Modellierung der Umweltbeleuchtung durch Fahrzeugscheinwerfer in der Fahrsimulation

Diplomarbeit

vorgelegt von

Simon Knebel

BMW Group

Institut für Computervisualistik
AG Computergraphik

Forschung und Technik
Projekt Fahrsimulation

Betreuer:	Dr. -Ing. Alexander Huesmann, BMW Group
Prüfer:	Prof. Dr. Stefan Müller, Universität Koblenz

Dezember 2004

Danksagung

An dieser Stelle möchte ich all jenen danken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Diplomarbeit beigetragen haben:

- Herrn Prof. Dr. Stefan Müller, für die Betreuung meiner Diplomarbeit und vor allem dafür, dass er mir das Thema der Lichtsimulation nähergebracht hat
- Herrn Dr.-Ing Alexander Huesmann, Marc Breithecker und Martin Strobl, für die engagierte Betreuung meiner Diplomarbeit seitens BMW
- Herrn Patrick Kuhl, für die Bereitstellung von Scheinwerfer-Messdaten der BMW Lichttechnik
- Den Mitarbeitern des Fahrsimulations-Projektes, für die angenehme und entspannte Arbeitsatmosphäre
- Patrick Gentzcke und meiner Schwester Susanne Knebel, für das mühsame Korrekturlesen
- Besonders bedanken möchte ich mich außerdem bei meiner Freundin Steffi Müller, dafür dass sie mir auch in stressigen Phasen der Diplomarbeit immer verständnisvoll zur Seite gestanden hat. Nicht zuletzt gilt mein Dank vor allem meinen Eltern Manfred und Maria Knebel, die mir während meines gesamten Studiums immer persönlichen und finanziellen Rückhalt gegeben haben.

Zusammenfassung

Im Rahmen dieser Diplomarbeit wird ein Konzept zur Integration der Umweltbeleuchtung durch Fahrzeugscheinwerfer für die BMW Fahrsimulation entwickelt. Als Basis dafür dient die Recherche über entsprechende Verfahren lokaler, globaler und texturbasierter Beleuchtungssimulationen und die Betrachtung der Schattenberechnungsverfahren planarer Schatten, Shadow Volumes und des Shadow Mappings. Unter Berücksichtigung photometrischer Aspekte der Beleuchtungssimulation und der Anforderungen seitens BMW hat sich die Beleuchtung mittels projektiver Texturen als geeignetes Mittel zur Simulation der Beleuchtung durch Fahrzeugscheinwerfer herausgestellt.
Zur weiteren Überprüfung dieses Verfahrens im Hinblick auf dessen potentielle Nutzung für die Fahrsimulation wird eine prototypische Implementierung der Beleuchtungssimulation entwickelt und in die firmeneigene Simulationssoftware SPIDER integriert. Die Realisierung erfolgt dabei mittels *OpenGL Shading Language* auf programmierbaren Graphikkarten.
Die prototypische Realisierung der Beleuchtungssimulation liefert qualitativ hochwertige Resultate, weist jedoch noch Performanceprobleme auf. Diese Erkenntnis wird bei der Entwicklung eines Implementierungskonzeptes zur Integration der Beleuchtungssimulation in SPIDER berücksichtigt. Lösungsalternativen für die genannten Probleme werden aufgezeigt.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ALC	Adaptive Light Control
ALS	Advanced Lighting Simulation
API	Application Programming Interface
ARB	(OpenGL) Architecture Review Board
BB	Bounding Box
BMW	Bayerische Motoren Werke
BRDF	Bidirektionale Reflexions und Distributions Funktion
BSP	Binary Space Partitioning
CAN	Controller Area Network
Cg	C for graphics
CIE-XYZ	Commission Internationale de l'Eclairage - *Farbsystem*
CPU	Central Processing Unit
DIN	Deutsche Industrie-Norm
FFP	Fixed Function Pipeline
flt	OpenFlight - *Datenformat*
fps	frames per second
FS	Fragment-Shader
GLSL	OpenGL Shading Language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HLSL	High Level Shading Language
IO	Input/Output
LOD	Level Of Detail
pfb	perfomer binary - *Datenformat*
RGB	Rot Grün Blau - *Farbsystem*
RTRT	Real Time Ray Tracing
SGI	Silicon Graphics Incorporated
SIMD	Single Instruction Multiple Data
SPIDER	Software Programming Interface for Distributed Real-Time Driving Simulation
VS	Vertex-Shader

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Um die Sicherheit der eigenen Fahrzeuge zu erhöhen und der Konkurrenz möglichst einen Schritt voraus zu sein, sind die Automobilhersteller ständig gezwungen, neue, innovative Systeme zu entwickeln. Für die erforderlichen Tests dieser Systeme ist man während ihres Entwicklungsprozesses auf zahlreiche Testfahrten und deren Auswertung angewiesen. Prototypische Systeme direkt im Fahrzeugbetrieb zu testen, ist aber mit einem großen zeitlichen und finanziellen Aufwand verbunden und kann insbesondere bei der Erprobung von Fahrerassistenzsystemen ein nicht zu unterschätzendes Sicherheitsrisiko darstellen. Vor allem durch die steigende Komplexität der Fahrzeuge wird die Anzahl der benötigten Testfahrten immer größer. Im Zuge der virtuellen Absicherung kommen aus diesem Grund in der Automobilindustrie immer öfter Fahrsimulatoren zum Einsatz, da diese den Herstellern eine realistische, sichere und reproduzierbare Umgebung für Entwicklung und Forschung bieten. Damit tragen Fahrsimulatoren dazu bei, den finanziellen und zeitlichen Aufwand durch eine Verringerung der notwendigen realen Testfahrten im Rahmen zu halten, und ermöglichen die Erprobung sicherheitskritischer Systeme. Lange bevor im Entwicklungsprozess Straßen-Testfahrten möglich sind, lassen sich in der simulierten Fahraufgabe Informationen über den subjektiven Eindruck der getesteten Systeme gewinnen. Die Reaktionen der Probanden sind natürlich nur dann mit dem Fahrverhalten im Straßenverkehr zu vergleichen, wenn auch die Simulation an die realen Straßenverhältnisse angelehnt ist.
Zu einer realistischen visuellen Darstellung gehört daher auch die Simulation von Nachtfahrsituationen, da die Wahrnehmung des Fahrers in diesen Verhältnissen beeinträchtigt ist. Dadurch kann es, beispielsweise aufgrund veränderter Geschwindigkeitswahrnehmung, zu einem veränderten Fahrverhalten kommen. Um dies berücksichtigen zu können, wurde die im Zuge des BMW *Adaptive Light Control* Projektes [LBR+99] entwickelte *Advanced Lighting Simulation* [LSB+01] in den firmeneigenen Fahrsimulator integriert. Durch den Einsatz einer neuen Simulationssoftware musste diese Realisierung zwischenzeitlich angepasst werden und basiert nun auf speziellen Lichtquellen des Szenengraphs *Performer* (vgl. Kapitel 2.1). Die Motivation dieser Arbeit ist, eine vergleichbare Beleuchtungssimulation zu entwickeln, die nicht an Performer gebunden ist.

1.2 Ziele der Arbeit

Das Ziel dieser Arbeit ist die Simulation von Nachtfahrten in der BMW Fahrsimulationsumgebung. Dazu findet zunächst eine Recherche über den derzeitigen Stand von echtzeitfähigen Lichtsimulationsverfahren statt. Darauf basierend liegt der Fokus der Arbeit auf der Simulation des Lichtkegels der Fahrzeugfrontscheinwerfer, also auf deren Beleuchtung der Umwelt. Die Beleuchtungssimulation wird für das Eigenfahrzeug[1] realisiert. Bei der Umsetzung muss darauf geachtet werden, dass die Simulation auch auf Fremdfahrzeuge übertragen werden kann.
Untrennbar mit einer Beleuchtung verbunden ist jedoch auch der dadurch verursachte Schattenwurf. Die Fahrzeugscheinwerfer haben nicht nur Einfluss auf die Beleuchtung der Umwelt, sondern auch auf deren Veränderung durch den Schattenwurf beliebiger Objekte in der Szene. Vor allem der Wechsel von Licht und Schatten in Nachtfahrsituationen erfordert eine erhöhte Aufmerksamkeit des Fahrers und kann zu Fehleinschätzungen von Geschwindigkeiten und Abständen führen.
Neben der Simulation des Lichtkegels wird daher in dieser Arbeit auch die Berechnung und Darstellung des Schattenwurfs betrachtet. Die Untersuchung verschiedener Schattenberechnungsverfahren und die Möglichkeiten, diese in Echtzeit im Fahrsimulator zum Einsatz zu bringen, werden betrachtet und ausgewertet.
Die Beleuchtung der Umwelt durch die Frontscheinwerfer des Eigenfahrzeugs wird prototypisch in die BMW Fahrsimulation integriert. Besonderes Augenmerk liegt dabei auf der Verwendung programmierbarer Graphikkarten, mit deren Hilfe die gewählten Verfahren beschleunigt werden sollen. Um einen Vergleich zu ermöglichen, werden dabei als Referenzimplementation einer Lichtsimulation die in *Performer* (vgl. Kapitel 2.1) integrierten projektiven Lichtquellen herangezogen. In der zu realisierenden Lichtsimulation wird hingegen auf eine von Performer unabhängige Lösung Wert gelegt.
Auf Basis der prototypischen Implementierung ist es möglich, Aussagen über die Performance und Qualität des gewählten Verfahrens zu treffen, und die Eignung für den Einsatz im Fahrsimulator zu bewerten. Ziel dieser Arbeit ist damit ferner, die Erkenntnisse aus Recherche und prototypischer Implementierung zu evaluieren, und anschließend auf dieser Basis ein Konzept für die Integration von Scheinwerferlicht und Schattenwurf in die BMW Fahrsimulationsumgebung SPIDER zu entwickeln.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in sieben Kapitel. Im Anschluss an diese Einleitung wird in zwei Kapiteln auf die zum Verständnis der Arbeit benötigten Grundlagen eingegangen. Dabei wird zunächst ein Überblick über die Fahrsimulation innerhalb der BMW Group Forschung und Technik gegeben. Die zum Einsatz kommende Hard- und Software wird als Basis für die Untersuchungen vorgestellt, um einen Eindruck von den Rahmenbedingungen zu vermitteln, an die die Fahrsimulation gebunden ist. Anschließend werden die Grundlagen der Computergraphik betrachtet, soweit sie

[1] Als Eigenfahrzeug wird in dieser Arbeit das Fahrzeug bezeichnet, aus dessen Sicht der Anwender die Fahrsimulation erlebt. Andere Fahrzeuge innerhalb der Simulation werden als Fremdfahrzeuge bezeichnet.

die Thematik dieser Arbeit betreffen. Dies sind hier die Funktionsweise der Graphik-Rendering Pipeline und der Einsatz von Vertex- und Fragment-Shadern sowie Beleuchtungsmodelle und deren radiometrische beziehungsweise photometrische Grundlagen.
Auf Basis dieser Grundlagen werden in Kapitel 4 verschiedene Aspekte echtzeitfähiger Beleuchtungssimulation betrachtet. Dazu zählen verschiedene Echtzeit-Beleuchtungsverfahren, eine photometrische Betrachtung der Beleuchtungssituation sowie die Schattenintegration. Die aus dieser Recherche gewonnenen Erkenntnisse gehen in die abschließende Bewertung ein.
Als Basis für die weitere Vorgehensweise wird in Kapitel 5 zunächst eine Anforderungsliste an eine Realisierung erstellt. Unter Berücksichtigung der daraus resultierenden Prioritäten können die während der Recherche gewonnenen Erkenntnisse in ein Konzept einer prototypischen Implementierung der Beleuchtungssimulation einfließen. Diese Implementierung wird in Kapitel 5.2 erläutert. Die Erkenntnisse der prototypischen Realisierung gehen zusammen mit Ergebnissen von Perfomancetests, in die Entwicklung eines Konzeptes zur Integration einer Beleuchtungssimulation in SPIDER ein.
In Kapitel 6 werden die Ergebnisse der prototypischen Realisierung auf deren Performance und Qualität hin untersucht und den formulierten Anforderungen gegenübergestellt.
Diese Gegenüberstellung dient als Basis für das in Kapitel 7 gezogene abschließende Fazit über die gewonnenen Erkenntnisse. Verbesserungsmöglichkeiten und Einsatzbereiche der behandelten Verfahren werden erörtert, und geben Anlass, ein Blick in die Zukunft zu werfen.

Kapitel 2

Fahrsimulation in der BMW Group Forschung und Technik

Im Fahrsimulationsprojekt der BMW Group Forschung und Technik werden drei statische und ein dynamischer Fahrsimulator eingesetzt, die zum einen von *ONYX2* Graphikrechnern der Firma *Silicon Graphic Inc.*, zum anderen von Linux-PC Clustern betrieben werden. Langfristiges Ziel ist dabei der Übergang zu „commercial of the shelf"-Produkten. Bei dieser Strategie wird teure Spezialhardware durch den Einsatz von untereinander vernetzten Workstations aus dem Handel ersetzt. Dies stellt durch immer leistungsfähigere Graphikkarten mittlerweile eine attraktive Alternative zu spezieller Graphikhardware dar. Als Referenz für diese Arbeit wird im Folgenden die verwendete Hard- und Software exemplarisch anhand eines der drei statischen Fahrsimulatoren dargestellt.

2.1 Fahrsimulationssoftware SPIDER

Für die BMW Fahrsimulation wird eine firmeneigene, sehr flexible Echtzeit-Simulationssoftware namens *SPIDER* (Software Programming Interface for Distributed Real-Time Driving Simulation) eingesetzt, die auf dem Szenengraph *Performer* von *Silicon Graphis Inc.* aufsetzt. Mit dieser Software sind simulierte Autofahrten mit Fremdverkehr in verschiedenen dreidimensionalen Szenarien wie Autobahnen, Landstraßen und Städten möglich. Die Software zeichnet sich vor allem durch ihren modularen Aufbau aus.
Die Simulation setzt sich unter anderem aus den im Folgenden aufgeführten Simulationskomponenten zusammen, die alle in eigenen Modulen realisiert sind:

- Fahrdynamikberechnung
- Sichtsimulation
- Soundsimulation
- Instrumentensimulation
- Anbindung der Bedienelemente im Fahrzeug

Zur Realisierung zusätzlicher Funktionalität ist es möglich dynamische Plug-Ins zu programmieren, die hinzugeladen oder entfernt werden können. Die modulare Architektur von SPIDER erlaubt es außerdem, die verschiedenen Aufgaben der Simulation verteilt im Netzwerk zu berechnen, und so durch die Integration mehrerer Workstations die Performance zu erhöhen. Der Austausch der dabei enstehenden Simulationsdaten erfolgt über den Datenpool von SPIDER:
Anfallender Datenoutput wird von den verschiedenen Modulen oder Plug-Ins im Datenpool registriert und benötigter Dateninput entsprechend von dort angefordert. Der Austausch von Kontroll- und Konfigurationsbefehlen wird dabei strikt vom Simulationsdatenaustausch getrennt, um diesen nicht zu stören. Die Konfiguration und Bedienung der Simulation erfolgt anhand eines einzigen zentralen GUI (Graphical User Interface).
Die 3D-Datenbasen, die bei den BMW Fahrsimulationsversuchen zum Einsatz kommen, liegen im *Open Flight(.flt)* beziehungsweise *Performer Binary(.pfb)* Datenformat vor. Zu jeder der graphischen Datenbasen existiert außerdem eine logische Straßenbeschreibung, die als Grundlage für die Positionierung der Fahrzeuge auf der Straße dient.

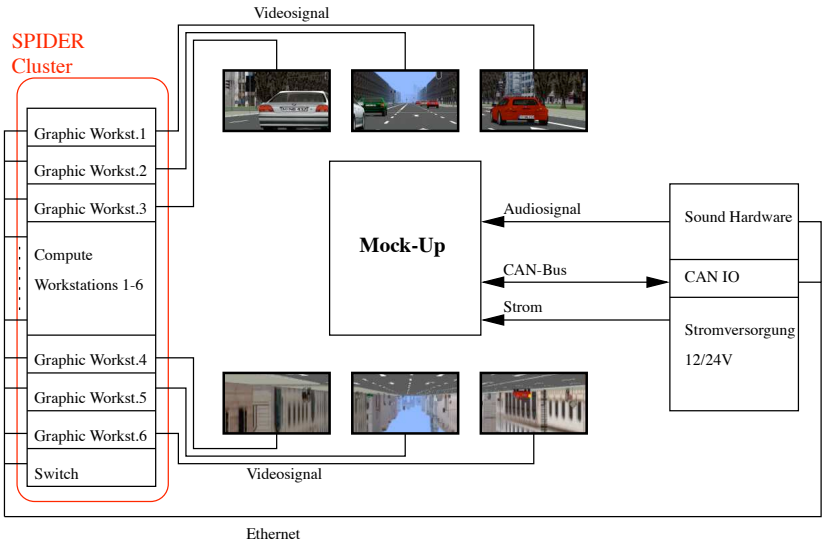


Abbildung 2.1: Der informationstechnische Aufbau eines statischen Simulators

2.2 Infrastruktur und Hardware

Da das Werkzeug Fahrsimulation für die verschiedensten Anwendungszwecke eingesetzt wird, muss sich die Fahrsimulationsumgebung ständig an neue Anforderungen anpassen können. Seitens der Software wird dies unter anderem durch das Plug-In-Konzept sichergestellt, doch auch die Fahrkabinen, sogenannte Mock-Ups, müssen adaptierbar sein. Mock-Ups stellen die Schnittstelle zwischen Mensch und Fahrsimulator dar. Damit sie an die jeweiligen Versuchsanforderungen angepasst werden können, sind sie nach dem „Baukastenprinzip“ konstruiert. Standardisierte

und schnell verfügbare Komponenten wie beispielsweise Lenkrad, Pedale oder Cockpit, können zur Ausstattung der Mock-Ups verwendet werden und bieten damit eine hohe Flexibilität.
Da sich die Simulation von Licht und Schatten auf die Sichtsimulation beschränkt, wird im Folgenden vor allem die in diesem Zusammenhang relevante Infrastruktur betrachtet, andere Komponenten werden vernachlässigt.
Wie im Versuchsaufbau in Abbildung 2.1 dargestellt ist, werden Eingabesignale des Mock-Ups über einen *CAN* (Controller Area Network)-Bus zu einer speziellen CAN-Schnittstelle eines IO-Rechners geschickt. Hier werden die CAN-Daten konvertiert und anschließend im SPIDER-Datenpool abgelegt. Nach einigen weiteren Stationen erreichen die Daten schließlich die Graphikrechner, die das Rendering der einzelnen Kanäle übernehmen. Der Fahrsimulator wird meist mit 6 Kanälen für die Sichtsimulation betrieben. Die Frontsicht aus dem Fahrzeug nimmt dabei 3 Kanäle (3 Plasma Bildschirme oder alternativ 3 Projektoren) und die Rückansicht 3 weitere Kanäle für die Sicht durch einen Innen- und zwei Außenspiegel (Plasmabildschirme) in Anspruch. Jeder dieser Kanäle wird von einer Graphik-Workstation mit 2 *Xeon* Prozessoren und einer *nVIDIA Quadro FX3000G* Graphikkarte berechnet, die an ein 1Gbit/s Ethernet Netzwerk angeschlossen ist. Um diesen Transfer nicht zu behindern, ist dieses Netzwerk nur für den Austausch von Simulationsdaten zuständig, alle Kontroll- und Konfigurationsbefehle werden wie bereits erwähnt über ein separates 100 Mbit/s Ethernet Netzwerk verschickt.

Kapitel 3

Grundlagen der Computergraphik

Als Basis für die Thematik der vorliegenden Arbeit werden im Folgenden die dafür erforderlichen Grundlagen der Computergraphik erläutert. Dies kann nur als kurze Übersicht angesehen werden, zur weiteren Vertiefung der Grundlagen sei daher auf [Shi02], [FvDFH96], [Wat93] und für Themen der echtzeitfähigen Computergraphik (siehe Kapitel 4 auf Seite 24) auf [AMH02] verwiesen.

3.1 Graphik Rendering Pipeline

Die Graphik Rendering Pipeline stellt den eigentlichen Kern der Computergraphik dar. Ihre Aufgabe besteht darin, ein 3D-Modell inklusive dessen Beleuchtung und Texturen auf eine perspektivisch korrekte zweidimensionale Repräsentation abzubilden. Die Geschwindigkeit des Renderingvorgangs kann dabei maximal so schnell sein wie das langsamste Glied in der Kette (*Bottleneck*), welches damit auch die Bildaktualisierungsrate (*fps = frames per seconds*) bestimmt. Mit der neuesten Generation der Graphikkarten ist es zudem möglich geworden, einige Aufgaben innerhalb der Rendering Pipeline in Form von sogenannten Vertex- und Fragment-Shadern auf programmierbaren GPUs auszuführen. Im Folgenden wird die klassische *Fixed Function Pipeline (FFP)* anhand der in Abbildung 3.1 dargestellten Standard *OpenGL*-Rendering Pipeline beschrieben. Deren gezielte Umgehung mittels Vertex- und Fragment-Shadern wird anschließend in Kapitel 3.2 auf Seite 9 erläutert. Jedes Objekt einer dreidimensionalen Szene besitzt ein eigenes lokales Koordinatensystem, auch als Objekt- oder Modellkoordinatensystem bezeichnet. Daher liegen zu Beginn der Pipeline alle Vertices in homogenen Objektkoordinaten vor. Im ersten Schritt wird die Modelview Matrix auf diese Koordinaten angewandt. Dadurch werden gleichzeitig zwei Dinge erledigt: Zuerst wird das Objekt im Weltkoordinatensystem positioniert, ausgerichtet und skaliert und somit in Weltkoordinaten umgerechnet (bezeichnet als Modeling Transformation). Anschließend erfolgt diese Positionierung und Ausrichtung auch für die Kamera, und das Objekt wird in das Koordinatensystem der Kamera umgerechnet (View Transformation). Als Ergebnis dieses Schrittes befindet sich das 3D-Objekt jetzt im Augkoordinatensystem,

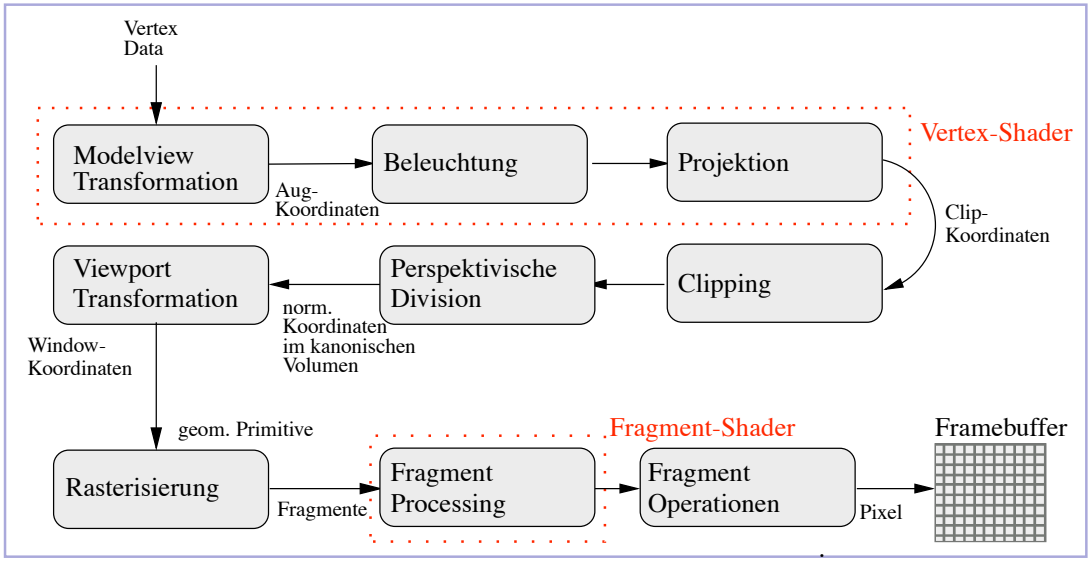


Abbildung 3.1: Die Graphik Rendering Pipeline im Hinblick auf die Vertex-Verarbeitung

die Kamera ist im Ursprung positioniert und entlang der negativen z-Achse[1] ausgerichtet.
Modeling- und View Transformationen sind in OpenGL aus Effizienzgründen direkt zur 4 x 4 Modelview Matrix zusammengefasst. Nach der Modelview Transformation befinden sich die Polygoneckpunkte und deren Normalen im Augkoordinatensystem. Daher findet hier auch die Beleuchtung der Vertices (*per-vertex*) statt. Die reinen per-vertex Operationen sind ab diesem Zeitpunkt abgeschlossen und die Pipeline geht zur *Primitive Assembly Phase* über, dem Aufbau geometrischer Primitive.
Durch die Anwendung der Projektionsmatrix wird das View Frustum in einen Quader und damit die Koordinaten der Vertices in Clip-Koordinaten konvertiert. Der Quader repräsentiert den Bereich der Szene, der später auf dem Bildschirm dargestellt wird. Das aus der Projektion resultierende Linkssystem mit nicht linear skalierter z-Achse dient zum Clipping der geometrischen Primitive. Objekte, die außerhalb des Quaders liegen, werden ignoriert und nicht weiter durch die Pipeline gereicht. Für den Fall, dass sich ein Objekt nur zur Hälfte innerhalb des Quaders befindet, muss ein neuer Vertex an dessen Rand erzeugt werden und der außerhalb liegende Teil des Primitives wird geclippt.
Im Anschluss an das Clipping erfolgt das Normalisieren der Koordinaten des Quaders durch die perspektivische Division, der Quader wird zum sogenannten *kanonischen Volumen*, einem Einheitswürfel der spezifiziert werden kann als $(x, y, z) \in [-1, 1]^3$. Zur Transformation der dreidimensionalen Koordinaten innerhalb des kanonischen Volumens in zweidimensionale Bildschirmkoordinaten findet die Viewport Transformation statt.

