal, world_vertex;

void main()
{
bool direct_light = true;
bool not_in_plane = false;
bool left_unbounded = false;
float t;
vec3 intersection; /* the intersection point */
vec3 A_plane; /* projection of A into plane (A') */
vec4 col;
/* line vector pointing in direction (not position) of A' */
vec3 A_vector = A - world_vertex;
/* check if we are "in front" of A */
if (world_vertex.z >= A.z)
    left_unbounded = true;
vec3 v = vec3(0.0,0.0,1.0);
vec3 P = vec3(0.0,0.0,B.z); /* B lies inside the plane */
/* project A only if it makes sense */
if (left_unbounded==false){
    t = dot((P-A),v);
    t = t/(dot((A-world_vertex),v));
    A_plane = A + t * (A-world_vertex);
}
```

```
/* now Q */
t = dot((P-world_vertex),v);
t = t/(dot(world_normal,v));
if (t>=0.0)
    intersection = world_vertex + t * world_normal;
else
    not_in_plane = true;
/* now check against the boundaries */
float h = 0.65;
if (
  (not_in_plane == true) ||
  (intersection.y >= (B.y+h)) ||
  (intersection.y < (B.y-h)) ||
  (intersection.x < B.x) ||
  ((left_unbounded==false) && (intersection.x >= A_plane.x)))
    direct_light = false;
if (direct_light == true)
    lod = 9.0;
else
    lod = 1000.0;
/*
    get the normalized normal and use it to generate a simple lookup
    into the light texture
*/
vec3 nnormal = normalize(world_normal);
vec2 normaltex = -vec2(nnormal.x,nnormal.y);
normaltex = (normaltex+1.0)/2.0;
/*
    simple correction factor to allow for increased brightness
    the more direct the angle is towards the windows
*/
float degfac = dot (nnormal,vec3(0.0,0.0,1.0));
degfac = (degfac+1.0)/2.0; /* extend to 360 degrees */
col = texture2D(tex1,normaltex,lod); /* direct lighting */
col = vec4(vec3(length(col.xyz)),1.0);
vec4 undertex = texture2D(tex2,gl_TexCoord[0].xy);
col = vec4(col.rgb*degfac,1.0);
gl_FragColor = undertex*col;
}
```

B.2. 'Fragment'-Variante 2 (Halbvektor)

B.2.1. Vertex-Programm

```
varying vec3 world_normal, world_vertex;

void main()
{
/* texture coordinates need to be interpolated */
gl_TexCoord[0]=gl_MultiTexCoord0;
/* compute the vertex position */
gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
/* we need to interpolate the normal (left in world space ) */
world_normal = gl_Normal;
/* get vertex in world coordinates */
```

```
vec4 world_vertex4 = worldmat * gl_Vertex;
world_vertex = world_vertex4.xyz;
}
```

B.2.2. Fragment-Programm

```
uniform sampler2D tex1, tex2;
uniform vec3 A,B;
varying vec3 world_normal, world_vertex;

void main()
{
vec4 col;
/* to create the boundary we get the half vector */
vec3 view_left = normalize(A - world_vertex);
vec3 view_right = normalize(B - world_vertex);
half_vector = view_left + view_right;
half_vector = normalize(half_vector);
/* set a cutoff-value (1.0...-1.0) */
cutoff_dot = dot (half_vector,view_left);
/*
get the normalized normal and use it to generate a simple lookup
into the light texture
*/
vec3 nnormal = normalize(world_normal);
vec2 normaltex = -vec2(nnormal.x,nnormal.y);
normaltex = (normaltex+1.0)/2.0;
/*
simple correction factor to allow for increased brightness
the more direct the angle is towards the windows
*/
float degfac = dot (nnormal,vec3(0.0,0.0,1.0));
degfac = (degfac+1.0)/2.0; /* extend to 360 degrees */
float lodfac;
vec3 nhalf = normalize(half_vector);
float open_fac = dot(nnormal,nhalf);
if (cutoff_dot<0.0 || open_fac>cutoff_dot)
    lodfac = 9.0;
else
    lodfac = 1000.0; /* account for reduced direct lighting */
col = texture2D(tex1,normaltex,lodfac);
col = vec4(vec3(length(col.xyz)),1.0);
vec4 undertex = texture2D(tex2,gl_TexCoord[0].xy);
col = vec4(col.rgb*degfac,1.0);
gl_FragColor = undertex*col;
}
```

C. Ergebnisdaten Shadervergleich

Die aktuellen Treiber für die NVidia-Karten erlauben es, den generierten Assemblercode zu betrachten. Zum Vergleich wurde auch eine NV4x-Emulation eingeschaltet und die Werte für diese GPUs ermittelt. (siehe Tabelle C.1). Mangels Hardware kann aber keine Aussage zu der Geschwindigkeit gegeben werden, da die Emulation nur in Software abläuft.