[1]Das Performer Koordinatensystem ist im Vergleich zu dem von OpenGL um 90 Grad gegen den Uhrzeigersinn rotiert. Die Kamera ist entlang der positiven y-Achse ausgerichtet.

Während der folgenden Rasterisierung wird für jedes Bildschirmpixel entschieden, welche Teile der geometrischen Primitive sie abbilden. Unter Berücksichtigung von Farbe, Tiefeninformationen und dem verwendeten Shadingmodell[2] werden sogenannte Fragmente generiert. Für diese kleinen quadratischen Flächen werden die Vertex-Attribute wie Farbe, Texturkoordinate und z-Wert interpoliert. Die Fragmente werden später als Pixel im Framebuffer abgebildet. Nach der Rasterisierung werden die Fragmente im Fragment-Processing weiteren Operationen unterzogen. Bisher sind die Objekte der Szene nur eingefärbt und schattiert, Texturen werden erst jetzt berücksichtigt. Dadurch, dass für die Fragmente während der Rasterisierung Textur- und Objektkoordinaten aus den entsprechenden Koordinaten der Vertices ermittelt wurden, ist es möglich, das zugehörige *Texel* (Texture Element) aus einem Texturspeicher zu laden und mit dem Fragment zu interpolieren (*Texture Mapping*). Während des Fragment-Processings werden eventuell noch Operationen wie beispielsweise Nebelberechnung oder *COLOR-SUM* (Kombination der primären und sekundären Farbe eines Fragmentes) durchgeführt.
Die Fragmente werden nun den *per-fragment* Operationen wie dem Blending zwischen Fragment- und Framebufferfarbe, oder verschiedenen Tests (Alpha-, Tiefen-, Stencilbuffertest usw.) unterzogen (für Details siehe [SA03]). Anschließend erfolgt die Speicherung der daraus resultierenden Pixelfarben im Framebuffer. Um zu verhindern, dass es zu Verzögerungen im Bildaufbau kommt, wird die Farbe eines jeden Pixels in einem doppelten Colorbuffer gespeichert, einem Speicherbereich des Framebuffers, der für jedes Pixel drei Werte (RGB) speichern kann. Dadurch dass sich um einen doppelten Buffer handelt, kann der Inhalt des vorderen Buffers bereits angezeigt werden, während im hinteren die berechneten Pixelwerte gespeichert werden. Ist der Bildaufbau abgeschlossen wird die Rolle der beiden Buffer ausgetauscht. Durch dieses Prinzip wird gewährleistet, dass der Bildaufbau für den Betrachter nicht sichtbar ist.

3.2 Vertex- und Fragment-Shader

Auf Graphikkarten mit programmierbarer GPU können bestimmte Teile der Pipeline selbst programmiert werden. Zu diesem Zweck besitzen die Graphikkarten programmierbare Vertex- und Fragmentprozessoren, die die Vertex- und Fragmentberechnungen der Fixed Function Pipeline durch Shader ersetzen. In Abbildung 3.1 sind die Bereiche der Fixed Function-Pipeline, die umgangen werden können, rot markiert. Der Vertex-Shader muss die Operationen von der Modelview Transformation bis zur Projektion, der Fragment-Shader das Fragment-Processing übernehmen.
Im Vertexprozessor werden vor allem folgende Vertex-Operationen erledigt:

- Vertex Transformationen
- Normalen Transformationen und Normalisierung

[2] Das Shadingverfahren entscheidet darüber, wie Vertex-Farben auf die entsprechenden Polygone übertragen werden. Dies kann für jeden Punkt innerhalb des Polygons entweder durch Mittelung der Eckpunktfarbwerte (Flat-Shading), deren lineare Interpolation (Gouraud-Shading) oder aber unter Berücksichtigung der interpolierten Eckpunktnormalen (Phong-Shading) geschehen.

- Texturkoordinatengenerierung und -transformation
- Beleuchtungsberechnungen
- Material- und Farboperationen

Der Vertexprozessor ersetzt die Vertexberechnungen der FFP und daher müssen bei dessen Verwendung alle gewünschten Operationen Bestandteil des Vertex-Shaders sein, der als Programm auf dem Prozessor läuft. Es ist nicht möglich, für einige der Operationen die FFP Funktionalität zu nutzen und andere innerhalb eines Shaders zu realisieren. Es sind generell keine Vertexoperationen im Shader realisierbar, für die Topologieinformationen oder Informationen über mehr als einen Vertex erforderlich sind. Als zwingenden Output liefert der Shader den aktuellen Vertex in Clip-Koordinaten. Zusätzlich kann er beispielsweise Texturkoordinaten, Normalen oder Farben an den Fragment-Shader übergeben.
Alle diese Daten durchlaufen anschließend bis zur Rasterisierung wieder die FFP und dienen dem Fragment-Shader interpoliert als Input. Dieser wird, analog zum Vertexprozessor, mittels Fragment-Shader programmiert. Er übernimmt vor allem folgende Fragment-Operationen:

- Texturzugriff und Texturfilter
- Nebelberechnungen
- Operationen auf interpolierten Werten
- Fragment-Farbwertberechnungen

Der Fragmentprozessor kann weder die eigene Position des Fragmentes verändern, noch Operationen ausführen, für die Informationen weiterer Fragmente benötigt werden. Durch die letztere Beschränkung ist es möglich, dass mehrere Fragmentprozessoren parallel nebeneinander arbeiten.
Die vom Fragment-Shader berechneten Informationen werden abschließend den Fragment Operationen der Standard Pipeline unterzogen und gelangen als Pixel in den Framebuffer. Mit Hilfe von Vertex- und Fragment-Shadern ist es möglich die CPU zu entlasten, da Graphikberechnungen auf die GPU verlagert werden und so auf dem Hauptprozessor Ressourcen frei werden. Die Verwendung der Shader macht es allerdings gleichzeitig nötig, die Funktionalität der umgangenen Bereiche der FFP in Form von Shadern nachzubilden.

3.3 Photometrie und Radiometrie

Um die in der Computergraphik verwendeten Beleuchtungsmodelle besser zu verstehen, werden im Folgenden die Begriffe Radiometrie und Photometrie und deren physikalische Größen erklärt.

Radiometrie: Die Radiometrie ist die Wissenschaft, die sich mit der physikalischen Erfassung und Messung elektromagnetischer Energie, beziehungsweise Strahlung beschäftigt. Sie wird daher auch als Strahlungsphysik bezeichnet.

Photometrie: Die Photometrie stellt das Äquivalent zur Radiometrie aus physiologischer Sicht dar. Sie beschreibt die Wirkung des Lichts, also eines elektromagnetischen Spektrums, auf die visuellen Sinnesorgane des Menschen. Die Photometrie stellt daher die entscheidende Basis für die Thematik der Lichtsimulation dar.

Die beiden Wissenschaften sind folglich eng verwandt: Die eine misst die elektromagnetische Strahlung, die andere interpretiert diese im Hinblick auf die Wahrnehmung des Menschen. Zu jeder radiometrischen Größe existiert eine korrespondierende photometrische Größe. In welchem Zusammenhang stehen diese beiden Größen jedoch zueinander und wie ist es möglich, einen Wert in das jeweils andere System umzurechnen?
Dieser Zusammenhang kann mit Hilfe zweier empirischer Größen hergestellt werden: dem spektralen Hellempfindlichkeitsgrad und dem photometrischen Strahlungsäquivalent. Der spektrale Hellempfindlichkeitsgrad $V(\lambda)$ beschreibt die relative Empfindlichkeit des menschlichen Auges in Abhängigkeit von der Wellenlänge des Lichts. Die Empfindlichkeit des Auges ist jedoch abhängig von der Beleuchungssituation, die in einer unterschiedlichen Anzahl angeregter Stäbchen und Zapfen resultiert. Der spektrale Hellempfindlichkeitsgrad ist daher unterschiedlich für

photopisches (Tages)-Sehen:	$V(\lambda)$,
mesopisches (Dämmerungs)-Sehen:	$V_{eq}(\lambda)$,
skotopisches (Nacht)-Sehen:	$V'(\lambda)$.

In Abbildung 3.2 ist der spektrale Hellempfindlichkeitsgrad im photopischen Bereich dargestellt. Diese Kurve ist durch empirische Tests ermittelt und in DIN 5031 standardisiert.

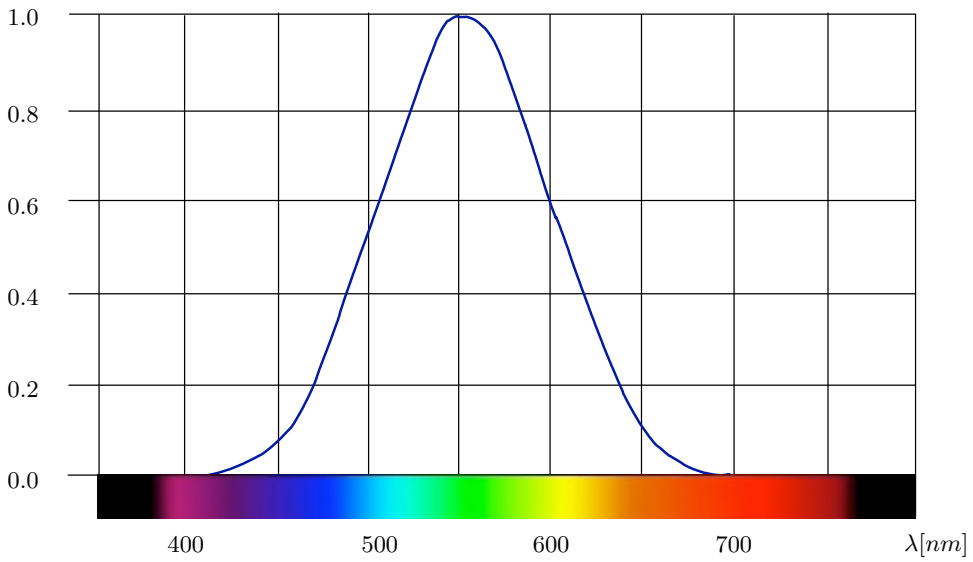


Abbildung 3.2: Spektraler Hellempfindlichkeitsgrad $V(\lambda)$

Das photometrische Strahlungsäquivalent K_m stellt den direkten Zusammenhang zwischen der Photometrie und der Radiometrie dar, indem es die strahlungsphysikalische Einheit Watt (W) ins Verhältnis zur lichttechnischen Größe Lumen (lm) setzt. Dadurch wird der Maximalwert der absoluten spektralen Empfindlichkeit ausgedrückt. Im photopischen Bereich ergibt sich dieser wie folgt:

$$K_m = 683\frac{lm}{W} \tag{3.1}$$

Mit Hilfe dieses photometrischen Strahlungsäquivalentes und des Hellempfindlichkeitsgrades, ist es nun möglich, aus einer beliebigen spektralen Größe $X(\lambda)$ ihr photometrisches Pendant X zu berechnen. Dazu wird über den vom menschlichen Auge wahrnehmbaren Wellenbereich des Lichts integriert und jeweils der spektrale Wert mit dem entsprechenden Hellempfindlichkeitsgrad gewichtet. Durch anschließende Multiplikation mit K_m erhält man, wie die Gleichung 3.2 zeigt, den photometrischen Wert X:

$$X = K_m \int_{380nm}^{780nm} X(\lambda) \cdot V(\lambda) d\lambda \tag{3.2}$$

Dieser Notation entsprechend wird eine spektrale Größe im Folgenden als mit der Hellempfindlichkeitskurve gewichtetes Pendant $X(\lambda)$ der photometrischen Größe X gekennzeichnet. Die Lichtmenge Q, eine photometrische Größe, berechnet sich daher wie folgt aus der spektralen Strahlungsmenge $Q(\lambda)$:

$$Q = 683\frac{lm}{W} \int_{380nm}^{780nm} Q(\lambda) \cdot V(\lambda) d\lambda \tag{3.3}$$

In Tabelle 3.1 sind die für die Photometrie relevanten Größen aufgelistet. Die entsprechenden Größen der Radiometrie stehen mit diesen, wie für die Strahlungsmenge gezeigt, gemäß Gleichung 3.2 im Verhältnis.
Die Berechnung der Werte erfolgt für kleinste Intervalle der Funktionen, für die die photometrische Größe als konstant angenommen werden kann. Solche als infinitesimal bezeichneten Werte sind mit d gekennzeichnet, dA beschreibt beispielsweise ein infinitesimal kleines Flächenelement.

	Einheit	Berechnung	spektrales Pendant	Einh.
Q Lichtmenge	$lm \cdot sec$	$Q = 683\frac{lm}{W}\int_{380nm}^{780nm} Q(\lambda) \cdot V(\lambda) d\lambda$	Strahlungsmenge	J
	Die Lichtmenge entspricht dem $V(\lambda)$ gewichteten gesamten Energieverlust (gemessen in Joule) einer Lichtquelle.			
Φ Lichtstrom	lm	$\Phi = \frac{dQ}{dt}$	Strahlungsfluß	W
	Der Lichtstrom beschreibt die Lichtleistung einer Lichtquelle in Lumen und entspricht der spektralen Strahlungsleistung in Watt.			
I Lichtstärke	$cd = \frac{lm}{sr}$	$I = \frac{d\Phi}{d\omega}$	Strahlstärke	$\frac{W}{sr}$
	Die Lichtstärke entspricht der ausgestrahlten Lichtstärke (in Candela) pro durchstrahltem Raumwinkel (in Steradiant).			
L Leuchtdichte	$\frac{cd}{m^2}$	$L = \frac{dI}{dA \cdot \cos\theta} = \frac{d^2\Phi}{dA \cdot \cos\theta \cdot d\omega}$	Strahldichte	$\frac{W}{m^2 sr}$
	Die Leuchtdichte beschreibt das Verhältnis zwischen der Lichtstärke und der Größe der leuchtenden Fläche. Dieses Verhältnis wird vom menschlichen Auge als Helligkeit wahrgenommen.			
E Beleuchtungsstärke	$lx = \frac{lm}{m^2}$	$E = \frac{d\Phi}{dA_e}$	Bestrahlungsstärke	$\frac{W}{m^2}$
	Die Beleuchtungsstärke ist eine reine Empfängergröße und beschreibt den einfallenden Lichtstrom pro Empfängerfläche (in Lux).			
B spez. Lichtausstrahlung	$lx = \frac{lm}{m^2}$	$B = \frac{d\Phi}{dA_s}$	spezifische Strahlungsemission	$\frac{W}{m^2}$
	Diese Größe stellt das Gegenstück zu E dar und repräsentiert den abgegebenen Lichtstrom pro Flächenelement der leuchtenden Fläche.			
ω Raumwinkel	$Steradiant$	$\omega = \frac{A_K}{r^2}$; $d\omega = \frac{dA \cdot \cos\alpha}{d^2} = \sin\theta \cdot d\theta \cdot d\phi$		
	Der Raumwinkel ist die räumliche Erweiterung des Bogenmaßes, der durch das Verhältnis zwischen einer bestimmten Fläche der Kugeloberfläche und dem quadrierten Kugelradius definiert ist (in Steradiant).			

Tabelle 3.1: Photometrische Größen

3.4 Beleuchtungsmodelle

Mit einem Beleuchtungsmodell wird die Interaktion zwischen verschiedenen Lichtquellen, Materialien und Geometrien berechnet. Die Berechnung erfolgt dabei, wie in Kapitel 3.1 beschrieben, meist für jeden Vertex (*per-vertex*), kann aber auch *per-fragment* während der Fragment-Operationen stattfinden.
Im Falle der per-vertex Beleuchtungsberechnung sorgt das Shading dafür, dass die aus der Beleuchtungsberechnung resultierenden Farben während der Rasterisierung interpoliert werden.
Entscheidend für die Beleuchtung einer Szene ist vor allem die Art der Lichtquelle. Im Bereich der Computergraphik unterscheidet man zwischen drei verschiedenen Lichtquellen (siehe Abbildung 3.3):

gerichtetes Licht: Eine unendlich weit entfernte Lichtquelle sendet Lichtstrahlen aus, die durch die große Distanz annähernd parallel auf ein Objekt der Szene treffen. Das Sonnenlicht wird beispielsweise durch gerichtetes Licht dargestellt. Gerichtetes Licht verursacht weiche Schattengrenzen.

Punktlicht: Beim Punktlicht handelt es sich um eine positionsabhängige Beleuchtung von einem Punkt der Szene aus. Von diesem werden Lichtstrahlen gleichmäßig in alle Richtungen versendet. Die Lichtquelle wird nur durch einen einzelnen Punkt und nicht wie in der Realität durch eine Fläche repräsentiert, daher entstehen nur harte Schattenkanten.

Spotlicht: Das Spotlicht besitzt die gleichen Eigenschaften wie eine Punktlichtquelle, mit der Ausnahme, dass der Punkt Lichtstrahlen nur in einen bestimmten Bereich aussendet. Dieser Bereich ist von einem Richtungsvektor und einem Öffnungswinkel abhängig.

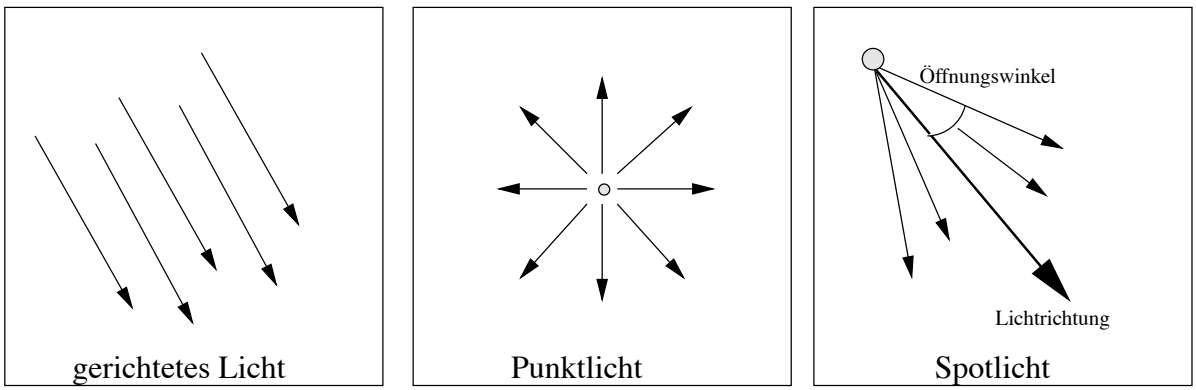


Abbildung 3.3: Drei verschiedene Lichtquellentypen

Zusätzlich zu den Lichtquellen hat die Oberflächenbeschaffenheit der Objekte, also sowohl ihre Reflexionseigenschaft als auch ihre Textur, Einfluss auf die Effekte der Beleuchtung. Mit einem Beleuchtungsmodell wird das einfallende Licht an einem Punkt der Oberfläche ermittelt, sodass

unter Berücksichtigung der materialspezifischen Reflexionseigenschaften daraus das ausfallende Licht berechnet werden kann, das die Kamera (bzw. den Betrachter) erreicht. So kann ermittelt werden, wie hell der Punkt innerhalb der Szene erscheint. Die Beleuchtung wird entweder lokal oder global bestimmt. Bei lokalen Beleuchtungsmodellen geht nur das direkt von den Lichtquellen einfallende Licht in die Berechnung mit ein. Dagegen wird bei globalen Modelle auch die Lichtinteraktion zwischen Objekten der Umgebung berücksichtigt.

3.4.1 Lokale Beleuchtungsmodelle

Lokale Beleuchtungsmodelle beschreiben die Effekte des direkt einfallenden Lichts auf einen Punkt der Szene. Indirekte Beleuchtung, spekulare Effekte (Spiegelungen) und Kaustiken (Spiegelungen über mehrere spekulare Oberflächen hinweg) werden nicht berücksichtigt. Ein Beleuchtungsmodell setzt sich meist aus einem *ambienten*, einem *diffusen* und einem *spekularen* Term zusammen, durch die das Beleuchtungsverhalten auf verschiedenen Oberflächen dargestellt werden:

$$L = ambient + diffus + specular \tag{3.4}$$

Auch wenn diese Gleichung eine sehr abstrakte Darstellung der Beleuchtung darstellt, die nicht viel mit den physikalischen Eigenschaften des Lichts zu tun hat, liefert dieses Komponentenmodell bereits gute Ergebnisse und ist einfach zu realisieren. Im Folgenden wird ein einfaches lokales Beleuchtungsmodell beschrieben, das dieser Aufteilung in die drei Komponenten entspricht. Zur besseren Unterscheidung werden RGB-Farbvektoren als Großbuchstaben und Richtungsvektoren als Kleinbuchstaben notiert. Zu beachten ist dabei, dass die RGB-Farbvektoren jeweils komponentenweise multipliziert werden. Abbildung 3.4 stellt die im Nachfolgenden verwendeten Vektoren- und Winkelbezeichnungen dar.

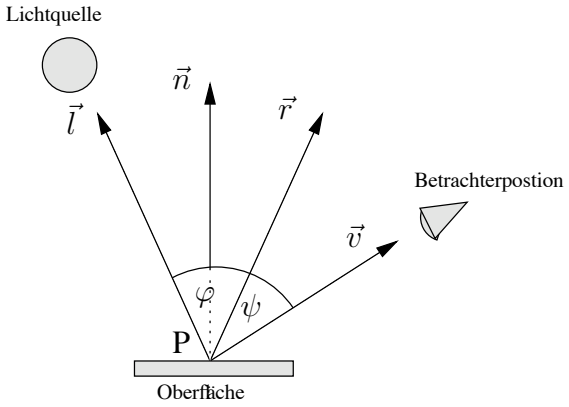


Abbildung 3.4: Die Richtungsvektoren der Beleuchtungsmodelle

Der diffuse Term des Beleuchtungsmodells beschreibt die Interaktion zwischen Licht und Oberfläche völlig diffuser (matter) Materialien. Solche Oberflächen wie beispielsweise Holz oder Kreide reflektieren das Licht einer Lichtquelle gleichmäßig in alle Richtungen. Eine veränderte Blickrichtung des Betrachters ändert somit nichts an der Farberscheinung der Oberfläche. Die diffuse Reflexion folgt dem *Lambert'schen Gesetz* (vgl. [Ban99], S. 37f.) und wird daher in der Literatur teilweise auch als *Lambert'sches Beleuchtungsmodell* bezeichnet. Der diffuse Term ist der einzige Teil des hier vorgestellten lokalen Beleuchtungsmodells, der einen physikalischen Hintergrund hat. Das reflektierte Licht einer ideal diffusen Oberfläche ist abhängig vom Kosinus des Winkels φ zwischen der Oberflächennormalen $\vec{n}$ des Punktes P und dem Vektor $\vec{l}$, der von P in Richtung Lichtquelle zeigt, beziehungsweise von deren Skalarprodukt falls $\vec{n}$ und $\vec{l}$ normiert sind. Unter Berücksichtigung der diffusen Materialfarbe $\vec{M}_{diff}$ und der einstrahlenden Lichtfarbe $\vec{L}$ ergibt sich damit für die Berechnung der diffusen Lichtfarbe:

$$diffus = \cos\varphi \, \vec{M}_{diff} \, \vec{L} \tag{3.5}$$

$$= (\vec{n} \cdot \vec{l}) \, \vec{M}_{diff} \, \vec{L} \tag{3.6}$$

Der größte Nachteil dieser Berechnung ist, dass alle nicht direkt beleuchteten Flächen schwarz bleiben und alle Objekte, egal wie weit sie von der Lichtquelle entfernt sind, gleich stark beleuchtet werden.
Zur Vermeidung schwarzer Flächen wird deshalb ein ambienter Term in das Beleuchtungsmodell aufgenommen. Dieser sorgt für eine Basisbeleuchtung, indem auf jeden Punkt der diffus beleuchteten Szene eine für alle Objekte konstante ambiente Beleuchtung addiert wird. Die Farbe des ambienten Lichts $\vec{L}_{amb}$ wird komponentenweise mit der ambienten Materialfarbe $\vec{M}_{amb}$ multipliziert:

$$ambient = \vec{M}_{amb} \, \vec{L}_{amb} \tag{3.7}$$

Um zusätzlich die fehlende Lichtabnahme in die Szene zu integrieren, wird das Beleuchtungsmodell um eine Abstandsfunktion $f(d)$ erweitert. Das erweiterte Beleuchtungsmodell besteht daher aus der Summe des ambienten und des durch die Abstandsfunktion gewichteten diffusen Term:

$$\vec{L} = ambient + f(d) diffus \tag{3.8}$$

Bisher ist in dieser Form nur die diffuse Reflexion realisiert. Die Integration des spekularen Terms bewirkt die Berücksichtigung der Blickrichtung des Betrachters. Durch die daraus resultierenden Glanzlichter wirkt die Erscheinung der Objekte um ein Vielfaches realistischer. Diese Erweiterung des Modells durch *Bui Tuong Phong* [Pho75] führte zu dem Namen des Phong'schen Beleuchtungsmodells, nicht zu verwechseln mit dem Phong-Shading (vgl. Seite 9). Im Gegensatz zur diffusen Reflexion ist der spekulare Term abhängig vom Blickpunkt des Betrachters, also dem *View Vektor* $\vec{v}$. Das Phong'sche Beleuchtungsmodell ist kein physikalisches, sondern ein empirisches Modell, das die Erscheinung eines visuellen Phänomens durch eine bestimmte Funktion approximiert. Dies geschieht im spekularen Term durch die Berechnung des Kosinus zwischen dem Vektor der idealen Reflexion $\vec{r}$ und dem View Vektor unter Berücksichtigung eines Materialkoeffizienten n, mit dem die Schärfe des Glanzlichts reguliert werden kann.

Die Berechnung des Kosinus kann bei normierten Vektoren wiederum durch das Skalarprodukt erfolgen. Dem diffusen und ambienten Term entsprechend, existiert auch eine spezielle spekulare Materialfarbe $\vec{M}_{spec}$.

$$specular = cos^n \psi \vec{M}_{spec} \vec{L} \tag{3.9}$$
$$= (\vec{v} \cdot \vec{r})^n \vec{M}_{spec} \vec{L} \tag{3.10}$$

Der spekulare Term (3.10) kann nun mit dem diffusen und dem ambienten zum Phong'schen Beleuchtungsmodell zusammengesetzt werden. Dieses wird im Folgenden gleichzeitig für mehrere Lichtquellen erweitert:

$$\vec{L} = ambient + f(d)(diffus + specular) \tag{3.11}$$
$$= \vec{M}_{amb}\, \vec{L}_{amb} + \sum_{i=1}^{n} f(d_i) \vec{L}_i ((\vec{n} \cdot \vec{l}_i) \vec{M}_{diff} + (\vec{v} \cdot \vec{r}_i)^n \vec{M}_{spec}) \tag{3.12}$$

Das Phong'sche Modell wird beispielsweise im *Raytracing* (siehe Kapitel 3.4.2) als lokale Beleuchtungskomponente eingesetzt und findet, als *Blinn-Phong Modell*, in leicht modifizierter Form Einsatz als Standardbeleuchtung von *OpenGL*.

3.4.2 Globale Beleuchtungsmodelle

Globale Beleuchtungsmodelle berücksichtigen im Gegensatz zu den lokalen Modellen bei der Berechnung der Beleuchtung für einen Punkt auch das Licht, das diesen Punkt durch Lichtinteraktion zwischen Objekten der Umgebung erreicht. Dadurch können abhängig vom jeweiligen Verfahren indirekte Beleuchtung, Schatten, spekulare Effekte und teilweise Kaustiken (Spiegelungen über mehrere spekulare Flächen) realisiert werden. Damit stellen die globalen Beleuchtungsmodelle die realistischsten Verfahren dar und sind im Hinblick auf ihre visuelle Qualität für alle Computergraphikanwendungen und so auch für die Fahrsimulation das Wunschverfahren. Leider ist der Berechnungsaufwand sehr hoch, woran ihr Einsatz in Echtzeitsystemen oft scheitert. In diesem Kapitel wird das Prinzip einiger Verfahren kurz verdeutlicht. Dies dient einer groben Orientierung, um die Bewertung der globalen Beleuchtungsverfahren aus Sicht der Fahrsimulation besser verstehen zu können.

Raytracing

Die Idee des Raytracings („Strahlverfolgung“) ist die Verfolgung der von der Lichtquelle ausgesendeten Lichtstrahlen bis hin zum Auge. Da das Auftreffen an der Betrachterposition sehr unwahrscheinlich ist, wird dieser Vorgang umgekehrt. Dank der Helmholtz Reziprozität[3] kann man auch die Strahlen vom Auge aus zur Lichtquelle verfolgen. Dazu wird in einer Schleife über alle Pixel der Bildebene durch das jeweils aktuelle Pixel mindestens ein Strahl geschickt und auf Schnittpunkte in der Szene untersucht. Am Schnittpunkt wird die Leuchtdichte durch ein lokales Beleuchtungsmodell berechnet und anschließend werden jeweils der gebrochene (transmittierte)

[3] Die Helmholtz Reziprozität beschreibt die Umkehrbarkeit des Lichtweges.

und der ideal reflektierte Strahl weiterverfolgt. Dieser Prozess wird solange rekursiv wiederholt, bis eine Lichtquelle getroffen wird oder eine andere Abbruchbedingung greift. Die Beleuchtungswerte der verschiedenen Schnittpunkte werden addiert und bestimmen die Farbe des Pixels. Das Raytracing ist somit ein pixelbasiertes Beleuchtungsmodell und kann gleichzeitig auch als Shadingtechnik angesehen werden. Gleichermaßen werden die Licht- und Reflexionsberechnung wie auch die Farbwertbestimmung der einzelnen Pixel vom Raytracing Verfahren erledigt.
Als lokales Beleuchtungsmodell kommt das in Kapitel 3.4.1 beschriebene Phong'sche Beleuchtungsmodell zum Einsatz. Die Schattenberechnung kann direkt in das Modell integriert werden. Dazu muss die Gleichung 3.12 um die Terme $\vec{L}_{trans}$ und $\vec{L}_{refl}$ erweitert werden. Zu jeder Lichtquelle wird außerdem ein Schattenstrahl („Schattenfühler") gesendet. Befindet sich kein Schnittpunkt (zumindest keiner eines undurchsichtigen Objektes) zwischen dem Punkt und der Lichtquelle, so trägt diese zur direkten Beleuchtung bei. Dies geht in die Gleichung mit ein, indem $\vec{L}_i$ mit einem $\delta_i = 1$ gewichtet wird. Liegt der Punkt im Schattenbereich der Lichtquelle i wird entsprechend $\delta_i = 0$ gesetzt:

$$\vec{L} = ambient + f(d)(diffus + specular) \tag{3.13}$$

$$= \vec{M}_{amb}\,\vec{L}_{amb} + \sum_{i=1}^{n} f(d_i)((\vec{n}\cdot\vec{l}_i)\vec{M}_{diff} + (\vec{v}\cdot\vec{r}_i)^n\vec{M}_{spec})\vec{L}_i\delta_i + \vec{L}_{trans} + \vec{L}_{refl} \tag{3.14}$$

Durch die Berücksichtigung von Transparenzen und Schattenberechnungen ist das Raytracing sehr viel realistischer als lokale Beleuchtungsmodelle. Demgegenüber steht ein hoher Rechenaufwand, der vor allem durch die Schnittpunktberechnungen entsteht. Obwohl es sich beim Raytracing um ein globales Beleuchtungsmodell handelt, können indirekte Beleuchtungseffekte durch die Szene nicht realisiert werden. Die Umgebung geht aber beispielsweise in Form von schattenwerfenden Objekten in das Verfahren mit ein. Nachteilig wirkt sich auch aus, dass Szenen bezüglich eines festen Blickpunktes berechnet werden, was den Einsatz in Echtzeitsystemen zusätzlich erschwert. Das Raytracing Verfahren wurde in mehreren Variationen weiterentwickelt. Besonders stochastische Raytracing Varianten kommen häufig zum Einsatz. Durch Sampling werden dabei mehrere Strahlen durch ein Pixel gesendet (*Pixel Jiitering*) und abhängig vom Verfahren werden am Schnittpunkt unterschiedlich viele Strahlen weiterverfolgt. Durch dieses Vorgehen wird die Pixelfarbe von mehreren Strahlen bestimmt, was weniger Aliasingeffekte und weichere Schattengrenzen zur Folge hat.

Radiosity

Das Prinzip des Radiosity-Beleuchtungsmodells ist, alle Flächen einer Szene als Lichtquellen anzusehen, egal ob es sich um echte Lichtquellen oder normale Oberflächenpolygone handelt. Jede dieser Flächen strahlt emittiertes und reflektiertes Licht in die Szene und kann somit zum Sender werden.
Aufwendig ist bei diesem Verfahren die Berechnung der sogenannten Formfaktoren. Sie bezeichnen den Anteil der versendeten Strahlung eines Senders s, der den Empfänger e erreicht. Der Formfaktor ist nur von der Geometrie der beiden Flächen abhängig. Da er aus einem doppelten Integral besteht, dessen Lösung sehr komplex ist, wird er meist nur angenähert. Für jedes

Polygon wird beim Radiosity Verfahren eine Gleichung aufgestellt, die den Strahlungsaustausch zwischen diesem Polygon als Empfänger und allen sendenden Flächen beschreibt:

$$B_e = E_e + \rho_e \sum_{s=1} B_s \cdot F_{es} \tag{3.15}$$

Für jede Empfängerfläche wird ein spektraler Reflexionsgrad ρ ermittelt, der die Reflexionseigenschaften des Materials widerspiegelt. Er wird zur Gewichtung des für alle Sender aufsummierten Produktes ihrer Radiosity B_s und dem Formfaktor zwischen Empfänger und Sender F_{es} benutzt. Das Ergebnis wird auf die Beleuchtungsstärke E des Empfängers addiert und ergibt damit dessen Radiositywert.
Da für jede Fläche der Szene eine solche Gleichung erstellt wird, entsteht ein großes Gleichungssystem, dessen Lösung sehr komplex ist. Aus diesem Grund nähern die meisten Radiosity Verfahren die Lösung an, um sie anschließend sukzessive zu verfeinern. Die bekanntesten Radiosity-Verfahren sind die Iterationsverfahren (*Jakobi*, *Gauss-Seidel*), das Relaxationsverfahren (*progressive Refinement*) und das *hierarchische Radiosity* (vgl. [CW93]).
Im Gegensatz zum Raytracing liefert das Radiosity Verfahren eine gute globale Lichtsimulation, weil die Lichtabstrahlung aller Flächen berücksichtigt wird. Ein großer Nachteil aber ist dessen Geometrieabhängigkeit und die schlechtere direkte Beleuchtung, die es beispielsweise schwierig macht, harte Schattenkanten akkurat zu repräsentieren. Da Radiosity von einer diffusen Umgebung ausgeht, ist es außerdem nicht möglich spekulare Effekte darzustellen.

Weiterentwicklungen und Variationen

Ein Verfahren, das sich gleichzeitig die Vorteile von Raytracing und Radiosity zunutze macht, ist das *Final Gathering*. Dabei erfolgt die Beleuchtung der Szene zunächst durch das Radiosity, wobei das direkte Licht nur für eine grobe Patchauflösung[4], das indirekte aber bis zur Konvergenz berechnet wird. Anschließend wird das direkte Licht durch das Verschießen negativer Radiosity wieder entfernt, und die Szene wird durch Raytracing dargestellt. An jedem Schnittpunkt findet eine Interpolation der Radiositywerte statt und die Lichtquellen werden mit vielen Schattenfühlern explizit abgetastet. Durch diese Kombination wird eine gute direkte und indirekte Lichtsimulation erreicht, obgleich Kaustiken mit diesem Verfahren auch nicht darstellbar sind. Nachteilig bleibt, dass das Final Gathering durch die Verwendung des geometrieabhängigen Radiosityverfahrens ebenfalls abhängig vom verwendeten Mesh ist. Zusätzlich führt die Kombination der beiden Modelle zu einem erhöhten Berechnungsaufwand.
Das von *Henrik Wann Jensen* entwickelte *Photon Mapping* [WJ01] besteht ebenso aus zwei kombinierten Beleuchtungsschritten. Im ersten Pass des Algorithmus, dem *Particle Tracing*, senden alle Lichtemitter Photonen aus. Treffen diese auf eine Oberfläche, werden sie abhängig von deren Reflexionseigenschaften in eine bestimmte Richtung reflektiert oder gebrochen. Alle Photonen, die auf eine diffuse Oberfläche treffen, werden dort in einer *Photonen Map* gespeichert. Im zweiten Pass wird ein Raytracer auf die Szene angewandt. Die per Raytracing ermittlete direkte Beleuchtung wird an jedem Schnittpunk auf die, mittels *Radiance Estimate* aus der Photonen

[4]Ein Patch bezeichnet in diesem Zusammenhang ein kleines Flächenelement, für das die Radiosity berechnet wird.

Map berechnete, indirekte Beleuchtung summiert. Das Photon Mapping kann im Gegensatz zu den anderen vorgestellten Modellen alle denkbaren visuellen Beleuchtungseffekte, wie beispielsweise Kaustiken und Beleuchtung in Zusammenhang mit durchsichtigen Materialien, qualitativ hochwertig berechnen. Die eigentlich lange Berechnungszeit kann durch verschiedene Optimierungen verkürzt werden, sodass dieses Verfahren beim Rendern von statischen Szenen immer mehr Verbreitung auch in kommerziellen Produkten findet.
Für vertiefende Informationen zu den globalen Beleuchtungsmodellen sei auf [Shi03, DBB03, SP94] verwiesen.

3.5 Schattenverfahren

Neben der Lichtsimulation berücksichtigt eine komplette Beleuchtungssimulation auch die Schattenberechnung und -darstellung. Im Folgenden werden daher die gängigsten Schattenverfahren erläutert.

Planar Shadows

Ein recht einfaches Verfahren zur Schattenberechnung stellen die planaren Schatten dar. Das Prinzip dieses Algorithmus liegt darin, die Polygone eines Objektes mit einer Projektionsmatrix auf eine Ebene der Szene zu projizieren und dadurch die Schattenregion zu ermitteln. Diese kann danach mit der Schattenfarbe gerendert werden. In einem zweiten Pass wird die restliche Szene gerendert.
Bei diesem Verfahren muss zunächst die Projektionsmatrix ermittelt werden, mit der die Projektion der Vertices eines Objektes auf die Ebene erfolgt. Die Art dieser Matrix ist abhängig von der verwendeten Lichtquelle. Bei gerichteten Lichtquellen kommt entsprechend eine parallele Projektion, bei Punktlichtern eine perspektivische Projektion zum Einsatzt. Zur Vereinfachung wird der Scheinwerfer hier als Punktlichtquelle angesehen. Ausgehend von der Position $\vec{l}$ dieser Lichtquelle muss die Projektion $\vec{p}$ des Punktes $\vec{o}$ der Schattenquelle (Occluder) auf die Empfängerebene E berechnet werden (siehe Abbildung 3.5). Die Projektion geschieht durch die in Gleichung 3.16 dargestellte Matrix. Die Matrix kann auf Basis des zweiten Strahlensatzes und der perspektivischen Projektionsmatrix hergeleitet werden (siehe Referenz [AMH02]). Mit Hilfe dieser Matrix erfolgt die Projektion des Objektes auf die Ebene, sodass anschließend die Schattenregion auf der Empfängerebene in Schattenfarbe gerendert werden kann.

$$\begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} n \cdot l + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & n \cdot l + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & n \cdot l + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & n \cdot l \end{pmatrix} \cdot \begin{pmatrix} o_x \\ o_y \\ o_z \end{pmatrix} \tag{3.16}$$

Im zweiten Pass wird der Rest der Szene gerendert. Der Schattenwurf eines Objektes kann jedoch mit der verwendeten Matrix nur für eine Ebene berechnet werden. Durch diese Tatsache

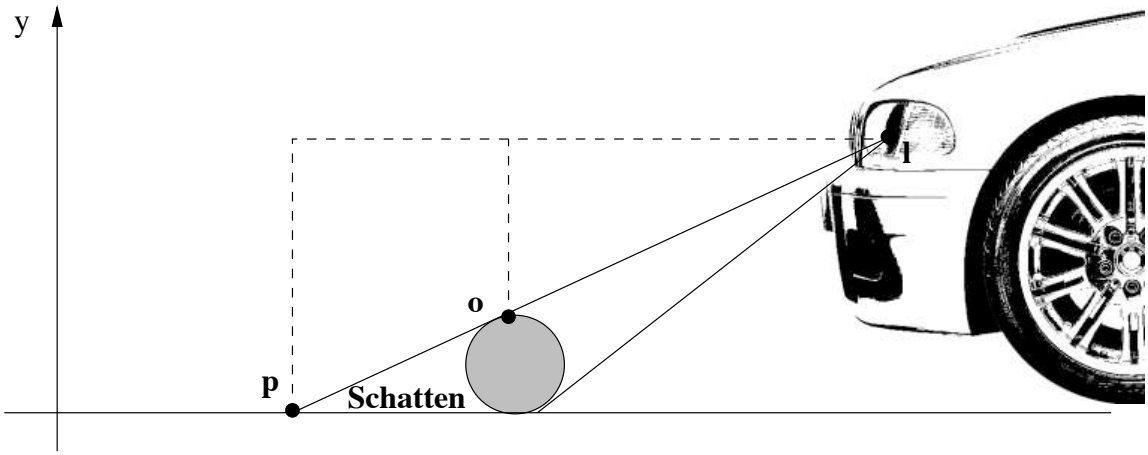


Abbildung 3.5: Schemadarstellung: planare Schatten

entsteht ein weiteres Problem:
Da der Schatten immer auf eine Ebene projiziert wird, kann es zu sogenannten Clipping-Fehlern kommen. Dabei reicht der projizierte Schatten über die Empfängerfläche[5] hinaus. Bei der Lösung dieses Problems hilft der Stencilbuffer: Um zu verhindern, dass der Schatten außerhalb der Empfängerfläche dargestellt wird, wird diese im Stencilbuffer markiert und der Schatten nur dort gerendert.
Verschiedene Erweiterungen machen es möglich, mit dem planaren Shadow-Algorithmus auch weiche Schattengrenzen zu erzeugen. Eine Betrachtung verschiedener Verfahren ist in Referenz [AMH02](Seite 254) zu finden.

Shadow Volume

Das Shadow Volume Verfahren beschreibt einen geometriebezogenen Algorithmus, der den Schattenwurf auf beliebige Objekte bestimmen kann. Er wurde erstmals von Franclin C. Crow [Cro77] vorgestellt und schließlich von Tim Heidmann [Hei91] um die Ausnutzung des Stencilbuffers erweitert.
Das Prinzip dieses Verfahrens liegt darin, ausgehend von einer Punktlichtquelle, die Lichtstrahlen durch die Vertices der einzelnen Objekte hindurch zu verlängern. Dadurch entsteht für jedes einzelne Objekt ein Schattenvolumen (siehe Abbildung 3.6), das von der Objekt-Silhouette aufgespannt wird. Alle Vertices innerhalb dieses Volumens befinden sich im Schatten.