GPU	V1 V	V1 F	V2 V	V2 F
NV3x	64/21	10/54	29/28	10/45
NV4x	47/19	10/54	29/26	10/43

Tabelle C.1.: Instruktionslängen der Shader (Vertex/Fragment-Programm)

Auflösung	V1 Vert	V1 Frag	V2 Vert	V2 Frag
20x15	21.5	12.3	17.0	12.5
40x30	21.2	12.3	16.8	12.3
80x60	21.0	12.1	16.7	12.3
160x120	20.6	11.6	16.0	11.8
320x240	17.7	9.8	13.7	10.0
640x480	11.3	6.1	8.7	6.2
800x600	9.0	4.8	6.9	4.9
1200x1600	6.7	3.9	5.5	4.0

Tabelle C.2.: Bilder pro Sekunde bei den beiden Shadervarianten

Zugriffe	ohne Detailstufe	höchste Detailstufe	niedrigste Detailstufe
1	37.0	28.1	27.2
2	20.8	13.6	14.3
3	15.2	9.3	10.2
4	11.7	7.0	8.2
5	9.5	5.6	6.8
6	7.3	4.7	5.8
7	7.0	4.1	5.0
8	6.1	3.6	4.4
9	5.3	3.2	4.0
10	4.8	2.9	3.6

Tabelle C.3.: Analyse Texturzugriffe

D. Beispielmaterialskript für die Shader

```
material seatcover
{
    technique
    {
        pass
        {
            vertex_program_ref lighting_vertex
            {
                param_named_auto worldmat world_matrix
                param_named_auto viewmat view_matrix
                param_named A float3 10.85 0.35 0.0
                param_named B float3 -6.85 0.35 1.4
            }
            fragment_program_ref lighting_frag
            {
                param_named lighttexture int 0
                param_named surfacetexture int 1
            }
            texture_unit
            {
                tex_address_mode clamp
                texture_source vidtex
                {
                    imagebase landscape_tunnel.dds
                }
            }
            texture_unit
            {
                texture seatcover.png
            }
        }
    }
}
```

E. Ogre-Engine

Ogre (*O*bject-oriented *G*raphics *R*endering *E*ngine) [OGR04] ist eine in C++ geschriebene Grafikengine. Sie ist unter der LGPL (Lesser GNU Public License) [LGP99] lizensiert, steht somit im Quelltext zur Verfügung und kann für eigene Zwecke angepaßt werden. Der englische Begriff *engine*, welcher als "Motor, Antrieb" übersetzt werden kann, drückt die Aufgaben dieses Systems aus, es bildet auf Anwendungsebene den Kern der Grafikberechnung.

Ogre entstand als Projekt von Steve Streeting im Jahre 2001 unter der Vorgabe, eine sauber strukturierte und zweckunabhängige 3D-Engine zu schreiben. Die aktuelle (Dezember 2004) Version ist 0.15.1, allerdings wird für Januar/Februar 2005 die Veröffentlichung der Version 1.0 erwartet. Der Aufbau ist plattformunabhängig, die Abstraktion geht dabei soweit, daß selbst die 3D-APIs wie OpenGL und DirectX in Form eines Plugins in das System eingebunden werden. Unterstützt werden dabei sämtliche für die jeweilige API möglichen Shaderformate und -sprachen (HLSL/Cg/GLSL/ARB).