[5] Von entscheidender Bedeutung ist hier der Unterschied zwischen Ebene und Fläche: Eine Ebene teilt einen Raum in zwei Halbräume auf und hat eine unendliche Ausdehnung. Die Fläche hingegen ist ein in zwei Dimensionen ausgedehnter Teil einer Ebene.

Der Ablauf des Algorithmus ist folgender:

1. Die Schattenvolumen werden bestimmt
2. Die Szene wird mit ambienten und emissiven Komponenten gerendert
3. Pro Pixel wird mittels Stencilbuffer entschieden, ob es im Schattenvolumen liegt
4. Die Szene wird mit diffusen und spekularen Komponenten gerendert während der Stencilbuffer die Schattenregionen ausmaskiert

Zunächst müssen die Silhouetten bestimmt werden, um die Schattenvolumen zu erzeugen. Das in Abbildung 3.6 gezeigte Beispiel stellt ein einfaches Quadrat und dessen Schattenvolumen dar. Meist sind die Objekte komplexer und das Schattenvolumen weniger leicht zu bestimmen. Zur Ermittlung der Objekt-Silhouette[6] kann man folgendermaßen vorgehen:
Zunächst müssen die Kanten lokalisiert werden, die die Silhouette bilden. Dazu werden für jede Kante die benachbarten Dreiecke bestimmt. Zwei Dreiecke gelten dabei als benachbart, wenn sie zwei gemeinsame Vertices besitzen. Eine Kante ist genau dann Teil der Silhouette, wenn sie ein beleuchtetes von einem nicht beleuchteten Dreieck trennt. Ob ein Dreieck des Objektes von der Lichtquelle beleuchtet wird oder nicht, hängt von dessen Sichtbarkeit s aus Richtung der Lichtquelle ab. Diese lässt sich durch das Skalarprodukt zwischen der Oberflächennormale $\vec{n}$ des Dreiecks und dem Lichtvektor $\vec{l}$ berechnen. Ist $s > 0$, dann ist das Dreieck der Lichtquelle zugewandt. Sobald die Silhouettenkanten auf diese Weise ermittelt sind, ergibt sich daraus das Schattenvolumen des Objektes: Jeder Punkt der Kanten wird aus Sicht der Lichtquelle, entlang des Strahls von der Lichtquelle durch diesen Punkt, ins Unendliche oder, bei gewünschter Eingrenzung des Einflusses der Lichtquelle, auf die Entfernung d projiziert. Der projizierte Punkt P' berechnet sich aus dem Punkt P der Silhouettenkante wie folgt:

$$\vec{P'} = \frac{\vec{P} + d(\vec{P} - \vec{L})}{|(\vec{P} - \vec{L})|} \tag{3.17}$$

Das komplette Schattenvolumen ergibt sich aus der Berechnung der einzelnen Punkte.
Die Entscheidung, ob sich ein Pixel innerhalb eines Schattenvolumens befindet oder nicht, kann sehr effizient mit Hilfe des Stencilbuffers getroffen werden. Zunächst wird dieser gelöscht und die komplette Szene wird mit ambienten und emissiven Komponenten in den Framebuffer gerendert. Anschließend werden die Frontfacing-Polygone des Schattenvolumens bei deaktivierten z- und Framebufferupdates gerendert. Jedes Frontfacing Polygon erhöht den Wert des aktuellen Pixels im Stencilbuffer, vorausgesetzt die Pixel des Schattenvolumens sind sichtbar. Entsprechend wird mit den Backfacing-Polygonen verfahren, mit dem Unterschied, dass ein Backfacing-Polygon den Wert innerhalb des Stencilbuffers dekremiert. Schließlich wird die Szene mit den diffusen und spekularen Komponenten ein weiteres Mal gerendert. Dabei dient der Stencilbuffer als Maske, sodass nur die Pixel berücksichtigt werden, deren Wert im Sterncilbuffer größer 0 ist. Durch dieses Vorgehen erhalten Pixel innerhalb der Schattenvolumen kein diffuses oder spekulares Licht.

[6] Es wird dabei von einem Objekt ausgegangen, das aus einem Dreiecks-Mesh besteht.

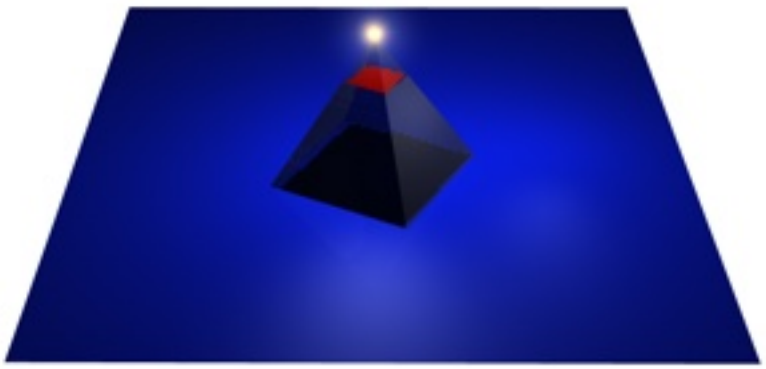


Abbildung 3.6: Durch eine Lichtquelle erzeugtes Schattenvolumen

Shadow Mapping

Der Shadow Mapping Algorithmus ist ein klassischer 2-Pass Algorithmus, der erstmals 1978 von Lance Williams [Wil78] vorgestellt wurde und Schatten auf beliebigen Oberflächen generieren kann. Im ersten Pass des Algorithmus wird die Szene aus Sicht der Lichtquelle gerendert, um eine Shadow Map zu erhalten, die während des zweiten Renderingpasses zur Bestimmung der Schattenregionen dient.
Im Gegensatz zu den planaren Schatten wird beim Shadow Mapping ein Spot-Licht vorausgesetzt. Wie in Kapitel 4.1 beschrieben, besitzt eine solche Lichtquelle die Parameter Position, Richtung und Öffnungswinkel. Damit wird ein Frustum aufgespannt, das beim Shadow Mapping zum Rendern aus Lichtquellensicht dient. Die in Relation zur Lichtquellenposition stehenden Tiefenwerte (z-Werte) der Szene werden dabei mit dem z-Buffer Algorithmus gerendert. Als Resultat enthält der z-Buffer für jedes Pixel den z-Wert des Objektes, das der Lichtquelle am nächsten liegt. Der Inhalt des z-Buffers wird auch als Shadow Map oder Depth Map bezeichnet und kann auch als separate Textur gespeichert werden.
Nach Erzeugung der Shadow Map, wird die Szene ein zweites Mal, nun aber aus Kamerasicht, gerendert. Für jedes Pixel wird anhand der Shadow Map entschieden, ob der aktuelle Pixel im Schattenbereich liegt oder nicht. Dazu wird die Distanz des Pixels zur Lichtquelle mit dem entsprechenden z-Wert der Shadow Map verglichen. Ist diese Distanz größer als der in der Shadow Map eingetragene Wert, so liegt der Pixel aus Sicht der Lichtquelle hinter dem Objekt das der Lichtquelle am nächsten ist. Das Pixel wird demzufolge in Schattenfarbe gerendert. Entsprechend den Shadow Volumes wird auch durch das Shadow Mapping lediglich der Kernschatten generiert. Um weiche Schattengrenzen (Penumbraregionen) zu erhalten, muss das Verfahren erweitert werden (vgl. [HLHS03]).

Kapitel 4

Echtzeitfähige Beleuchtungssimulation

Als Voraussetzung für die Entwicklung eines Konzeptes für die Beleuchtungssimulation im BMW-Fahrsimulator, werden in diesem Kapitel zunächst verschiedene bestehende Verfahren zur Lichtsimulation recherchiert. Zusätzlich zu dieser Übersicht unterschiedlicher Lösungsansätze wird die Beleuchtungssituation auch aus photometrischer Sicht betrachtet, um die Realisierung, soweit möglich und gewünscht, an ein reales Beleuchtungsmodell anlehnen zu können.
Da auch der Schatten einen Teil der Beleuchtungssimulation darstellt, werden im Anschluss daran gängige Schattensimulationsverfahren vorgestellt, die zusammen mit den Erkenntnissen der Recherche über Beleuchtungssimulationen in die abschließende Bewertung eingehen.

4.1 Recherche bestehender Verfahren

Im Vordergrund der Recherche stehen die verschiedenen Möglichkeiten, eine dynamische Szene mit dynamischen Lichtquellen, wie sie in der Fahrsimulation zu finden sind, in Echtzeit zu beleuchten. Der Fokus liegt dabei auf der Simulation der Beleuchtung durch Fahrzeugfrontscheinwerfer.

4.1.1 *OpenGL* Standard Beleuchtung

In *OpenGL* wird standardmäßig ein lokales Beleuchtungsmodell eingesetzt. Wie in Kapitel 3.4.1 beschrieben, handelt es sich dabei um das *Blinn-Phong Modell*. Dementsprechend basiert das Modell auf der Annahme, dass das Licht aus einem ambienten, einem diffusen und einem spekularen Anteil besteht. Durch diese Komponenten wird die Beleuchtung angenähert, indirekte Beleuchtung und Schatten werden standardmäßig nicht dargestellt. Die Lichtquellen können entweder positionsunabhängig sein und somit gerichtetes Licht abstrahlen, oder aber positionsabhängig sein und damit Punktlichtquellen oder Spotlichter darstellen. Welche der drei Typen eine *OpenGL*-Lichtquelle (`glLight()`) darstellt, wird durch ihre Parameter bestimmt. Standardmäßig handelt es sich um gerichtetes Licht, das durch Angabe einer Position mittels `GL_POSITION`

Parameter[1] zur Punktlichtquelle und durch Angabe einer Lichtrichtung (`GL_SPOT_DIRECTION`) und einem Öffnungswinkel (`GL_SPOT_CUTOFF`) zum Spotlicht wird.
Die Beleuchtung durch *OpenGL* findet *per-vertex* statt und ist eine ohne Einschränkungen echtzeitfähige Beleuchtung, die durch die Vernachlässigung von indirekter Beleuchtung und Schattenwurf jedoch wenig realistisch wirkt. Lichtquellen haben in *OpenGL* zudem keine Geometrie und sind daher nicht sichtbar. Bei Bedarf müssen sie durch emittierende Polygone simuliert werden. Aufgrund der Beschränkung auf die einfachen drei Basislichtquellen ist es schwierig, komplexe Lichtquellen wie Fahrzeugscheinwerfer zu simulieren. Durch die Kombination von acht verschiedenen *OpenGL*-Lichtquellen[2] wurde dies jedoch für den BMW Fahrsimulator realisiert [SBL99]. Die scheinwerfereigene Lichtverteilungskurve kann durch dieses Vorgehen jedoch nur grob angenähert werden. Je mehr Lichtquellen in der Szene platziert werden, desto länger dauert der Renderingvorgang. Um eine bessere Darstellung der Lichtverteilungskurve zu erhalten, wurde später in [LSB+01] eine texturbasierte Beleuchtung für die BMW Fahrsimulation vorgestellt (vgl. Kapitel 4.1.3).

4.1.2 Klassische globale Beleuchtungsmodelle

Im Echtzeitrendering ist es derzeit die Norm, für die Beleuchtung lokale Beleuchtungsmodelle zu benutzen. Da die realistischste Beleuchtung jedoch nach wie vor die globalen Modelle bieten, werden diese nachfolgend anhand der klassischen Beleuchtungsmodelle auf den möglichen Einsatz im Fahrsimulator und damit auf ihre Echtzeitfähigkeit untersucht.
Der Einsatz des Raytracings, das eine sehr gute Repräsentation der direkten Beleuchtung bietet, wäre für die Simulation des Lichts der Fahrzeugscheinwerfer sehr wünschenswert. Bisher galt dieses Verfahren aber, vor allem durch die vielen durchzuführenden Schnittpunktoperationen und der Abhängigkeit von einem festen Betrachterblickpunkt, als nicht einsetzbar in dynamischen Echtzeitsimulationen. Nichtsdestotrotz gehen die Bemühungen weiter, dies in Zukunft zu erreichen. Bereits in Referenz [WSBW01] wird das klassische Raytracing durch Verwendung koherenter Strahlenpakete und effizienter Nutzung der CPU Architektur beschleunigt. Der wichtigste Aspekt, der diese Beschleunigung ermöglicht, ist die Zusammenfassung von vier koherenten Strahlen zu einem Strahlenpaket. Diese Zusammenfassung der Strahlen macht die Nutzung der *SIMD*[3]-Erweiterung moderner Prozessoren möglich. Mit Hilfe dieser Erweiterung werden für alle vier Strahlen parallel sowohl die Schnittpunkte innerhalb der Szene berechnet, als auch das Shading durchgeführt. Zusammen mit einer ebenfalls SIMD-optimierten Schnittpunktberechnung und einer effektiven Nutzung des CPU–Caches erreicht das von Wald et al. entwickelte *RTRT*/OpenRT[4]-System schon interaktive Bildwiederholungsraten. Basierend auf diesem System beschreibt Purcell in [PBMP02] die Möglichkeit, das Raytracing auf programmierbaren Graphikkarten zu realisieren. Dazu wird das Raytracing in einen Streaming Prozess umformuliert

[1] Die Position der Lichtquelle wird in homogenen Koordinaten angegeben, die w-Koordinate entscheidet dabei, ob es sich um gerichtetes Licht ($w = 0$) oder eine Punktlichtquelle ($w \neq 0$) handelt.
[2] Die Anzahl der verfügbaren Lichtquellen ist beschränkt durch die verwendete Hardware.
[3] Single Instruction Multiple Data
[4] RTRT (Real Time Ray Tracing) bezeichnet die Raytracing Engine, während OpenRT deren API darstellt. Inzwischen ist daraus ein komplettes Rendering System entwickelt worden.

und so an die Architektur der GPU angepasst. Diese hardwareseitige Optimierung reicht jedoch auch nicht aus, um die Bildwiederholungsraten entscheidend zu erhöhen. Dass an einen Raytracing Einsatz in dynamischen Szenen derzeit leider noch nicht zu denken ist, zeigt zudem auch die von Wald et al. [WBS03] beschriebene Raytracingimplementierung für dynamische Szenen, deren Ergebnisse zwar bereits interaktiv, aber bei weitem noch nicht echtzeitfähig sind.

Auch die anderen globalen Beleuchtungsmodelle bereiten große Probleme in dynamischen Szenen. Der Einsatz von Radiosity ist aus zwei Gründen schwierig: Zum einen ändern sich die Formfaktoren sobald sich die Geometrie innerhalb der Szene ändert, und zum anderen ist Radiosity eine *per-vertex* Beleuchtung. Damit ist die Qualität der Beleuchtung abhängig von der Tesselierung der Szene. Benedek und Szécsi stellen in [BS02] ein Verfahren vor, mit dem das Photon Mapping in interaktiven Umgebungen einsetzbar wird. Die Beschleunigung des Verfahrens basierte dabei auf einer veränderten Datenstruktur. Die Autoren unterscheiden zwei verschiedene *kd*-Bäume[5], in denen die Photonen gespeichert werden, die auf diffuse Oberflächen treffen. Der erste Baum repräsentiert die statischen, der zweite die dynamischen Objekte. Durch diese Aufteilung wird vor allem die zur Rekonstruktion der Bäume benötigte Zeit verkürzt. In einer dynamischen Szene wie sie im Fahrsimulator zu finden ist, scheitert diese Vorgehensweise aber an der Tatsache, dass dort im Bezug auf die Lichtquelle fast keine statischen Objekte zu finden sind.
Auch wenn die globalen Beleuchtungsmodelle die bestmögliche Qualität und den größten Realismus bieten, ist an einen echtzeitfähigen Einsatz derzeit nicht zu denken. Selbst die enormen Leistungssteigerungen der Graphikhardware reichen noch nicht aus, diese Verfahren im Darstellungsbereich von 60 Bildern pro Sekunde zu realisieren, was jedoch für den Einsatz im Fahrsimulator erforderlich ist.

4.1.3 Texturbasierte Beleuchtungsverfahren

Neben der Beleuchtung durch die klassischen Beleuchtungsmodelle gibt es auch die Möglichkeit, die Beleuchtung auf Objekten durch verschiedene texturbasierte Verfahren zu simulieren. Der größte Vorteil an dieser Art von Beleuchtungssimulation ist, dass sie unabhängig von der Geometrie der Szene ist.

Light Mapping

Lange bevor erste Beleuchtungstechniken auf programmierbarer Graphikhardware eingesetzt wurden, hat sich vor allem in der Spieleindustrie ein Beleuchtungsverfahren namens *Light Mapping* durchgesetzt. Das Light Mapping ist mit der normalen Texturierung zu vergleichen. Auf ein Polygon werden dabei zwei verschiedene Texturen gemappt, zum einen die Textur selbst und zum anderen eine spezielle Light Map. Diese besteht aus sogennanten *Lumeln*[6], die verschiedene Helligkeiten repräsentieren. Diese beiden Texturen werden, meist in zwei verschiedenen

[5] *kd*-Bäume stellen eine Verallgemeinerung eines binären Suchbaums mit k-Dimensionen dar.
[6] Ein Lumel entspricht einem Pixel der Light Map, vergleichbar mit dem Texel einer Textur.

Durchgängen, auf das Polygon gemappt. Die Farbe eines Pixels ergibt sich dabei aus der Multiplikation der entsprechenden Texturfarben. Alternativ können die beiden Texturen auch summiert oder geblendet werden. Das Light Mapping ist auf eine diffuse Beleuchtung ausgelegt, da sich diese mit einem veränderten Blickpunkt nicht ändert. Durch die daraus resultierende Möglichkeit, das Light Mapping in Szenen mit diffuser statischer Beleuchtung offline vorberechnen zu können, hat diese Technik eine weite Verbreitung im Bereich von 3D-Spielen gefunden. Aus der Light Map und der Textur kann offline eine neue Textur berechnet werden, die dann während der Fragment Operationen ganz normal zur Texturierung verwendet wird. Dieses Vorgehen macht es möglich, realistische diffuse Beleuchtungseffekte im Vorfeld, beispielsweise mit Radiosity, zu berechnen.
Da die Beleuchtung für Szenen mit statischer Beleuchtung offline vorberechnet werden kann, ist das Light Mapping in diesen Fällen in Echtzeit möglich. Sobald sich jedoch eine Lichtquelle bewegt, ändert sich die Lichtverteilung auf den angestrahlten Polygonen in Form und Intensität, und die entsprechende Light Map muss neu berechnet oder transformiert werden. Dies schränkt den Echtzeiteinsatz in Szenen mit dynamischer Beleuchtung ein. Light Maps sind auch aus einem anderen Grund für die Beleuchungssimulation im Fahrsimulators nicht gut geeignet. Ausschlaggebend für die Lichtsimulation sind hier nämlich vor allem die Fahrzeugscheinwerfer, die sehr komplexe Lichtverteilungskurven besitzen. Um solche Scheinwerfer zu simulieren, ist es von entscheidender Bedeutung deren Lichtabnahme im Raum zu beachten. Dies kann mit einer zweidimensionalen Light Map nicht realisiert werden.

Dreidimensionales Light Mapping

Was die zweidimensionalen Light Maps an ihre Grenzen führt, kann jedoch mit dreidimensionalen Light Maps realisiert werden. Genau wie in herkömmlichen Light Maps wird deren Helligkeitsverteilung durch die Werte der Lumel repräsentiert. Der Unterschied liegt darin, dass die Light Maps hier einen dreidimensionalen Körper aus Lumeln darstellen. Repräsentiert wird dieser durch ein dreidimensionales Array von Farbwerten, von denen jedem eine Position im Raum zugeordnet ist. Obwohl die Light Map eine räumliche Ausdehnung hat, bewirkt sie nur dort einen Effekt, wo sie Polygone in der Szene schneidet. Die Lichtquellen selbst können dabei, wie von McReynolds et al. beschrieben [MBG+99], als einfache texturierte Geometrie repräsentiert werden. Sie stellt den Ausgangspunkt der Light Map dar und wird daher hell texturiert. Durch den Einsatz dreidimensionaler Light Maps ist es somit möglich, flächige Lichtquellen darzustellen.
Die Vorteile, die der Einsatz der dreidimensionalen Texturen zur Beleuchtungssimulation hat, sind der hohe Realitätsgrad, mit dem eine dreidimensionale Lichtverteilung abgebildet werden, kann und die einfache Bestimmung der Texturkoordinaten. Demgegenüber steht jedoch ein hoher Speicherbedarf der dreidimensionalen Texturen. Eine 32bit Textur, die beispielsweise bei einer zweidimensionalen Ausdehnung von 256 x 256 Pixeln lediglich einen Speicherplatzbedarf von 256Kb benötigt, verlangt in einer dreidimensionalen Form von 256 x 256 x 256 Pixeln bereits 64Mb. Dies entspricht einem Viertel des auf der verwendeten *nVIDIA QuadroFX 3000G* verfügbaren Speicherplatzes, der damit auch bei Verwendung von Texturkompressionsverfahren schnell an seine Grenzen stößt.

Projektive Texturen

Eine Methode, die in Fahrsimulationen bereits häufig zur Beleuchtung eingesetzt wird, ist die Verwendung projektiver Texturen. Die Grundidee, die hinter dieser Methode steht, ist, wie bei den vorgestellten Light Maps auch, die Darstellung der Helligkeitsverteilung durch eine Textur. Der Unterschied liegt darin, dass die zweidimensionale Light Map nicht auf eine Oberfläche gemappt, sondern projiziert wird. Dieser Vorgang ist vergleichbar mit der Projektion eines Dias auf ein Objekt. Der Projektor stellt dabei die Lichtquelle dar, die eine Oberfläche durch ein Dia (die Light Map) anstrahlt. Dadurch können bei diesem Verfahren, im Gegensatz zu den zweidimensionalen Light Maps, perspektivische Informationen berücksichtigt werden.
Die projektive Texturierung erfordert vor allem die korrekte Berechnung der Texturkoordinaten für alle Polygone. Durch Transformationen, die mit der perspektivischen Projektion innerhalb der Graphik Pipeline vergleichbar sind, werden die Objektkoordinaten der Polygone in zweidimensionale Koordinaten umgerechnet. Mit Hilfe dieser Texturkoordinaten können die Helligkeitswerte aus der Lichttextur ermittelt werden. Gleichung 4.1 beschreibt die Berechnung dieser Koordinaten.

$$\begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix} = \begin{pmatrix} 0.5 & 0.0 & 0.0 & 0.5 \\ 0.0 & 0.5 & 0.0 & 0.5 \\ 0.0 & 0.0 & 0.5 & 0.5 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} * \begin{pmatrix} Light \\ Projection \\ Matrix \end{pmatrix} * \begin{pmatrix} Light \\ View \\ Matrix \end{pmatrix} * \begin{pmatrix} Model \\ \\ Matrix \end{pmatrix} * \begin{pmatrix} x_M \\ y_M \\ z_M \\ w_M \end{pmatrix} \quad (4.1)$$

Zunächst werden die Vertices, die in Modellkoordinaten vorliegen, den Transformationen der *Model Matrix* unterzogen und damit in Weltkoordinaten umgerechnet. Soweit stimmt die Vorgehensweise noch mit der der Standard Renderingpipeline überein. Im Anschluss daran werden die Vertices jedoch nicht ins Kamerakoordinatensystem, sondern mittels *LightView Matrix* in das Koordinatensystem des Lichts transformiert. Die Projektion findet entsprechend des Licht-Frustums mittels *LightProjection Matrix* statt. Nach Anwendung dieser Matrizen liegen die Texturkoordinaten eigentlich schon vor, die Werte befinden sich nun jedoch im Bereich zwischen -1 und 1, während Texturen einen Wertebereich von 0 bis 1 besitzen. Abschließend werden die Vertices daher mit der angegebenen Matrix multipliziert, um sie in diesen Wertebereich zu mappen.
Die für die Vertices berechneten Texturkoordinaten werden für jedes Fragment interpoliert. Während der Fragment Operationen kann damit die Bestimmung der Helligkeit aus der Textur erfolgen. Bevor der Texture-Lookup erfolgen kann, wird die vierdimensionale wie folgt in eine zweidimensionale Texturkoordinate umgerechnet, um die perspektivische Information zu erhalten:

$$(s,t) = (\frac{s}{q}, \frac{t}{q}) \quad (4.2)$$

Die Beleuchtungsqualität dieses Verfahrens ist durch die Berücksichtigung der Lichtinformation (in Form einer Lichttextur) pro Fragment nicht abhängig von der Tesselierung der Oberflächen, sondern nur von der Auflösung der Textur.
Segal et al. beschrieben bereits in Referenz [SKvW$^+$92], wie dieses Verfahren zur Realisierung schneller Schatten- und Lichtberechnungen eingesetzt werden kann. Um die fehlende räumliche

Ausdehnung der Light Map zu berücksichtigen, dividierten sie dabei die z-Koordinate der projektiven Texturkoordinate durch deren homogene Koordinate w. Dadurch wird der tatsächliche Tiefenwert in Abhängigkeit zur Lichtquelle ermittelt. Dieser kann dann zur Realisierung einer Lichtabnahme im Raum verwendet werden.
In Referenz [LKK99] wurde gezeigt, dass die projektive Textur auch zur Beleuchtung im Fahrsimulator eingesetzt werden kann. Das vorgestellte Verfahren benutzt die projektive Textur als Repräsentation der Lichtverteilung der Scheinwerfer für eine Beleuchtungsberechnung pro Vertex. An jedem Vertex wird die einfallende Leuchtdichte mit Hilfe der projektiven Textur ermittelt, die in einem zweiten Renderingschritt mit einem maximalen Beleuchtungswert multipliziert wird, und so die Beleuchtung an den Polygoneckpunkten darstellt. Die so bestimmten Helligkeitswerte werden anschließend für die einzelnen Fragmente interpoliert. Das Verfahren verfügt bereits über eine sehr gute Beleuchtungsberechnung, da diese jedoch per-vertex stattfindet, kann die Unabhängigkeit der projektiven Texturierung von der Szenentriangulierung nicht ausgenutzt werden. Zusätzlich wirkt sich der benötigte zweite Renderingpass negativ auf die Performance aus.
Unabhängig von der Geometrie der Szene verhält sich hingegen das projektive Beleuchtungsverfahren, das von *BMW* [LSB+01] in die damalige Fahrsimulationssoftware integriert wurde, um die Entwicklungen des Projektes *Adaptive Light Control* [LBR+99] zu evaluieren. Mittels spezieller projektiver *Performer*-Lichtquellen wurde diese Simulation auch in SPIDER realisiert. Die Beleuchtung findet dabei nicht *per-vertex* statt, sondern die Lumel der auf die Oberfläche projizierten Beleuchtungstextur werden mit den Texeln der Oberflächentextur verrechnet. Dadurch ist die Beleuchtung nicht mehr abhängig von der Tesselierung der Szene. Die durch Messungen ermittelte Beleuchtungstextur wird auf die Oberfläche vor dem Fahrzeug projiziert. Eine korrekte Berechnung der Beleuchtung findet nicht statt, da der Einfallswinkel der Lichtstrahlen unberücksichtigt bleibt. Trotz dieser Einschränkung liefert die Simulation qualitativ gute Resultate. Der Nachteil des Verfahrens ist hingegen, dass die Beleuchtungssimulation durch die Verwendung spezieller Performer-Lichtquellen auf dieses System beschränkt ist. Sie kann daher nur anhand spezieller vordefinierter Parameter verändert werden – eine Erweiterung oder Anpassung der Funktionalität ist mit dieser „Black Box“-Realisierung nicht möglich.
Mit einem ähnlichen Beleuchtungsansatz ist auch die Nachtfahr-Simulationssoftware *Nightdriver* [WPKB02] der Firma *Hella KG* in Zusammenarbeit mit dem *Heinz Nixdorf Institut* erweitert worden, um eine geometrieunabhängige Beleuchtung zu erhalten [GBMP03]. Diese ist jedoch, wie das von Lecocq et al. vorgestellte Verfahren, durch den erforderlichen zweiten Renderingvorgang nur beschränkt echtzeitfähig.

4.2 Photometrische Betrachtung der Beleuchtungssituation

Eine Beleuchtungssimulation wirkt umso realistischer, je mehr sie an die physikalischen Gesetze der Photometrie angelehnt ist. Welche der dafür erforderlichen Informationen sind jedoch in SPIDER verfügbar und wie kann eine möglichst realistische Berechnung der Beleuchtung aussehen? Im Folgenden wird diese Fragestellung genauer betrachtet, und es wird aufgezeigt, wie berechnete Beleuchtungswerte richtig dargestellt werden können.

4.2.1 Photometrische Beleuchtungsberechnung

Als Grundlage für eine photometrisch basierte Beleuchtung dient die Rendering-Equation von Kajiya [Kaj86], die die Basis vieler Beleuchtungsmodelle ist:

$$L_o(dA_e, d\vec{\omega}_0) = L_e(dA_e, d\vec{\omega}_0) + \int_s \rho(dA_e, \vec{\omega}_i, \vec{\omega}_i) L_i(dA_e, \vec{\omega}_o) \cos\theta_i d\vec{\omega}_i \tag{4.3}$$

Diese Gleichung stellt eine physikalische Beschreibung der natürlichen Interaktion von Licht und Material für ein infinitesimal kleines Flächenelement dA_e und den Raumwinkel $d\vec{\omega}_o$ dar. Die Leuchtdichte[7] L_o, die von dem Flächenelement in Richtung $d\vec{\omega}_o$ ausgestrahlt wird, ergibt sich aus der eigenen in Richtung $d\vec{\omega}_o$ ausgestrahlten Leuchtdichte des Flächenelementes und des von anderen Lichtquellen der Umgebung eingestrahlten Lichts, das in Richtung $d\vec{\omega}_o$ reflektiert wird. Wie stark das einfallende Licht reflektiert wird, hängt von der sogenannten *BRDF*[8] ab. Da eine solche Betrachtung der Lichtsimulation für die Fahrsimulation zu aufwendig ist, werden zwei Einschränkungen gemacht:

1. Die Integration über alle Raumwinkel wird durch die Summe des Lichts aller Lichtquellen ersetzt.

2. Die BRDF wird in Annahme diffuser Materialien durch den konstanten Reflexionsgrad ρ ersetzt, der das Verhältnis zwischen ein- und ausfallendem Lichtstrom mit einem Wert im Bereich $[0.0, 1.0]$ angibt.

Unter Berücksichtigung dieser Annahmen kann man die Rendering-Equation wie folgt vereinfachen:

$$L_o(dA_e, d\vec{\omega}_0) = L_e(dA_e, d\vec{\omega}_0) + \rho \sum_{s=1}^{n} L_i(dA_e, \vec{\omega}_o) \tag{4.4}$$

Ist es möglich die Leuchtdichte für jede einzelne Lichtquelle zu bestimmen, so kann auf diese Weise die daraus resultierende Leuchtdichte für das aktuelle Flächenelement bestimmt werden. Für die Berechnung der Beleuchtung ist jedoch ausschlaggebend, welche photometrischen Größen zur Lichtberechnung verfügbar sind. Als Basis für die folgende Betrachtung dienen gemessene Beleuchtungsstärken verschiedener Scheinwerfer, die in Form von Lichtbündelquerschnitten[9] vorliegen. Zu deren Bestimmung wird ein Scheinwerfer auf ein sogenanntes Goniometer montiert und ein Lichtsensor in einer bestimmten Entfernung angebracht. Das Goniometer erlaubt es, den Scheinwerfer sehr genau in beliebigen Winkeln auf den Sensor auszurichten. Dadurch kann die Beleuchtungsstärke durch das Licht eines Scheinwerfers auf einer Empfängerfläche ermittelt werden.
Alternativ zu dieser sehr genauen Messung, kann ein Lichtbündelquerschnitt auch, wie in Abbildung 4.1 schematisch dargestellt, an einer Messwand ermittelt werden. Für alle in einem be-

[7] Im Gegensatz zu der in Kapitel 3.4 verwendeten Lichtfarbe $\vec{L}$ wird hier mit L die Leuchtdichte bezeichnet.
[8] Die Bidirektionale Reflexions und Distributions Funktion beschreibt die Wechselwirkung zwischen Licht und Materie unter Berücksichtigung aller verschiedenen Ein- und Ausfallswinkel.
[9] Ein Lichtbündelquerschnitt stellt einen vertikalen Schnitt durch den Lichtkegel eines Scheinwerfers in einer bestimmten Distanz dar.

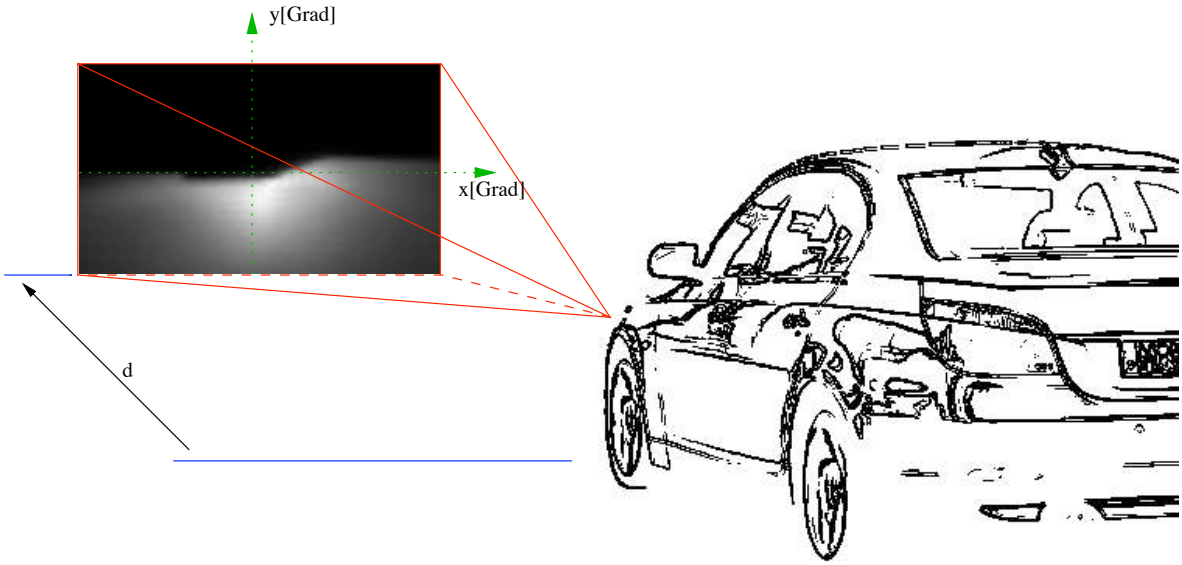


Abbildung 4.1: Ermittlung der Scheinwerfer Beleuchtungswerte an einer Messwand

stimmten Raster eingestellten Winkel des Messbereiches liegen die Beleuchtungsstärken für eine bestimmte Distanz d vor. Wie kann jedoch die Beleuchtungsstärke eines beliebigen Punktes auf der Fahrbahnoberfläche korrekt bestimmt werden? Die Lösung dieser Frage findet sich in der Definition der Beleuchtungsstärke.
Für infinitesimal kleine Empfängerflächen entspricht die Beleuchtungsstärke dem einfallenden Lichtstrom ϕ pro Empfängerfläche A_e (vgl. im Weiteren Gleichung 4.5). Der einfallende Lichtstrom kann auch durch die Lichtstärke I und den entsprechenden Raumwinkel ω dargestellt werden. Da dieser für infinitesimal kleine Flächen durch deren Multiplikation mit dem Kosinus des Flächenneigungswinkels α im Verhältnis zum quadrierten Abstand d^2 zur Lichtquelle angenähert werden kann (vgl. Tabelle 3.1), entfällt die Flächenabhängigkeit, und es ergibt sich Gleichung 4.6.

$$E = \frac{d\phi}{dA} = \frac{I \cdot d\omega}{dA} = \frac{I \cdot dA \cdot \cos\alpha}{dA \cdot d^2} \tag{4.5}$$

$$E = \frac{I \cdot \cos\alpha}{d^2} \tag{4.6}$$

Mit Hilfe dieser Gleichung ist es möglich, die Beleuchtungsstärken für eine bestimmte Position in der Szene anzunähern. In Abbildung 4.2 ist ein Beleuchtungsszenario dargestellt, für das im Folgenden die Berechnung durchgeführt wird. Der Punkt L repräsentiert einen Scheinwerfer, der einen Punkt P auf der Fahrbahnoberfläche beleuchtet. Beide besitzen eine Normalen n_P

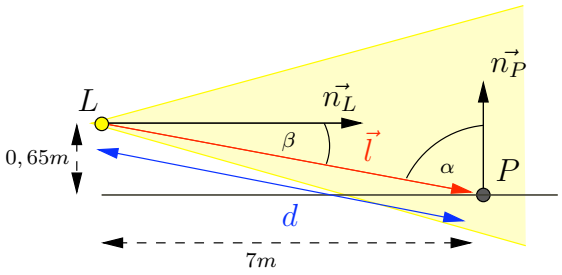


Abbildung 4.2: Berechnung der Beleuchtungsstärke

bzw. n_L. Die Beleuchtungsstärken des Scheinwerfers liegen in Form gemessener Werte vor, die in einer Distanz von 25 Metern ermittet wurden. Anhand des Punktes P wird die Berechnung der Beleuchtungsstärke dargestellt. Gegeben sind:

$$\begin{aligned} L &= (0.0m, -0.65m, 0.0m) \\ \vec{n_L} &= (1.0m, 0.0m, 0.0m) \\ P &= (0.0m, 0.0m, 7.0m) \\ \vec{n_P} &= (0.0m, 1.0m, 0.0m) \end{aligned}$$

Zur Berechnung werden zusätzlich der Vektor $\vec{l}$ (von L nach P), der Lichtabstrahlwinkel β, der Lichteinfallwinkel α und die Entfernung d des Punktes P von der Lichtquelle benötigt. Diese Informationen können wie folgt ermittelt werden:

$$\begin{aligned} \vec{l} &= P_1 - L = (0.0m, -0.65m, 7m) \\ d &= \sqrt{{l_x}^2, {l_y}^2, {l_z}^2} = 7.030m \\ \beta &= \arcsin \frac{L_y - {P_1}_y}{d} = -5.3° \end{aligned}$$

Der Kosinus des benötigten Lichteinfallswinkel lässt sich durch das Skalarprodukt zwischen dem negierten und normierten Lichtvektor und der Oberflächennormalen an Punkt P bestimmen:

$$\begin{aligned} \vec{l_{norm}} &= \frac{\vec{l}}{d} \\ \cos\alpha &= < \vec{n_1}, -\vec{l_{norm}} > \\ &= 0,092 \end{aligned}$$

Damit sind alle Informationen ermittelt, die zur Berechnung der Beleuchtungsstärke an Punkt P benötigt werden. Mit Hilfe des Winkels β kann die Beleuchtungsstärke $E_M(0.0, \beta)$ aus den vorliegenden Messwerten bestimmt werden. Anschließend wird der Lichtstrom, der von der Lichtquelle in Richtung des Winkels β fließt, berechnet. Da die Messung mit senkrecht einfallendem

Licht durchgeführt wurde, entfällt der Kosinus des Einfallswinkels, da er 1 ergibt.

$$\begin{aligned} E_M(0.0, \beta) &= 0,33lx \\ I &= \frac{E \cdot d^2}{\cos \alpha} = 0,33lx \cdot 625m^2 = 206,25cd \end{aligned} \tag{4.7}$$

Mit Hilfe des Lichtstroms, kann anschließend die Beleuchtungsstärke an Punkt P_1 bestimmt werden (vgl. Gleichung 4.6):

$$E(P_1) = \frac{206,25cd \cdot 0,092}{7,030m} = 2,6lx \tag{4.8}$$

Auf diese Weise kann für jeden beliebigen Punkt der Szene dessen Beleuchtungsstärke durch die Lichtquelle berechnet werden. Wie kann man nun aber auf die entsprechende Leuchtdichte schließen, wie sie in der Rendering Equation verwendet wird? Da bereits die Annahme gemacht wurde, dass nur diffuse Materialien vorliegen, kann man die Flächen als Lambert Strahler interpretieren, die das Licht gleichmäßig in alle Richtungen ausstrahlen. Für diesen Fall kann man die Leuchtdichte wie folgt aus der Beleuchtungsstärke berechnen:

$$L = \frac{E}{\pi} \tag{4.9}$$

Entsprechend des vorgestellten Vorgehens kann auch die Beleuchtungsstärke eines Punktes durch den zweiten Scheinwerfer ermittelt werden. Die Scheinwerfer stellen jedoch nicht die einzigen Lichtquellen der Szene dar. Die Schwierigkeit der Lichtberechnung in der SPIDER Umgebung beginnt bei der Berücksichtigung weiterer Lichtquellen wie dem Mond oder Straßenlaternen. Für diese sind keine photometrischen Informationen vorhanden. Um die Leuchtdichte gemäß Gleichung 4.4 berechnen zu können, müsste die Leuchtdichte aller Lichtquellen bestimmbar sein. Im folgenden Kapitel wird betrachtet, wie die berechneten Leuchtdichten in RGB-Werten dargestellt werden können, und wie Lichtquellen verschiedener Leuchtdichtebereiche auf den beschränkten Darstellungsbereich eines Ausgabemediums skaliert werden können.

4.2.2 Darstellung von Beleuchtungswerten

Liegen für eine Szene berechnete Leuchtdichtewerte vor, gilt es diese den Eigenschaften des Ausgabegerätes entsprechend auf RGB-Werte zwischen 0 und 1 abzubilden. Dabei ergeben sich zwei Fragestellungen:

1. Welchem RGB-Wert entspricht ein bestimmter Leuchtdichtewert?
2. Wie können Leuchtdichtewerte auf einem Bildschirm abgebildet werden, wenn diese außerhalb seines darstellbaren Leuchtdichtebereichs liegen?

Die erste Fragestellung hängt entscheidend mit der Bedeutung eines Farbwertes zusammen. Wie und mit welcher Leuchtdichte beispielsweise der Wert $RGB = (0.0, 0.0, 1.0)$ dargestellt wird,

ist abhängig von dem verwendeten Ausgabegerät. Vereinfacht ausgedrückt wird er vom Ausgabegerät als „leuchte volle Kraft blau“ interpretiert. Das bedeutet aber im Gegenzug auch, dass man dadurch auf verschiedenen Ausgabegeräten als Resultat sowohl eine unterschiedliche Farbdarstellung als auch unterschiedliche Helligkeiten erhält. Der RGB-Farbraum ist demnach nicht eindeutig. Aus diesem Grund besitzt jedes Ausgabemedium eine RGB-Farbmatrix, die idealerweise durch Messungen bestimmt, oder aus Hersteller-Angaben übernommen werden kann. Mit Hilfe dieser Matrix ist es möglich, die RGB Farbwerte in den CIE[10]-XYZ Farbraum umzurechnen.
Dieser Farbraum beschreibt alle vom Menschen wahrnehmbaren Farben in Form von Normspektralkurven $X(\lambda), Y(\lambda), Z(\lambda)$ und repräsentiert einen eindeutigen und geräteunabhängigen Farbraum. Die Normspektralkurven stellen die Empfindlichkeitskurven eines Standard-Beobachters auf vom CIE entworfene imaginäre Lichtquellen dar, die unterschiedliche charakteristische Eigenschaften haben:

- eines der Lichter trägt keinerlei Farb-, sondern nur Helligkeitsinformationen
- die anderen beiden Lichter tragen nur Farb- und keine Helligkeitsinformationen

Die Empfindlichkeitskurven $X(\lambda), Y(\lambda), Z(\lambda)$ (Normspektralkurven) dieses Lichts können entsprechend Gleichung 3.2 in die sogenannten Normfarbwerte XYZ umgerechnet werden. Die Y-Komponente dieses Systems trägt die photometrische Information (ohne die Skalierung mit K_m) und ist notwendig, um die Leuchtdichte mit einer RGB-Farbe in Beziehung zu setzen.
Wie kann nun ein berechneter Leuchtdichtewert durch eine RGB Farbe dargestellt werden? Als Beispiel sei folgende Farbmatrix gegeben (Quelle: [Mül03]):

$$M = \begin{pmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{pmatrix} = \begin{pmatrix} 0,183 & 0,053 & 0,057 \\ 0,09 & 0,18 & 0,033 \\ 0,0 & 0,02 & 0,032 \end{pmatrix} \tag{4.10}$$

Mit Hilfe der Y-Komponente dieser Farbmatrix, multipliziert mit dem , kann nun ein Leuchtdichtewert (bspw. $50\frac{cd}{m^2}$) in einen Farbwert umgerechnet werden. Dabei wird davon ausgegangen, dass das Licht keine Farbinformation enthält und somit aus gleichen RGB Anteilen c besteht.

$$L = K_m \cdot (Y_r + Y_g + Y_b) \tag{4.11}$$

$$Y_r + Y_g + Y_b = \frac{L}{K_m} \tag{4.12}$$

$$0,09 \cdot c + 0,18 \cdot c + 0,033 \cdot c = \frac{50}{683} \tag{4.13}$$

$$c = 0,242 \tag{4.14}$$

[10] Commission Internationale de l’Eclairage

Art der Beleuchtung	durchschnittliche max. Beleuchtungsstärken (lx)
Vollmondnacht	0,2
Tageslicht	27.000
direkte Sonneinstrahlung	100.000
Beleuchtung[12] durch:	
Straßenlaternen	20
1 Halogen-Scheinwerfer KFZ	100
1 Xenon-Scheinwerfer KFZ	220

Tabelle 4.1: Beispiele verschiedener Beleuchtungsstärken

Vorausgesetzt die bestimmten Werte liegen im Bereich der vom Ausgabemedium darstellbaren Leuchtdichtewerte, kann durch dieses Vorgehen die berechnete Leuchtdichte photometrisch konsistent auf das Ausgabegerät übertragen werden[11]. So kann auf dem Ausgabegerät der berechnete Wert in der korrekten Leuchtdichte dargestellt werden.