Die durch die Ogre-API angebotenen Klassen und Methoden sind durchweg objektorientiert, eine Erweiterung und Verfeinerung wird an den meisten Stellen durch virtuelle Methoden ermöglicht, das Eventhandling durch abgeleitete Klassen realisiert.

Die Verwaltung einer Szene geschieht durch den Szenenmanager, welcher wiederum in Form von Plugins spezialisiert wird (zum Beispiel binary space partitioning/octree). Objekte werden in Form eines speziellen .mesh-Dateiformates eingebunden, welches neben einem binären Format auch eine XML-Repräsentation besitzt und somit auf einfache Weise zu verarbeiten ist. Exporter für alle größeren Modelling-Programme (Maya, 3DS, Blender) stehen zur Verfügung und liefern neben den Oberflächendaten auch die Skelette (engl. *bones*) und dazugehörige Animationen und natürlich die Texturkoordinaten (UV). Hardware-beschleunigtes Skinning über die Gewichtung der Eckpunkte ist möglich, genauso wie das Laden von biquadratischen Bézierflächen. Ebenfalls möglich sind progressive Meshdaten (verwendete Daten abhängig von der Detailstufe).

Ein Ressourcemanager kümmert sich um das Laden und Bereitstellen von Daten (auch gepackt, zum Beispiel als ZIP), Partikelsysteme für Spezialeffekte stehen zur Verfügung genauso wie Skyplanes, Skydomes und Billboards. Transparente Objekte werden automatisch korrekt angeordnet, Textur- und Stencilschatten lassen sich auf einfache Weise per API-Kommando einschalten.

Für die Entwicklung wird ein Speichermanager mit Debugmöglichkeit angeboten, Fehler werden über Exceptions gemeldet.

Eine Fülle von unabhängigen Erweiterungen durch die OpenSource-Gemeinschaft hat bereits zu Anbindungsmöglichkeiten an Physik-Engines wie ODE (Kollisionen und Beschleunigung) geführt.

Besonders hervorzuheben ist die umfangreiche Dokumentation und eine große Anzahl an Beispielen, so daß ein schneller Einstieg möglich ist.

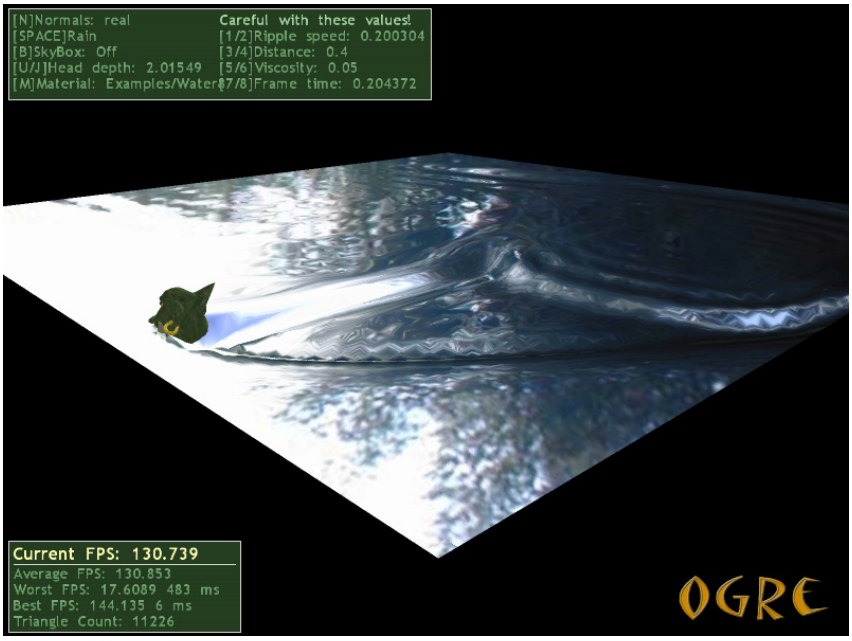

Abbildung E.1.: Ogre Wasser-Simulation

F. Glossar

AGP (Accelerated/Advanced Graphics Port): beschleunigter/erweiterter Grafikport, Grafikbus, der speziell für Anwendungen mit großen Datenmengen (Texturen, Objektdaten) entworfen wurde. Maximale Busbandbreite bei AGP 3.0 (8x-Modus) zur Grafikkarte: 4 Byte x 533.3 MHz = 2.1 GByte/sek. Aber: in Gegenrichtung steht nur 1x zur Verfügung.