In der Praxis ist es jedoch selten möglich, die berechneten Leuchtdichten ohne Skalierung auf dem Ausgabemedium abzubilden, da der berechnete Leuchtdichtebereich meist außerhalb des vom Gerät Darstellbaren liegt. Tabelle 4.1 stellt typische Beleuchtungsstärken dar, wie sie in der Realität vorkommen. Es ist offensichtlich, dass solche Beleuchtungswerte von Ausgabegeräten nicht erreicht werden können. Für das Ausgabegerät mit der in Gleichung 4.10 notierten Farbmatrix kann man beispielsweise eine maximale Leuchtdichte von $207cd/m^2$ bestimmen, was nicht zur Darstellung solcher Wertebereiche ausreicht. Falls die Leuchtdichten skaliert werden müssen, wird das Tone Mapping verwendet um die berechneten Beleuchtungswerte auf die des Gerätes abzubilden. Dieser Fall ist bei der Simulation von Tageslichtszenen die Regel, doch auch in Nachtlichtszenen kann je nach Anzahl und Stärke der Scheinwerfer ein Tone Mapping erforderlich sein.
Die einfachste Möglichkeit, die berechneten auf die darstellbaren Werte abzubilden, stellt das lineare Mapping dar. Dabei entspricht die maximale berechnete Leuchtdichte dem Farbwert $RGB = (1, 1, 1)$. Da die Lichtquellen in der Regel einige Größenordnungen heller sind als die Umgebung, führt dies meist dazu, dass die Lichtquellen zwar sichtbar, der Rest der Szene jedoch fast schwarz dargestellt wird. Ein etwas verbesserter Eindruck entsteht, wenn die Helligkeitswerte mit Ausnahme der Lichtquellen auf den Wertebereich $[0, 1[$ und nur Lichtquellen auf 1 skaliert werden. Damit ist die Szene besser ausgeleuchtet. Der Nachteil an dieser Skalierung ist, dass der Helligkeitseindruck verloren geht, weil eine Veränderung der Lichtemission der Lichtquellen keinen Effekt mehr zeigt. Bessere Tone Mapping Verfahren skalieren die Werte nicht linear,

[11] Eine korrekt eingestellte Gamma-Korrektur vorausgesetzt, denn ein linearer Anstieg der RGB-Werte resultiert meist nicht in einem linearen Anstieg der Leuchtdichtewerte des Ausgabegerätes.

sondern beispielsweise logarithmisch oder durch Histogramm-Anpassung (vgl. [SS00],[Mat97]). Mit dem geeigneten Tone Mapper ist es möglich, auch Lichtquellen in stark unterschiedlichen Leuchtdichtebereichen auf einem beschränkten Darstellungsbereich abzubilden.
Aus Performancesicht besonders geeignet ist der sogenannte *URQV*-Tone Mapper ([Mül03]), da dieser aus einer relativ einfachen Berechnung besteht. Um eine berechnete Leuchtdichte L eines Wertebereichs von 0 bis L_{max} auf den Bereich zwischen 0 und 1 abzubilden, wird folgende Formel verwendet:

$$L' = \frac{s \cdot L}{s \cdot L - L + L_{max}} \tag{4.15}$$

Auf welchen Zielwert ein berechneter Beleuchtungswert abgebildet wird, hängt entscheidend von dem Skalierungsfaktor s ab. Wird $s = 0$ gewählt, werden die Werte linear zwischen 0 und 1 skaliert, durch ein größer gewähltes s werden die Werte nicht-linear skaliert.

4.3 Schattenintegration

Neben der Lichtsimulation ist die Schattenberechnung ein wichtiger Bestandteil einer vollständigen Beleuchtungsberechnung. Bisher ist in der BMW Fahrsimulation lediglich der von den Fahrzeugen erzeugte Schatten realisiert (siehe Abbildung 4.3). Es handelt sich dabei um ein dunkles Schattenpolygon, das unterhalb des Fahrzeugs auf die Fahrbahn gerendert wird. Der so dargestellte Schattenwurf besitzt eine statische Geometrie und eine, im Bezug zum Fahrzeug, statische Position. Diese Form der Schattendarstellung ist für die Simulation der im Zusammenhang mit der Beleuchtung durch Fahrzeugscheinwerfer auftretenden Schatten nicht geeignet. Im Folgenden werden daher die Vor- und Nachteile der in Kapitel 3.5 beschriebenen Verfahren betrachtet, und deren Eignung für den echtzeitfähigen Einsatz wird überprüft.

Abbildung 4.3: Vorberechnete Fahrzeugschatten zur Darstellung auf der Fahrbahn

Planare Schatten Dadurch, dass beim planaren Schattenverfahren lediglich schattenverursachende Objekte ein zweites Mal gerendert werden müssen, werden hohe Rendergeschwindigkeiten erreicht. Durch die Berechung der Projektionsmatrix, die für jedes dieser Objekte erforderlich ist, nimmt die Performance mit jeder weiteren Schattenquelle ab. Die Performance des Verfahren

ist ausreichend hoch, um es in Echtzeit einzusetzen. Für die Schattensimulation in der Fahrsimulation ist es jedoch durch die Beschränkung auf planare Flächen nicht geeignet.

Shadow Volumes Der Einsatz der Shadow Volumes in komplexen Szenen ist problematisch, da die Erstellung der Schattenvolumen sehr aufwendig ist, und ihre Datenstrukturen in diesen Szenen viel Speicherplatz in Anspruch nehmen. Zudem nimmt die Komplexität einer Szene durch zusätzlich benötigte Polygone zu, deren Anzahl vor allem durch weitere Lichtquellen enorm schnell ansteigt. Da die Schattenvolumen meist sehr viele Pixel umfassen, kann die Pixel-Füllrate zum Problem und damit die Rasterisierung zum Bottleneck werden. Dies wirkt sich besonders negativ aus, wenn viele Objekte sehr nah an der Lichtquelle positioniert sind und dadurch sehr große Schattenvolumen entstehen. Der Vorteil des Verfahrens liegt hingegen in der Tatsache begründet, dass es objektbasiert ist. Dadurch werden die Kernschatten sehr korrekt und ohne Aliasingartefakte erzeugt. Außerdem kann dieses Verfahren mit beliebiger Graphikhardware umgesetzt werden, vorausgesetzt ein Stencilbuffer ist vorhanden.
Ein Problem, das im Zusammenhang mit der Verwendung des Stencilbuffers steht, wurde von Carmack durch den *z-fail*-Test gelöst: Befindet sich die Betrachterposition innerhalb eines Schattenvolumens (die near-clipping plane schneidet ein Schattenvolumen), so bedeutet ein Wert gleich 0 im Stencilbuffer nicht mehr, dass der Pixel im Schatten liegt. Carmack führt den bisherigen Stenciltest daher in umgekehrter Richtung aus und verfolgt den Sichtstrahl aus dem Unendlichen bis zur Betrachterposition. Der Wert des Stencilbuffers wird wieder bei Eintritt in ein Volumen inkrementiert und bei Austritt dekrementiert. Tritt der Strahl dabei öfter in ein Volumen ein als aus, so befindet sich der Pixel im Schatten.
Dynamische Szenen machen es erforderlich, dass die Schattenvolumen ständig neu berechnet werden. Der beschriebene Algorithmus ist jedoch nicht in der Lage, dies in Echtzeit für komplexe Szenen zu bewältigen. Daher finden sich in der Literatur einige Versuche, verschiedene Aspekte des Basisalgorithmus zu optimieren.
Roettger et al. [RIE02] zeigen in dem von Ihnen vorgestellten Algorithmus, dass das Shadow Volume Verfahren auch ohne die Verwendung des Stencilbuffers realisiert werden kann. Stattdessen nutzen sie den Alpha- oder den Screenbuffer, um die Schattenmaske zu generieren. Da in diesen Buffern keine Subtraktionsoperationen durchführbar sind, nutzen sie äquivalente Blendingoperationen um die gespeicherten Werte zu verdoppeln oder zu halbieren. Die Maske wird außerdem mit niedrigerer Auflösung im Buffer gerendert und anschließend in eine Textur kopiert. Diese wird vergrößert verwendet, um ein Rechteck in Größe des Bildschirms zu texturieren und dieses in den Framebuffer zu rendern. Obwohl die Szene dadurch ein weiteres Mal gerendert werden muss, weil die Tiefenwerte in der gleichen Auflösung wie die Textur vorliegen müssen, können die Autoren das Shadow Volume Verfahren etwas beschleunigen. Der Fokus ihrer Veröffentlichung liegt jedoch auf dem Verzicht auf den Stencilbuffer und so verwundert es nicht, dass der erreichte Geschwindigkeitsgewinn im Bereich einer einstellig höheren Framerate liegt.
Batagelo und Junior hingegen versuchen in [Bat99] die Anzahl der Schattenvolumen zu verringern und dadurch einen Geschwindigkeitsgewinn zu erreichen. Dazu verwenden sie einen Shadow Volume BSP[13]-Baum, eine effiziente Datenstruktur zur Speicherung der Sichtbarkeits-

[13] Binary Space Partitioning

ordnung der Szene. Mit Hilfe des BSP-Baums ist es möglich verdeckte Schattenvolumen zu lokalisieren, die anschließend nicht mehr weiter berücksichtigt werden müssen. In Szenen, in denen potentiell viele Schattenvolumen verdeckt sind, eignet sich dieses Verfahren gut. In ihren Tests konnten die Autoren die Frameraten in einigen statischen Testszenen mit dynamischen Lichtquellen sogar verdoppeln. Da die Datenstruktur jedoch im Voraus berechnet und bei Veränderung der Betrachterposition aktualisiert werden muss, rechnet sich der Overhead der BSP-Baum-Berechnung nicht in dynamischen Szenen, wie sie in der Fahrsimulation zu finden sind.
Darüber hinaus gibt es verschiedene Variationen des Basisalgorithmus, die weitere Verbesserungen und Optimierungen des Shadow Volume Algorithmus vorstellen. Everitt und Kilgard haben beispielsweise in [EK02] einen robusten Shadow Volume Algorithmus vorgestellt. Dieser verwendet einerseits den *z-fail* Test, um die Unabhängigkeit von der Betrachterposition zu gewährleisten und platziert andererseits die Far-Clipping Ebene in unendlicher Entfernung, um Clipping Probleme zu vermeiden. Diese Technik wird dabei hardwareunterstützt realisiert. Ähnlich der Vorgehensweise bei den planaren Schatten ist es auch beim Shadow Volume Verfahren möglich, weiche Schattengrenzen zu erzeugen. Dazu können mehrere Schattenvolumen geblendet werden. Dies resultiert jedoch auch wieder im einer höheren Pixel-Füllratet. Eine gute Übersicht über verschiedene Verfahren zur Generierung von weichen Schatten ist von Hasenfratz et al. in [HLHS03] verfasst worden.
Letztlich steht beim Shadow Volume Algorithmus eine qualitativ hochwertige Schattenberechnung einem, in dynamischen und komplexen Szenen, großen Berechnungsaufwand gegenüber. Da solche Szenen jedoch in der Fahrsimulation die Regel sind, ist das Verfahren in Echtzeit nicht für komplette Szenen realisierbar.

Shadow Mapping Der größte Vorteil des Shadow Mapping Technik ist deren Unabhängigkeit von der Szenengeometrie. Das macht diese Technik zu einem sehr schnellen Verfahren, das auch in komplexen Szenen eingesetzt werden kann. Zudem wird das Shadow Mapping mittlerweile größtenteils von der Graphikhardware unterstützt. Nachteilig wirkt sich hingegen die Abhängigkeit der Qualität von der Auflösung der Shadow Map und der Tiefe des z-Buffers aus. Hier gilt es, die richtige Mitte zu finden: Wird die Auflösung zu niedrig gewählt, treten vor allem bei weiter entfernten Objekten Aliasingeffekte an den Schattengrenzen auf. Eine zu hohe Auflösung der Shadow Map resultiert jedoch in einem erhöhten Speicherbedarf. Auch die Quantisierung des z-Buffers führt zu Problemen. Diese kann zur Folge haben, dass der z-Wert der Shadow Map ein anderer ist als der des aktuellen Pixels, welches aber oberhalb oder unterhalb der Fläche eines Objektes liegt. Dies resultiert in einer falschen Selbstbeschattung. Durch Subtraktion eines bestimmten Bias-Wertes können diese falschen Schatten jedoch wieder entfernt werden.
Die Qualität der Schattendarstellung wächst generell mit der Leistungsfähigkeit der Graphikhardware. Insbesondere durch die Unterstützung höherer Auflösungen oder größerer z-Buffer und demnach in Abhängigkeit von dem vorhandenen Graphikkartenspeicher. Stamminger und Drettakis reduzieren Aliasingeffekte mit dem von ihnen in [SD02] vorgestellten perspektivischen Shadow Map. Das Prinzip ihres Verfahrens ist es, die Auflösung der Shadow Maps an die aktuelle Betrachtersicht anzupassen, indem die Map scheinbar eine hohe Auflösung für nahegelegene und eine niedrige Auflösung für weiter entfernte Objekte aufweist. Dazu werden die Shadow

Map Generierung und der Schattentest nach der perspektivischen Division der Graphikpipeline und somit in normalisierten Koordinaten des kanonischen Volumens durchgeführt. Da die perspektivische Projektion hier bereits abgeschlossen ist, repräsentiert das Volumen eine orthographische Sicht auf die Szene. Wird der Schattentest mit einer in diesem Raum generierten Shadow Map der, werden perspektivische Aliasingeffekte vermieden. Die Tatsache, dass das perspektivische Mapping auf der Projektion aus Betrachtersicht basiert, bringt jedoch einige Nachteile mit sich. Die Abhängigkeit von der Betrachterposition führt zum Beispiel dazu, dass die Richtung der Lichtquellen verändert wird. Wimmer et al. haben dieses Problem mit den von ihnen in Referenz [WSP04] vorgestellten „Light Space Perspective Shadow Maps“ beseitigt, indem sie die perspektivische Transformation und Division in Abhängigkeit vom Lichtkoordinatensystem durchgeführt haben.
Das Shadow Mapping wird mittlerweile von einem Großteil der Graphikhardware unterstützt[14]. Der Schattentest inklusive Texturzugriff, sowie das Setzen der Fragmentfarbe entsprechend des Schattentest-Resultats, werden dem Entwickler dadurch von der Graphikkarte abgenommen und führt zu einer deutlichen Beschleunigung des Algorithmus bei Verwendung entsprechender Hardware. Für detailiertere Informationen zum Einsatz des Shadow Mappings auf der Graphikhardware sei auf [Kil01] verwiesen.

4.4 Bewertung

Im Zuge der Recherche wurden Verfahren vorgestellt, mit denen einerseits die Umweltbeleuchtung mittels Fahrzeugscheinwerfer, andererseits der dadurch verursachte Schattenwurf realisiert werden könnte. Die bereits herausgestellten Vor- und Nachteile der Beleuchtungs- und Schattensimulationsverfahren werden im Folgenden im Hinblick auf deren potentiellen Einsatz in der Fahrsimulation bewertet. Unter dem Gesichtspunkt einer möglichst realistischen Simulation wird anschließend untersucht, ob und wie diese Verfahren mit der vorgestellten photometrischen Beleuchungsberechnung kombiniert werden können.
Nach wie vor stellen die globalen Beleuchtungsverfahren die Lichtsimulation mit den qualitativ hochwertigsten Resultaten dar. Neben den bereits vorgestellten Verfahren gibt es vielfältige, meist auf den Basisalgorithmen aufsetzende Versuche, die globale Beleuchtung auf echtzeitfähige Geschwindigkeiten zu beschleunigen. Als Beispiel genannt seien hier Tole et al. [TPWG02], die für bestimmte Samples der Szene deren Shadingwerte zwischenspeichern (*cachen*), um sie bei erforderlichem Update GPU-basiert interpolieren zu können, oder Dmitriev et al. [DBMS02], die ein Photon Mapping ähnliches Verfahren in interaktiven Szenen einsetzen. Trotz großer Forschungsaktivitäten in diesem Bereich ist eine wirklich echtzeitfähige Simulation globaler Beleuchtung derzeit noch nicht möglich. Zwar werden teilweise bereits interaktive Frameraten bis hin zu etwa $40fps$ für komplexe Szenen erreicht. Doch ist keines der Implementierungen globaler Beleuchtungsmodelle imstande, dynamische Szenen mit dynamischen Lichtquellen mit $60fps$ zu rendern, wie es für die Fahrsimulation erforderlich ist. Auf Basis der heute zur Verfügung stehenden technischen Möglichkeiten, ergibt sich aus der Recherche die Einschränkung,

[14] bspw. SGI RealityEngine, InfiniteReality, nVIDIA ab GeForce3 u.w.

eine auf einem lokalen Beleuchtungsmodell basierende Lichtsimulation zu realisieren. Die lokale Beleuchtungssimulation in OpenGL ist eine auf Ebene der Vertices realisierte Beleuchtung und damit abhängig von der Triangulierung der Szene. Um bestehende Datenbasen in der BMW Fahrsimulation weiter nutzen zu können, ist es jedoch Ziel, eine von der Geometrie der Szene unabhängige Lichtsimulation zu entwickeln (vgl. Realisierungsanforderungen in Tabelle 5.1). Der Einsatz von OpenGL zur Beleuchtung scheitert zum einen an der Geometrieabhängigkeit und zum anderen an der sehr unrealistischen Beleuchtungsdarstellung.
Für die Simulation der Beleuchtung durch Fahrzeugscheinwerfer kommen von denen im Laufe der Recherche vorgestellten Beleuchtungsverfahren daher nur die Beleuchtung mittels dreidimensionaler oder projektiver Texturen in Betracht. Der Speicherplatzbedarf der Verfahren und deren Möglichkeit, die Helligkeitsabnahme realistisch darzustellen, stehen sich bei den beiden Verfahren gegenüber. Eine realistische Lichtverteilung durch eine dreidimensionale Textur darzustellen ist auf jeden Fall einfacher und präziser. Doch wie die in Kapitel 4.1.3 beschriebenen Verfahren zeigen, sind auch durch projektive Beleuchtung mit simulierter Lichtabnahme gute Resultate zu erzielen. Ob der Nachteil der etwas eingeschränkten Qualität der projektiven Texturen, durch den Vorteil des geringeren Speicherbedarfs kompensiert wird, oder ob die Verwendung dreidimensionaler Texturen zu bevorzugen ist, kann erst nach Betrachtung der Anforderungen an eine Realisierung in Kapitel 5.1 entschieden werden. Neben dieser Abwägung darf jedoch der größte Nachteil der in dieser Form beschriebenen Algorithmen, nämlich der zwingend erforderliche zweite Renderingpass nicht vernachlässigt werden. Eine große Anforderung an die Umsetzung eines der beiden Verfahren stellt daher deren Realisierung in nur einem Renderingpass dar.
Bei der Gegenüberstellung der beschriebenen Schattenberechnungsverfahren und deren Eignung für den Einsatz in der Fahrsimulation scheiden die planaren Schatten aufgrund ihrer Beschränkung auf planare Flächen aus. Abzuwägen ist daher zwischen dem geometriebasierten Shadow Volume- und dem bildbasierten Shadow Mapping-Verfahren. Da die Performance in der Fahrsimulation durch die Integration eines Schattenverfahrens, die beide einen zweiten Renderingpass erfordern, sehr beeinträchtigt wird, ist die Performance der Verfahren mit einer höheren Priorität zu betrachten als deren Darstellungsqualität. Aus diesem Grund für komplexe Szenen der Fahrsimulation das Shadow Mapping den Shadow Volumes vorzuziehen, da der große Aufwand der Schattenvolumenberechnung nicht mit der etwas besseren Schattendarstellung gerechtfertigt werden kann. Auch die Erzeugung von Penumbra-Schattenregionen ist durch dessen Geometrieunabhängigkeit einfacher durch das Shadow Mapping zu realisieren.
Bei der Betrachtung des Prinzips der projektiven Beleuchtung im Hinblick auf die skizzierte photometrische Beleuchtungsberechnung wird deutlich, dass durch deren Kombination eine effiziente und photometrisch basierte per-pixel Beleuchtung möglich wäre. Dazu kann für jedes Fragment mittels projektiver Texturierung der entsprechende Beleuchtungswert der Messtextur bestimmt, und daraus, entsprechend des in Kapitel 4.2 beschriebenen Vorgehen, dessen Beleuchtungsstärke berechnet werden.
Um abschließend entscheiden zu können, wie die Beleuchtungssimulation am sinnvollsten für die BMW Fahrsimulation realisiert werden kann, wird im nächsten Kapitel eine Anforderungsliste an die Realisierung erstellt. Diese ermöglicht zusammen mit den Erkenntnissen der Bewertung die Definition eines Realisierungskonzeptes.

Kapitel 5

Realisierung

Als Ausgangspunkt für die Realisierung einer Beleuchtungssimulation dient in diesem Kapitel zunächst eine Anforderungsliste, die die gewünschten Ergebnisse und Prioritäten auflistet. Anhand dieser Anforderungen ist es möglich, eine geeignete Vorgehensweise zur Realisierung der Beleuchtungssimulation aus den beschriebenen Verfahren herauszufiltern, und in einer prototypischen Implementierung zu evaluieren. Im Anschluss daran fließen die Erfahrungen dieser Implementierung in die Entwicklung eines Realisierungskonzeptes zur Integration von Licht- und Schattensimulation in SPIDER ein.

5.1 Anforderungsliste

Da das Thema Simulation der Umweltbeleuchtung durch Fahrzeugscheinwerfer in der Fahrsimulation ein größeres Themengebiet darstellt als es im Rahmen dieser Diplomarbeit realisiert werden kann, wurde eine priorisierte Anforderungsliste erstellt. In der nachfolgenden Tabelle sind einerseits die Anforderungen aufgelistet, die im Zuge dieser Arbeit erreicht werden sollen, und andererseits jene, die weniger relevant sind. Ausschlaggebend dafür sind Vorgaben seitens BMW. Bei den relevanten Anforderungen entspricht die Position des jeweiligen Punktes auch seiner Priorität.

	Anforderungen an eine Realisierung:
+	Keine Performer spezifische Lösung
+	Realisierung in einem Renderingschritt (Ziel: 60 fps)
+	Bisherige Datenbasen sind weiterhin nutzbar
+	Von Channel und Kamera unabhängige Realisierung
+	Lichtsimulation ist auf Fremdfahrzeuge übertragbar
+	Möglichst realitätsnahe Darstellung der Scheinwerferlichtkegel
+	Eine nachträgliche Integration der ALC-Funktionalität ist möglich
	folgende Einschränkungen werden dafür akzeptiert:
-	Die Schattensimulation wird nicht realisiert
-	Eine photometrisch absolut korrekte Lichtsimulation ist nicht erforderlich
-	Alle Flächen werden als diffus betrachtet

Tabelle 5.1: Anforderungsliste Realisierung

5.2 Prototypische Implementierung

Die gesammelten theoretischen Erkenntnisse über die echtzeitfähige Beleuchtungssimulation fließen in diesem Kapitel in eine prototypische Implementierung ein. Zunächst wird das Implementierungsskonzept skizziert und anschließend dessen Realisierung erläutert.

5.2.1 Implementierungskonzept

Wie die Recherche ergeben hat, bieten sich zwei verschiedene Vorgehen zur Simulation der Beleuchtung an: die projektive und die dreidimensionale texturbasierte Beleuchtung. Welches der beiden sich besser für diesen Zweck eignet, ist anhand ihrer Vor- und Nachteile schwer auszumachen. Die Tatsache jedoch, dass die Lichtverteilungen der Scheinwerfer, wie bereits in Kapitel 4.2 beschrieben, in Form von gemessenen Lichtbündelquerschnitten vorliegen, hat schließlich die Entscheidung zugunsten des projektiven Beleuchtungsverfahren herbeigeführt. Da die Messdaten nur zweidimensional zur Verfügung stehen, müssen sie für eine Verwendung in dreidimensionaler Form durch eine Berechnung approximiert werden. Die Qualität der dreidimensionalen Texturen ist somit nicht größer als die der zweidimensionalen projektiven Texturen und ihr Einsatz aufgrund des hohen Speicherbedarfs nicht gerechtfertigt.

Die bisher in der BMW Fahrsimulation integrierte Beleuchtungssimulation, die mit der Technik der projektiven Texturen realisiert wurde, lieferte zwar bereits qualitativ gute Ergebnisse, hatte jedoch wie bereits beschrieben einen entscheidenden Nachteil: ihre Performer-Abhängigkeit. Die prototypische Implementierung der Beleuchtungssimulation erfolgt daher mittels OpenGL Shading Language (vgl. dazu Kapitel 5.2.2) direkt auf der Graphikkarte. Die entwickelten Shader können bei Bedarf auch von anderen Systemen angesteuert werden, solange sie die benötigten Informationen übergeben bekommen. Durch die GPU-Programmierung wird zudem versucht, die CPU zu entlasten. Der Vertex-Shader ist vor allem für die Ermittlung der projektiven Texturkoordinaten und der erforderlichen Szeneninformationen verantwortlich. Durch ein veränderbares Lichtfrustum wird die Möglichkeit gewahrt, nachträglich ein adaptives Kurvenlicht in die Simulation zu integrieren. Soweit möglich werden Informationen, die SPIDER an die Shader sendet, bereits in der Form übermittelt, in der sie dort benötigt werden. Zur Realisierung wird das Multitexturing der Graphikkarten genutzt, um mehrere Texturen in einem Renderingpass verarbeiten zu können.

Damit die realisierte Beleuchtungssimulation unabhängig von der verwendeten Datenbasis bleibt, wird die eigentliche Beleuchtung im Fragment-Shader auf Basis der Fragmente durchgeführt. Mit diesem Vorgehen wird sichergestellt, dass die Tesselierung der Szene keinen Einfluss auf die Qualität der Beleuchtung hat.
Um eine realitätsnahe Beleuchtungssimulation gewährleisten zu können, werden Teilaspekte der in Kapitel 4.2 erläuterten photometrischen Beleuchtung berücksichtigt. In die Bestimmung der Beleuchtungsstärke des aktuellen Fragments geht die Distanz zur Lichtquelle ein, um den tatsächlichen Beleuchtungswert aus dem vorhandenen Wert der projektiven Textur zu bestimmen. Als weitere Lichtquelle werden die Szenenhelligkeit (Sonne) und emissive Flächen berücksich-

tigt. Um die daraus berechneten Beleuchtungswerte qualitativ zufriedenstellend auf den Wertebereich zwischen 0 und 1 abbilden zu können, wird ein einfaches Tone Mapping realisiert werden. In Tabelle 5.2 sind die Eigenschaften der zu realisierenden prototypischen Implementierung zusammengefasst. Die Realisierung erfolgt mit Hilfe von Shadern. Dadurch ist die Beleuchtungs-

Basisverfahren	projektive texturbasierte Beleuchtung
Art der Beleuchtung	per-fragment
Realisierung mittels	OpenGL Shading Language Multitexturing
Berücksichtigung von	Kamera- und Channel-Unabhängigkeit Lichtdistanz Szenenhelligkeit emissiven Flächen einstellbarem Lichtfrustum

Tabelle 5.2: Eckpunkte der prototypischen Beleuchtungssimulation

simulation klar in zwei Aufgabenbereiche getrennt. Zunächst muss der Vertex-Shader für eine korrekte Bestimmung der projektiven Texturkoordinaten sorgen. Mit diesen wird anschließend im Fragment-Shader die Helligkeit aus der Lichttextur bestimmt, die, unter Berücksichtigung der Distanz zur Lichtquelle und der Szenenhelligkeit, zur Berechnung der Fragmentbeleuchtung dient.

5.2.2 Shader-Programmierung in SPIDER

Die Möglichkeiten der Integration der Shader-Programmierung in SPIDER sind entscheidend an die entsprechende Unterstützung der Shadersprachen seitens Performer gebunden.

Die von SGI Performer zum Zeitpunkt der Programmierungsphase verfügbare Version 3.1.1 unterstützt eine GPU-Programmierung, die sich auf die Verwendung der OpenGL Extensions `GL_ARB_vertex_program` und `GL_ARB_fragment_program` beschränkt. Deren Nutzung ist seit Performer 3.1 durch die Zustandselemente `pfVertexProgram` und `pfFragmentProgram` und die Klasse `pfGProgramParms` möglich. Die OpenGL Extensions erlauben jedoch weder eine intuitive Programmierung, noch ist deren Einsatz auf lange Sicht sinnvoll, da sie eine Programmierung der Graphikkarte nur in einer an Assembler angelehnten Sprache ermöglichen. Sehr viel erfolgversprechender ist die Verwendung von High-Level Shading Sprachen wie HLSL, Cg oder GLSL. Erstere ist die von *Microsoft* entwickelte High Level Shading Language, die im Zusammenhang mit *DirectX* eingesetzt wird und damit für die OpenGL basierte Fahrsimulationssoftware nicht in Frage kommt. Die zwei verbleibenden Alternativen sind daher die von *nVIDIA* entwickelte Sprache Cg (C for graphics) und die vom *OpenGL Architecture Review Board (ARB)* freigegebene OpenGL Shading Language (GLSL), die seit OpenGL 1.5 durch die Extensions `GL_ARB_shader_objects`, `GL_ARB_vertex_shader`, `GL_ARB_fragment_shader` sowie

`GL_ARB_shading_language_100` verfügbar ist. In beiden Fällen ist eine Integration in Performer derzeit nur über Umwege möglich. Der Cg Compiler ist in der Lage, aus Cg Vertex- und Fragment-Shadern `pfVertexProgram` und `pfFragmentProgram` konformen Programmcode zu generieren. Es besteht also die Möglichkeit, die entsprechenden Shader in Cg zu programmieren, und den generierten Assemblercode in die Performer Zustandselemente `pfVertexProgram` und `pfFragmentProgram` zu übernehmen. Um die Nutzung von GLSL in Performer zu ermöglichen, ist es derzeit noch erforderlich auf OpenGL direkt zurückzugreifen. Dazu können innerhalb von Performer Callbacks an Pipes, Channels oder Knoten angehängt werden. Diese können beliebigen OpenGL Code enthalten und können damit auch genutzt werden, um die Vertex- und Fragment-Shader zu kompilieren und auf der Graphikkarte zu installieren.
Beide Vorgehensweisen sind nicht besonders elegant, doch in der zum Zeitpunkt der Erstellung dieser Arbeit vorhandenen Performer Version stellen sie die einzige sinnvolle Möglichkeit dar, GPU Programmierung in SPIDER zu integrieren. Zwei Tatsachen haben letztlich dazu geführt, dass die Entscheidung zwischen den beiden High Level Shading Sprachen zugunsten von GLSL ausgefallen ist:
Erstens wurde vom OpenGL ARB im September 2004 die OpenGL 2.0 Spezifikation veröffentlicht. Mit Erscheinen dieser Version ist die OpenGL Shading Language und deren API von einer Extension zu einer Kernfunktionalität von OpenGL geworden. Dadurch ist in Zukunft mit einer weiten Verbreitung von GLSL zu rechnen, was es lohnenswert macht, trotz der genannten Schwierigkeiten schon jetzt auf diese Sprache zu setzen. Obwohl die Problematik der Shader Integration in Performer damit noch nicht gelöst ist, machen zweitens Aussagen seitens SGI Hoffnung, dass dies nur ein temporäres Problem bleibt: Mit der für Anfang 2005 angekündigten neuen Version 3.2 soll die OpenGL Shading Language in Performer direkt unterstützt werden.
Anmerkung: Noch während der Fertigstellung der Diplomarbeit im Dezember 2004 wurde die neue Performer Version 3.2 vorgestellt. Damit wird die OpenGLShading Language jetzt offiziell mittels eigener Klassen in Performer unterstützt. Damit hat sich im Nachhinein die Entscheidung für GLSL als Programmiersprache als richtig erwiesen (vgl. dazu auch Kapitel 5.3).

Die Kompilierung und Installation der Shader erfolgt während der Initialisierung von SPIDER in einem Pipe-Callback. Durch die genannten ARB-Extensions stellt die OpenGL API alle dafür notwendigen Methoden zur Verfügung. Zur Implementierung werden die Extensions zu OpenGL 1.5 verwendet. Nach der Überprüfung der Graphikkarte auf GLSL-Unterstützung werden drei `GLhandleARB`-Variablen erzeugt: zwei als sogenannte Shaderobjekte und eine als Programmobjekt. Für die Shaderobjekte wird zunächst der Vertex- und Fragment-Shader Quellcode geladen und kompiliert. Die Shaderobjekte werden nach erfolgreicher Kompilierung an das Programmobjekt (`programObject`) angehängt und dieses wird gelinkt. Durch das Linken des Programms wird ein gültiges GLSL Programm erzeugt und die benutzerspezifischen Variablen werden initialisiert. Mit dem Aufruf

```
glUseProgramObjectARB(programObject);
```

wird das Programm schließlich auf der Graphikkarte installiert. Ab diesem Zeitpunkt ist die Fixed Function Pipeline, wie in Kapitel 3.2 beschrieben, für bestimmte Transformationen deaktiviert und stattdessen müssen die Shader diese Funktionalität ersetzen. Jeder Zugriff auf die Shader läuft in SPIDER über das Programmobjekt ab, welches daher als Attribut eines von überall zugänglichen Objektes verfügbar gemacht wird. Um Informationen an die Shader übergeben

zu können, müssen im Shader Variablen vom Typ `uniform` deklariert sein. Dieser Variablentyp ermöglicht den Datenaustausch zwischen Programm und Shader und ist für Daten gedacht, die sich in unregelmäßigen Abständen ändern können. Neben den `uniform`-Variablen gibt es zwei weitere Arten von Variablen. Für Daten, die sich bei jedem Vertex ändern, werden `attribute`-Variablen verwendet. Mit Hilfe dieser Variablen können Attribute eines Vertex an den Shader gesendet werden. Variablen hingegen, die der Vertex-Shader an den Fragment-Shader übergibt, sogenannte `varying`-Variablen, durchlaufen auf ihrem Weg zwischen den beiden Shadern die Fixed-Function-Pipeline, die für deren Interpolation sorgt.
Ein Wert `value` vom Datentyp `int` wird wie folgt an eine uniform Variable `var` gesendet:

```
GLint location = glGetUniformLocationARB(programObject, "var");
glUniform1iARB(location, value);
```

Entsprechend gibt es auch Möglichkeiten, andere einfache Datentypen oder auch Vektoren und Matrizen an die Shader zu übergeben. Damit ist die Kommunikation zwischen SPIDER und Shadern sichergestellt. Als nächstes wird ermittelt, welche Informationen an welcher Stelle und zu welchem Zeitpunkt in SPIDER ermittelt und an die Shader gesendet werden müssen, damit diese die Beleuchtungssimulation durchführen können.

5.2.3 Schnittstellen zu SPIDER

Die Beleuchtungsberechnung, die in den Shadern realisiert wird, setzt einige Informationen zur vorliegenden Beleuchtungssituation voraus. Zum einen müssen Geometrieinformationen, wie Kamera-, Fahrzeug- und Scheinwerferposition und -ausrichtung, ermittelt und an den Vertex-Shader gesendet werden, um die projektiven Texturkoordinaten korrekt und unabhängig von der Kameraposition berechnen zu können. Ferner benötigt der Fragment-Shader Informationen über die Beleuchtung und außerdem Zugriff auf die verwendeten Textureinheiten.

Bei der Kommunikation zwischen SPIDER und den Shadern sind zwei Unterschiede zwischen der Datenorganisation von Performer und GLSL zu berücksichtigen, damit die von SPIDER ermittelten Daten auch in der Form gesendet werden, in der sie die Shader interpretieren:

Koordinatensysteme: Die Koordinatensysteme, die von Performer und GLSL (OpenGL) verwendet werden, liegen zwar beide im Rechtssystem vor, werden jedoch jeweils unterschiedlich interpretiert. Während bei Performer die y-Achse im Bezug auf die Kamera nach vorn und die z-Achse nach oben gerichtet ist, ist die Kamera in OpenGL entlang der negativen z-Achse ausgerichtet, während die y-Achse nach oben zeigt. Dies entspricht bildlich einer 90 Grad Rotation um die x-Achse. Diese Tatsache muss bei Transformationen beachtet werden. Um aus einer Performer Transformation die äquivalente OpenGL Transformation zu erhalten, müssen die y- und z-Komponenten der Transformation vertauscht und die neue z-Komponente anschließend negiert werden.

Matrizen: Bei der Übergabe einer Matrix von SPIDER an GLSL muss auf deren Elementanordnung geachtet werden. Während Performer die Matrizen in Reihenordnung verwaltet, werden Matrizen in OpenGL spaltenweise abgelegt. Eine vierdimensionale Performer-Matrix(`pfMatrix`) in Reihenordnung wird in Form eines Arrays (`GLfloat[16]`) an den

Shader übergeben und dort als Matrix in Spaltenordnung interpretiert (`uniform mat4`). Die Matrix muss daher in transponierter Form übergeben werden.

Neben der Art der zur Beleuchtungssimulation benötigten Daten spielt auch der Zeitpunkt eine Rolle, zu dem diese Daten verfügbar sind oder benötigt werden. Während bei einigen Daten die einmalige Übergabe an den Shader ausreicht, ist bei anderen Informationen eine ständige Aktualisierung notwendig.
In Tabelle 5.3 sind die von den beiden Shadern benötigten Daten mit deren Datentyp[1], Verwendungszweck und erforderlicher Aktualisierungsfrequenz (Update) aufgelistet. Auf die Berechnung der verschiedenen Matrizen wird im folgenden Kapitel eingegangen.

Benötigte Daten	**Datentyp**	**Verwendungszweck**	**Update**	**Shader**
inverse View-Transformation	`mat4`	Transformation von View- in Weltkoordinaten	pro Frame	VS
Fahrzeug-Transformation	`mat4`	Transformation von Welt- in Fahrzeugkoordinaten	pro Frame	VS
Licht-Transf. (L+R)	`mat4`	Transformation von Fahrzeug- in Lichtkoordinaten	pro Frame	VS
Licht-Projektion	`mat4`	Transformation von Licht- in Licht-Clip-Koordinaten	bei Änderung	VS
Licht-Texturen (L+R)	`sampler2D`	Zugriff auf Licht-Textureinheiten	bei Änderung	FS
Modell-Texturen	`sampler2D`	Zugriff auf Modell-Textureinheit	bei Änderung	FS
Tageszeit	`float`	Bestimmung der Szenenhelligkeit	bei Änderung	FS

Tabelle 5.3: Zur Beleuchtungssimulation an die Shader gesendete Daten

5.2.4 Berechnung der projektiven Texturkoordinaten

Das Prinzip der Beleuchtung mittels projektiver Texturen besteht, wie bereits in Kapitel 4.1.3 beschrieben, aus der Ermittlung der Texturkoordinaten mittels Projektion. Die Berechnung der projektiven Texturkoordinaten aus Sicht der Scheinwerfer stellt die wesentliche Aufgabe des Vertex-Shaders dar. Diese Koordinaten lassen sich für den aktuellen Vertex (`gl_Vertex`) entsprechend Gleichung 4.1 berechnen.

[1]Notiert als GLSL Datentypen Matrix, Float und Sampler. Sampler ist ein Datentyp, der den Zugriff auf eine Textureinheit ermöglicht.

Leider ist eine Berechnung auf diesem direkten Wege in der prototypischen Realisierung für SPIDER nicht möglich, sondern kann nur über Umwege stattfinden. Der erste Grund liegt darin, dass die Modelltransformationen in GLSL nur in Form der Modelview Matrix (`gl_ModelViewMatrix`) und nicht als Model Matrix verfügbar sind. Die Berechnung der projektiven Texturkoordinaten auf Basis des Viewkoordinatensystems ist jedoch wenig sinnvoll, da die Berechnung dadurch nicht blickunabhängig wäre und der Lichtkegel für alle View- und Channeleinstellungen gleich berechnet würde. Aus diesem Grund werden die Viewinformationen aus der Modelview Matrix entfernt, indem sie mit der inversen View Matrix von links multipliziert wird. Dadurch ist es möglich, die Model Matrix zu isolieren und den aktuellen Vertex in Weltkoordinaten zu transformieren.
Der zweite Grund für notwendige Zusatzberechnungen ist darin zu sehen, dass die Koordinaten der Scheinwerfer im Fahrzeugmodell ermittelt werden und daher in Modellkoordinaten vorliegen. Mit Hilfe der Transformationsmatrix des Fahrzeugs kann jedoch eine Umrechnung ins Weltkoordinatensystem erfolgen. Durch die unabhängige Berechnung der Beleuchtung für beide Frontscheinwerfer ist es notwendig, die projektiven Texturkoordinaten aus Sicht beider Lichtquellen zu berechnen. Im Vertex-Shader müssen demnach zwei Berechnungen folgender Art durchgeführt werden:

$$\begin{pmatrix} s \\ t \\ r \\ q \end{pmatrix} = \begin{pmatrix} Texture \\ Indices \\ Matrix \end{pmatrix} * \begin{pmatrix} Light \\ Projection \\ Matrix \end{pmatrix} * \begin{pmatrix} Light \\ View \\ Matrix \end{pmatrix} * \begin{pmatrix} Car \\ Transf. \\ Matrix \end{pmatrix} * \begin{pmatrix} View \\ Matrix \\ Inverse \end{pmatrix} * \begin{pmatrix} Model \\ View \\ Matrix \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{5.1}$$

Jeder Vertex wird in die Koordinaten der Scheinwerfer transformiert, projiziert und mit der *Texture Indices Matrix* in den Wertebereich der Texturen gemappt (vgl. dazu die in Gleichung 4.1 auf Seite 28 angegebene Matrix). Bevor diese Berechnungen durchgeführt werden können, müssen zunächst die benötigten Matrizen in SPIDER ermittelt und an den Vertex-Shader übergeben werden.

View Matrix Inverse Die inverse View Matrix wird benötigt, um die Model Matrix aus der Modelview Matrix zu bestimmen. Sie setzt sich aus den Betrachterinformationen[2] (`Driver`) und dem Channel Offset zusammen und wird in der SPIDER-Klasse `Channel` ermittelt. Diese Klasse übernimmt die Einstellungen und die Aktualisierung der Performer Channels und ist daher geeignet, während des Channel-Updates die berechnete inverse View Matrix an den Vertex-Shader zu senden.
Mit Hilfe von SPIDER-Methoden wird zunächst die Betrachterposition und dessen Ausrichtung in Weltkoordinaten bestimmt und als dreidimensionale Vektoren `driverPos` und `driverDir` gespeichert. Die Ermittlung der Translations- und Rotationsinformationen des Channel Offsets erfolgt entsprechend.
Durch die Multiplikation der Channel Offsetmatrix von der rechts an die Driver Transformationsmatrix wird die inverse View Matrix bestimmt. Diese beiden Matrizen werden jeweils aus

[2] Der Betrachter repräsentiert den Fahrer des Eigenfahrzeugs in der Simulation.

der Multiplikation der entsprechenden Translation und Rotation zusammengesetzt und ergeben in folgender Reihenfolge die inverse View Matrix:

$$\begin{pmatrix} View \\ Matrix \\ Inverse \end{pmatrix} = \begin{pmatrix} Driver \\ Translation \\ Matrix \end{pmatrix} * \begin{pmatrix} Driver \\ Rotation \\ Matrix \end{pmatrix} * \begin{pmatrix} ChannelOffset \\ Translation \\ Matrix \end{pmatrix} * \begin{pmatrix} ChannelOffset \\ Rotation \\ Matrix \end{pmatrix} \tag{5.2}$$

Bei dieser Berechnung ist darauf zu achten, dass die Transformationen, wie in Kapitel 5.2.3 auf Seite 45 beschrieben, entsprechend des OpenGL-Koordinatensystems zusammengesetzt werden. Da es sich um eine inverse Transformation handelt, wird die Reihenfolge von Rotation und Translation vertauscht, die Rotation wird somit jeweils von rechts mit der entsprechenden Translationsmatrix multipliziert[3]. Das nachfolgende Codefragment stellt die Berechnung der inversen View Matrix dar. Zur Berechnung der Rotationsmatrizen wird ein dynamisches Koordinatensystem (`pfDCS`) von Performer verwendet, das deren Erstellung vereinfacht:

```
pfMatrix driverRotMat, offsetRotMat;
pfMatrix driverTransMat  = pfMatrix (1.0,0.0,0.0, driverPos[0],
                                     0.0,1.0,0.0, driverPos[2],
                                     0.0,0.0,1.0,-driverPos[1],
                                     0.0,0.0,0.0, 1.0);
pfDCS* driverDCS = new pfDCS();
driverDCS->setRot( driverDir[2],  driverDir[1], -driverDir[0]);
driverDCS->getMat(driverRotMat);

[... analog: offsetTransMat, offsetRotMat ...]

pfMatrix viewMatInv = driverTransMat * driverRotMat * offsetTransMat
                    * offsetRotMat;
```

Die so bestimmte inverse View Matrix muss nun transponiert werden, um aus einem Reihenvektor einen Spaltenvektor zu bilden. Dies geschieht mit Hilfe der Performer Methode
`pfMatrix::getCol(int col,float *x,float *y,float *z,float *w)`
für jede Spalte (`col`) der Matrix `viewMatInv`. Als Parameter werden der Methode die entsprechenden Elemente eines Arrays übergeben. Das Array wird anschließend an den Vertex-Shader übergeben, wie in Kapitel 5.2.2 erläutert.
Da sich durch die Bewegung des Fahrzeugs auch die Kameraposition entsprechend verändert, muss die inverse View-Transformation einmal pro Frame aktualisiert werden.

Car Transformation Matrix Nach der Anwendung der inversen View Matrix liegt der aktuelle Vertex in Weltkoordinaten vor. Um diesen in das Koordinatensystem des Fahrzeugs zu transformieren, werden die Fahrzeugtransformationen benötigt. Die Klasse `SimCarModel` ist in SPIDER für die Funktionalität des Eigenfahrzeugs zuständig und verfügt über eine Methode zur

[3]Die resultierenden Matrizen können eigentlich auch direkt aus Rotations- und Translationsinformationen zusammengesetzt werden - aus Analogiegründen zu den folgenden Berechnungen wird die Matrixmultiplikation hier jedoch mit aufgeführt.