API (Application Programming Interface): Definition einer Programmierschnittstelle für eine Anwendung.

Anti-aliasing: bezeichnet Maßnahmen, die gegen Aliasing-Effekte ergriffen werden. Darunter versteht man in der Signaltheorie die Entstehung von (Alias-)Frequenzen in einem abgetastetem Signal, welche im Original nicht vorhanden waren. In der Computergrafik kann dieser Effekt zum Beispiel bei der Wiedergabe einer diagonalen Linie auf einem Bildschirm sichtbar werden. Ist die Linie "dünn", ist also die für ihre Darstellung nötige hohe Ortsfrequenz nicht "erreichbar" (Pixelabstand zu groß), so bilden sich bei der Rasterung Aliasfrequenzen. Man bemerkt eine "Treppchenbildung". Es gibt eine Reihe von Anti-Aliasing-Methoden mit unterschiedlichen Ansätzen, gemeinsam ist die Erhöhung der Anzahl der Abtastwerte, die bei der Einfärbung eines Pixels einfließen.

Bump-Mapping: Verfahren, um eine Polygonoberfläche unterhalb der durch die Eckpunktauflösung vorgegebenen Grenzen noch in ihrer Topologie verändern zu können. Es ändert dafür die interpolierten Normalen. Ein Höhenparameter wird zum Beispiel durch eine Textur vorgegeben (1.0 = größtmögliche Höhe, 0.0 = kleinstmögliche Höhe). Der Eindruck der Unebenheit wird allein durch eine Modulation der Schattierung (engl *shading*) der jeweiligen Oberflächenpixel erzeugt.

Displaycontroller: Steuereinheit, welche spezielle Kommandos oder Benutzereingaben verarbeitet und in Pixelinformationen (*engl. bitmaps*) umsetzt. Oft in Software realisiert.

High Dynamic Range (HDR) Bilder: allgemeine Bezeichnung für Bilder mit hohem Dynamikbereich, die mehr als die traditionellen 8 Bit pro Farbkanal speichern. Üblich sind 16-Bit und Gleitkommazahlen. Letztere erlauben die Speicherung von Helligkeitswerten über mehrere Zehnerpotenzen und erlauben somit eine vollständige Abbildung der Szenenleuchtdichte.

Mipmapping: (von lat. *multum in parvo*, "viele in einem"). Anti-Aliasing-Verfahren für Texturen. Da Texturen eine feste Auflösung besitzen und perspektivisch korrekt auf eine Objekt abgebildet werden müssen kommt es bei den in Echtzeit üblichen Transformationsverfahren zu Aliasingstörungen. Wie aus der Signaltheorie bekannt,

ist es daher nötig, die Texturen vorzufiltern. Da dies Rechenzeit kostet, werden beim Mipmapping-Verfahren bereits vorgefilterte Texturen in den Videospeicher geladen. Für jede Stufe wird die Dimension in beide Richtungen solange um Faktor 2 reduziert bis eine Auflösung von 1x1 erreicht ist. Alle Texturen zusammen haben einen Platzbedarf von $\frac{4}{3}$ * Originalgröße.

Normal-Mapping: spezielles *Bump-Mapping*-Verfahren, bei dem die verwendete Normalentextur fest mit den Texturkoordinaten eines Objekts verbunden ist. Wird verwendet, um ein niedrig aufgelöstes Polygonmodell höher aufgelöst erscheinen zu lassen. Der Begriff wird oft synonym zu *Bump-Mapping* verwendet.

Parser: Lexikalischer Analysator. Zerlegt eine Zeichenkette wie "imagepath /mnt/daten" in ihre logischen Bestandteile (im Beispiel: erstes Wort = Kommando, weitere Worte = Optionen für dieses Kommando) und prüft, ob die vorgeschriebene Syntax eingehalten wurde. Das Ergebnis ist der Inhalt der Zeichenkette in einer von Programmcode auswertbaren Form.

PCI (Peripheral Component Interconnect): Standard zur Anbindung von Peripherie-Komponenten. Definiert einen PC-Übertragungsbus (physikalisch und auf Protokollebene), um darüber die Fähigkeiten eines PC-Systems erweitern zu können. Bietet in der bei Desktopsystemen üblichen Variante (33,3 MHz/32 Bit) nur eine Bandbreite von 133 MByte/sek. Die modernste PCI-X 2.0-Variante bietet zwar 2,1 GByte/sek, größere Verbreitung wird aber der PCI-Express finden.

PCI-Express: gleiche Aufgaben wie der PCI-Bus, die Übertragung findet aber seriell statt. Es ist ein lokaler Bus (direkte Kommunikation mit CPU), dadurch hoher Datendurchsatz. Es gibt verschiedene physikalische Ausführungen, die sich in der Anzahl der gleichzeitig nutzbaren Kanäle (engl. *lane*, also eigentlich "Fahrbahn") unterscheiden. Spezifiziert sind Abstufungen von 1-32. Eine 16x-Grafikkarte kann theoretisch in beide Richtungen gleichzeitig mit 4 GByte/sek übertragen. Es wird erwartet, daß PCI-Express den bislang üblichen PCI-Bus ersetzen wird.

photorealistisch: Eigenschaft, welche in der Computergrafik verwendet wird, um Bilder zu beschreiben, deren Inhalt das menschliche Auge als "wirklichkeitsgetreu" oder "real" wahrnimmt.

Plugin: zu deutsch etwa "Steckmodul". Im Allgemeinen eine ausführbare Datei, welche die Funktionalität einer Bibliothek oder eines Programms zur Laufzeit erweitert. Oft in Form einer dynamischen Bibliothek realisiert.

Thread: deutsch "Faden". Ein Programm im üblichen Sinne läuft innerhalb eines Prozessraumes ab. Dabei erfolgt die Ausführung der Instruktionen sequentiell. Für bestimmte Aufgaben kann es sinnvoller sein, eine Aufteilung in Teilaufgaben vorzunehmen, welche dann parallel ausgeführt werden können. Bei einem Einzelprozessorsystem wird die scheinbare Parallelität durch Kontextumschaltung erreicht, dabei wird jeder Thread an einer Stelle unterbrochen und zu einem späteren Zeitpunkt wieder aufgenommen.

Vertex (pl. Vertices): Eck-/Verbindungspunkt. Ein Polygon-3D-Modell wird durch Grundformen (engl. *primitives*) beschrieben, z.B. Punkte, Linien, Dreiecke, Vierecke. Ein Dreieck besitzt z.B. 3 Eckpunkte oder Vertices.

Videocontroller: Steuereinheit, welche die Informationen eines Grafikpuffers zur Ausgabe auf einem Anzeigegerät (Monitor, Fernseher) aufbereitet. Der Puffer wird zu diesem Zweck mit einer definierten Wiederholrate zeilenweise ausgelesen und die Pixelinformationen werden entweder über Tabellen (engl. *LUT, Lookup-Table*) oder als direkter Intensitätswert eines Pixels auf dem Monitor dargestellt.

Index

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, daß ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfaßt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Düsseldorf, 5. Januar 2005

Datum (Carsten Juttner)