Aktualisierung der Fahrzeugdaten. Die Berechnung der Car Transformation Matrix, die aus den Weltkoordinaten des Fahrzeugs und dessen Ausrichtung bestimmt wird, ist daher in diese Methode integriert worden. Da es sich bei dieser Matrix, im Gegensatz zur vorher berechneten inversen View Matrix, nicht um eine inverse Transformation handelt, gehen in die Matrixberechnung die jeweils umgekehrten[4] Transformationen des Fahrzeugs ein. Die Reihenfolge von Rotation und Translation sind demnach vertauscht:

```
pfMatrix carRotMatR;
pfMatrix carTransMatR = pfMatrix (1.0,0.0,0.0,-carPos[0],
                                  0.0,1.0,0.0,-carPos[2],
                                  0.0,0.0,1.0, carPos[1],
                                  0.0,0.0,0.0, 1.0);
pfDCS *carDCS  =  new pfDCS();
carDCS->setRot = (-carDir[2], -carDir[1], carDir[0]);
carDCS->getMat(carRotMatR);

pfMatrix simCarMat =  carRotMatR * carTransMatR;
```

Light View Matrix In der BMW Fahrsimulation ist jedes Fahrzeug bereits mit Scheinwerferinformationen versehen. Abhängig von LOD Einstellungen werden weiter entfernte Scheinwerfer durch Performer Punktlichtquellen, Scheinwerfer in kürzerer Distanz zum Betrachter durch emissive Polygone dargestellt.
Bei der Initialisierung von SPIDER werden die Fahrzeugmodelle durch den *Flight-Loader* in Performer geladen und an den Szenengraph angefügt. An dieser Stelle besteht über den Szenengraph Zugriff auf den Fahrzeug-Szenengraph, und die Lichtquellen des Fahrzeugs können in diesem ermittelt werden. Im Anhang 7 ist ein solcher Fahrzeug-Szenengraph schematisch dargestellt.
Neben den Modellkoordinaten der beiden Frontscheinwerfer wird deren Ausrichtung benötigt, um die Light View Matrix berechnen zu können. Eine Möglichkeit wäre, die Normale der Lichtquellen zu berücksichtigen. Da es jedoch einerseits wünschenswert ist, eine Integration von adaptivem Kurvenlicht zu integrieren und andererseits die Scheinwerfer abblenden zu können, wird die Ausrichtung durch eine steuerbare Rotationsmatrix repräsentiert. Dadurch ist es später möglich, die Ausrichtung der Scheinwerfer zu regulieren. Diese Flexibilität macht es jedoch auch erforderlich, die Light View Matrix pro Frame zu aktualisieren – wird die ALC-Unterstützung jedoch nicht gewünscht, genügt es, die Matrix nur bei Änderung der Scheinwerferausrichtung zu aktualisieren. Die Berechnung der Light View Matrix des linken Scheinwerfers (analog gilt dies auch für den rechten Scheinwerfer) mit Lichtquellenposition `leftLPos` und gewünschter Rotation der Scheinwerfer um `leftLRot` ergibt sich wie folgt:

```
pfMatrix leftLRotMatR;
pfMatrix leftLTransMatR = pfMatrix (1.0,0.0,0.0,-leftLPos[0],
                                    0.0,1.0,0.0,-leftLPos[2],
                                    0.0,0.0,1.0, leftLPos[1],
                                    0.0,0.0,0.0, 1.0);
```

[4] Mit R markierte Matrizen stellen die jeweils umgekehrte Transformation dar.

```
pfDCS *leftLDCS  =  new pfDCS();
leftLDCS->setRot = (-leftLRot[2], -leftLRot[1], leftLRot[0]);
leftLDCS->getMat(leftLRotMatR);

pfMatrix leftLViewMat =  leftLRotMatR * leftLTransMatR;
```

Light Projection Matrix Die letzte noch zu berechnende Matrix beschreibt die benötigte Projektion. Damit die Texturkoordinaten dem Seitenverhältnis der Lichttextur entsprechend korrekt berechnet werden können, ist die Einstellung des Frustums von entscheidender Bedeutung. Um ein Clipping an der *near*-Ebene (n) des Frustums zu vermeiden, wird diese sehr klein gewählt. Die horizontalen und vertikalen Begrenzungen des Frustums (rechts = r, links = l, oben = t, unten = b), lassen sich aus der Distanz der *near*-Ebene und dem Tangens der Öffnungswinkel berechnen, zusätzlich wird eine *far*-Ebene (f) festgelegt. Die Lichtprojektion wird mit Hilfe der perspektivischen Matrix wie folgt ermittelt und anschließend transponiert an den Vertex-Shader übergeben:

```
pfMatrix lightProjMat = pfMatrix (2*n/(r-l), 0.0, (l+r)/(r-l),  0.0,
                                  0.0, 2*n/(t-b), (b+t)/(t-b),  0.0,
                                  0.0,       0.0,-(n+f)/(f-n), -1.0,
                                  0.0,       0.0,        -1.0,  1.0);
```

Berechnung durch den Vertex-Shader Sobald die Matrizen in SPIDER ermittelt sind, werden sie an die entsprechenden `uniform`-Variablen des Vertex-Shaders gesendet. Der Programmcode des im Folgenden erläuterten Vertex-Shaders ist aus Seite 51 einzusehen. Der Vertex-Shader nutzt die übermittelten Matrizen zur Bestimmung der projektiven Texturkoordinaten `projPosL` und `projPosR` gemäß Gleichung 5.1 (vgl. Zeilen 22 bis 26 im Code).
Damit der Fragment-Shader im Anschluss die Beleuchtung korrekt berechnen kann, wird im Vertex-Shader zusätzlich die Distanz der Lichtquellen zu dem aktuellen Vertex ermittelt und als `distanceL` beziehungsweise `distanceR` an den Fragment-Shader gesendet, da dieser keinen Zugriff auf die Geometrie hat (Zeilen 28,29). Die Bestimmung der emissiven Materialfarbe in Zeile 34 hingegen müsste eigentlich nicht im Vertex-Shader erfolgen, wird aber aufgrund eines Fehlers des verwendeten Graphikkartentreibers schon vom Vertex-Shader ermittelt und anschließend an den Fragment-Shader übergeben.
Da die Fixed Function Pipeline durch den Einsatz der Shader umgangen wird, muss der Vertex-Shader außerdem deren Aufgaben übernehmen. Dazu gehört zum einen die Transformation des eingehenden Vertex (`gl_Vertex`) in Clip-Koordinaten und andererseits die Berechnung der Texturkoordinaten. Für letzteres steht in GLSL eine Variable zur Verfügung, die Zugriff auf die Standard Texturkoordinaten von OpenGL hat. In Zeile 36 wird mit Hilfe dieser Variablen (`gl_MultiTexCoord0`), die Texturkoordinate dieses Vertex für die Textureinheit 0 festgelegt. Die Berechnung der Clip-Koordinaten erfolgt anhand der intern verfügbaren `gl_ModelViewProjectionMatrix`. Diese Matrix transformiert den Vertex und speichert ihn in der Variablen `gl_Position` (Zeile 38). Das Speichern der Clipkoordinaten in dieser Variablen stellt zugleich die einzige Operation dar, die zwingend im Vertex-Shader erfolgen muss.

```
const GLcharARB *vertexCode = {

        uniform mat4 viewMatInv;        // View-Matrix Kamera
        uniform mat4 simCarMat;         // Transformationsmatrix Fzg.
        uniform mat4 leftLViewMat;      // Transformationsmatrix linkes Licht
        uniform mat4 rightLViewMat;     // Transformationsmatrix rechtes Licht
        uniform mat4 lightProjMat;      // Licht-Projektionsmatrix

        varying vec4 emissiveColor;     // emissive Vertexfarbe
        varying float distanceL;        // Entfernung linke Lichtquelle zu Vertex
        varying float distanceR;        // Entfernung rechte Lichtquelle zu Vertex
        varying vec4 projPosL;          // projektive Texturkoordinate (aus Sicht linkes Licht)
        varying vec4 projPosR;          // projektive Texturkoordinate (aus Sicht rechtes Licht)

        // Matrix zur Skalierung und Translation der Texturkoordinaten:
        const mat4 scaleTranslMat=mat4( 0.5,0.0,0.0,0.0,
                                        0.0,0.5,0.0,0.0,
                                        0.0,0.0,0.5,0.0,
                                        0.5,0.5,0.5,1.0);

        void main(void){
                // Koordinaten:     Automobil<-Welt<-Kamera<-Modell
                vec4 cc_vertex = vec4(simCarMat * viewMatInv * gl_ModelViewMatrix * gl_Vertex);
                // Koordinaten:     Licht<-Automobil
                vec4 lc_vertexL = vec4(leftLViewMat     *   cc_vertex);
                vec4 lc_vertexR = vec4(rightLViewMat    *   cc_vertex);
                // Koordinaten:     Clip<-Licht
                projPosL        = vec4(scaleTranslMat * lightProjMat * lc_vertexL);
                projPosR        = vec4(scaleTranslMat * lightProjMat * lc_vertexR);
                // Berechnung der Entfernungen Licht-Vertex:
                distanceR = length(lc_vertexL);
                distanceL = length(lc_vertexL);
                // Ermittlung der emissiven Materialfarbe:
                emissiveColor = gl_FrontMaterial.emission;
                // Texturkoordinatenbestimmung:
                gl_TexCoord[0]  = gl_MultiTexCoord0;
                // Rendering Pipeline:
                gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
        }
};
```

Abbildung 5.1: *Programmcode:* Vertex-Shader

5.2.5 Fragment-Beleuchtung

Die Fragment-Beleuchtung stellt nach der Berechnung der Texturkoordinaten den zweiten Teil der realisierten Beleuchtungssimulation dar. Sie ist sowohl für die Berechnung der Beleuchtung, als auch für eine optimierte Darstellung der berechneten Werte verantwortlich.

Beleuchtungsberechnung Die Beleuchtungsberechnung findet mit Hilfe der vom Vertex-Shader gesendeten und von der Fixed Function Pipeline (vgl. Abbildung 3.1) interpolierten `varying`-Variablen, den von SPIDER geladenen Texturen und der Szenenhelligkeit statt. In SPIDER werden dazu die Lichttexturen der beiden Scheinwerfer an zwei verschiedene Textureinheiten gebunden, damit der Fragment-Shader in einem Renderingpass auf beide Texturen Zugriff[5] hat. Mit Hilfe von Samplern (`sampler2D`), speziellen uniform-Variablen, besteht im Shader Zugriff auf die Textureinheiten. Der Fragment-Shader (vgl. im Folgenden den Quellcode auf Seite 54) der prototypischen Implementierung verwendet drei Sampler: ein Sampler für den Zugriff auf die Textureinheit, welche die Szenentexturen speichert (`decalMap`) und zwei weitere für den Zugriff auf die Textureinheiten, an die die Lichttexturen gebunden sind (`lightMapL` und `LightMapR`). Innerhalb der SPIDER-Klasse `Sun` wird zudem die „time of day“ Einstellung ermittelt. Diese vom GUI aus regelbare Größe wird an die Variable `uniform tod` des Fragment-Shaders gesendet, um die Beleuchtung durch eine Sonnenlichtquelle mit in die Berechnung einfließen zu lassen.
Die Beleuchtung eines Fragmentes wird durch die beiden Scheinwerfer, die Sonne und durch die Materialeigenschaften bestimmt. GLSL ermöglicht den Zugriff auf verschiedene Materialeigenschaften. Aufgrund der Beschränkung auf diffuse Materialien, geht jedoch nur die emissive Farbe des Fragmentes in die prototypische Realisierung ein. Diese orientiert sich an der vereinfachten Renderingequation, die in Gleichung 4.4 auf Seite 30 dargestellt ist. Die Summe des Lichts, das von verschiedenen Lichtquellen auf das aktuelle Fragment einfällt, wird mit dem Reflexionskoeffizient multipliziert und auf das emissive Licht addiert. Im Gegensatz zu dem, in der Renderingequation beschriebenen Vorgehen, geht jedoch das emmissive Licht in der implementierten Beleuchtungsberechnung in der gleichen Weise in die Berechnung ein wie die anderen Lichtquellen auch. Der Grund dafür liegt in den Materialeigenschaften der Straßenschilder der Szene begründet: Zur Imitation der stark reflektierenden Oberfläche der Schilder, sind diese innerhalb der verwendeten Datenbasen als emissive Objekte modelliert. Eine Addition der emissiven Lichtfarbe auf die Texturfarbe der Schilder, würde unweigerlich zu einer Farbverfälschung führen, da die Schilder nicht in der Farbe Licht ausstrahlen, in der sie texturiert sind. Das emissive Licht wird daher auf das Licht der anderen Lichtquellen addiert und anschließend mit der Materialfarbe multipliziert.
Zunächst werden jedoch die Texturfarbe des Fragmentes aus der Textureinheit 0 (`decalMap`), und die Beleuchtungsstärken aus den Textureinheiten 1 und 2 mit Hilfe der projektiven Texturkoordinaten bestimmt. Bei der Ermittlung des Beleuchtungswertes, muss die homogene Koordinate

[5] Es ist ausreichend, für beide Scheinwerfer die gleiche Textur zu wählen und diese an nur eine Textureinheit zu binden. Um einen direkten Vergleich verschiedener Scheinwerfer zu ermöglichen, werden in dem hier dargestellten Vertex-Shader jedoch zwei getrennte Textureinheiten genutzt.

w überprüft werden: Ist deren Wert kleiner 0, so befindet sich das aktuelle Fragment hinter der Lichtquelle und demnach findet hier keine Beleuchtung durch den Scheinwerfer statt (vgl. Zeilen 28 bis 34). Anschließend kann entsprechend Gleichung 4.7 auf Seite 33 die Lichtstärke des Lichtes bestimmt werden, das vom Scheinwerfer in Richtung des Fragmentes ausgestrahlt wird (Zeilen 46,47). Dies geschieht durch Multiplikation der Beleuchtungsstärke mit der quadrierten Messdistanz. Die mögliche Lichtabschwächung durch den Lichteinfallswinkel wird dabei im Folgenden zur Vereinfachung ignoriert.
Während der Lichteinfallswinkel vernachlässigt wird, geht der Abstand zur Lichtquelle (`distanceL`, `distanceR`) mit in die Berechnung ein, um die Lichtabnahme zu berücksichtigen. Die Lichtabnahme wird dabei für das emissive Licht weniger groß gewählt als für das Licht der Scheinwerfer, um dadurch den reflektierenden Effekt der Straßenschilder zu simulieren. Das emissive Licht wird so berechnet, dass es ausgehend von der Position der Scheinwerfer mit zunehmender Distanz linear abnimmt und ab einer vorgegebenen Entfernung (im Shader bspw. 500m) den Wert 0 erreicht hat. Negative Werte die bei der Berechnung entstehen können, werden auf den Wert 0 abgebildet (Zeile 39). Die Beleuchtung der Scheinwerfer hingegen nimmt, wie die Zeilen 41 und 42 zeigen, entsprechend Gleichung 4.7 quadratisch ab. Um einen zu starken Anstieg der Beleuchtung bei einer Distanz kleiner einem Meter zu vermeiden, wird der Nenner jeweils um 1 erhöht.

Darstellung der Beleuchtungswerte Bevor die Farbe des Fragmentes abschließend festgelegt werden kann, müssen die berechneten Beleuchtungswerte noch addiert und auf den Bereich $[0, 1]$ gemappt werden. Das emissive Licht wird im dargestellten Fragment-Shader getrennt von den restlichen Lichtquellen behandelt, da es mit einer anderen Lichtabnahme betrachtet wird und bereits im gewünschten Bereich vorliegt. Die Beleuchtungswerte der Scheinwerfer und der Sonne müssen hingegen noch mittels Tone Mapping in den Wertebereich von 0 bis 1 skaliert werden. Das Tone Mapping erfolgt durch einen *URQV*-ToneMapper (vgl. Kapitel 4.2.2, Seite 36). Ausschlaggebend für eine qualitativ gute Darstellung der Beleuchtung ist die Wahl der Steuervariablen `measureDist2` (quadrierte Messdistanz), `s` (Skalierungsfaktor Tone Mapper), `refLight` (Referenz Lichtquelle = stärkste mögliche Beleuchtung) und `sun_max` (maximale Beleuchtung durch die Sonnenlichtquelle). Die Messdistanz ist in der Regel bei allen Messungen identisch und wird daher nur selten geändert werden müssen. Auch das Sonnenlichts kann, je nach gewünschtem Einfluss der Sonne, konstant innerhalb des Programmcodes festgelegt werden. Es ist hingegen sinnvoll, den Skalierungsfaktor des Tone-Mappers und die Helligkeit der Referenzlichtquelle vom GUI aus zu steuern, um die Beleuchtungssituation den gewünschten Verhältnissen anpassen zu können. Um einen Vergleich verschiedener Scheinwerfertypen zu ermöglichen, sollte die Referenzhelligkeit entsprechend des hellsten Scheinwerfers eingestellt werden. Wird jedoch nur ein bestimmter Scheinwerfertyp eingesetzt, ist es ratsam dessen maximale Beleuchtungsstärke als Referenz zu verwenden. Bei dessen Festlegung muss die eventuell vorliegende Skalierung der Werte innerhalb der Textur beachtet werden. Der Skalierungsfaktor des Tone Mappers lässt sich am besten empirisch den aktuellen Verhältnissen anpassen, sollte jedoch bei einem Vergleich mehrerer verschiedener Scheinwerfer konstant bleiben, da er die Abstrahlcharakteristik der Scheinwerfer verändert.

```
const GLcharARB *fragmentCode = {

        uniform sampler2D decalMap;      // Zugriff auf Textureinheit 0
        uniform sampler2D lightMapL;     // Zugriff auf Textureinheit 1
        uniform sampler2D lightMapR;     // Zugriff auf Textureinheit 2
        uniform float tod;               // time of day

        // Vom VS gesendete, von der FFP interpolierte Variablen:
        varying vec4 projPosL, projPosR, emissiveColor, distanceL, distanceR;

        const float measureDist2, s, refLight, sun_max;   // Variable Steuerparameter, diese müssen
                                                          // hier initialisiert, oder von SPIDER als
                                                          // uniform Variablen gesandt werden!
        float sun = tod * sun_max;
        float light_max = refLight + sun_max;
        float lightL = 0.0; lightR = 0.0;
        vec4 textureColor;

        // URQV ToneMapper:
        float toneMapper(in float light){
                return clamp( (s * light)/(s * light - light + light_max), 0.0, 1.0 );
        }

        void main(void){
                // Texturfarbe des Fragmentes bestimmen:
                textureColor = texture2D(decalMap,gl_TexCoord[0].st);
                // Beleuchtungsmesswert links bestimmen (nur vor Scheinwerfer):
                if (projPosL.w >= 0) {
                        lightL = texture2DProj(lightMapL,projPosL).r;
                }
                // Beleuchtungsmesswert rechts bestimmen (nur vor Scheinwerfer):
                if (projPosR.w >= 0) {
                        lightR = texture2DProj(lightMapR,projPosR).r;
                }
                // Leuchtstärke berechnen:
                float I_left  = lightL * measureDist2;
                float I_right = lightR * measureDist2;
                // emissives Licht mit der Distanz veringern
                vec4 c_emissive = vec4(emissiveColor * clamp((1.0-distanceR*0.002),0.0,1.0));
                // einfallendes Licht (ohne Gewichtung durch Winkel) berechnen:
                float l_incident = toneMapper(  I_left/(1.0 + (distanceL * distanceL))
                                              + I_right/(1.0 + (distanceR * distanceR)) + sun);
                vec4 c_incident = vec4(l_incident, l_incident, l_incident, 1.0);
                // Fragmentfarbe festlegen
                gl_FragColor = textureColor * clamp(c_incident + c_emissive, 0.0, 1.0);
        }
};
```

Abbildung 5.2: *Programmcode:* Fragment-Shader

Die Beleuchtungsstärken durch die Scheinwerfer und die Sonne, die für das aktuelle Fragment bestimmt wurden, werden nun mittels *URQV*-Tone Mapper entsprechend Gleichung 4.15 auf Seite 36 auf den Wertebereich $[0, 1]$ abgebildet (vgl. Zeile 20). Um sicherzustellen, dass der Tone Mapper auch dann einen Wert zwischen 0 und 1 zurückgibt, wenn die maximal mögliche Beleuchtung nicht korrekt eingestellt wurde, wird der Rückgabewert zusätzlich, wie in Zeile 21 dargestellt, mittels `clamp(...,0.0,1.0)` auf den Wertebereich $[0, 1]$ begrenzt.
Nachdem das auf das Fragment einfallende Licht in Form eines Vektors gespeichert wurde (Zeile 43), liegt das emissive und das einfallende Licht jeweils in Form eines Vektors mit einem Komponentenwertebereich von 0 bis 1 vor. Um die endgültige Fragmentfarbe zu erhalten, werden die beiden Vektoren addiert und mit der Texturfarbe multipliziert. Werte größer 1 werden auf 1 abgebildet. Dieses Vorgehen ignoriert bewusst das Verhältnis zwischen der berechneten einfallenden Beleuchtung und dem emissiven Licht, denn auch die anderen, während der Beleuchtungssimulation nicht beachteten Lichtquellen der Szene (beispielsweise die Scheinwerfer anderer Fahrzeuge), stellen emissive Flächen dar. Um diese, unabhängig von den Scheinwerfern des Eigenfahrzeugs, immer in der gleichen Helligkeit im Verhältnis zur Distanz darstellen zu können, wird das emissive Licht in Zeile 45 ohne jegliche Skalierung auf den berechneten Wert addiert und wie bereits beschrieben auf den Bereich von 0 bis 1 beschränkt.

5.3 Entwicklung eines Realisierungskonzeptes für SPIDER

Die Entwicklung eines Realisierungskonzeptes zur Integration der Beleuchtungssimulation in SPIDER setzt die Betrachtung der Ergebnisse der prototypischen Realisierung im Hinblick auf Performance und Qualität voraus. Diese Erkenntnisse sind Teil der Ergebnisevaluation in Kapitel 6, gehen jedoch auch als Teil der Realisierung in die Konzeptentwicklung für die SPIDER-Integration ein. Das Realisierungskonzept besteht aus vier Kernpunkten:

1. **GPU Programmierung:**
   Wie kann die Shader-Funktionalität am sinnvollsten in SPIDER genutzt werden und worauf ist bei der Realisierung zu achten?

2. **Beleuchtungssimulation:**
   Kann die Integration der implementierten Beleuchtungssimulation in dieser Art und Weise zukünftig auch in SPIDER erfolgen, oder müssen Einschränkungen gemacht werden?

3. **Integration Fremdverkehr:**
   Ist es möglich die realisierte Beleuchtungssimulation auf Fremdfahrzeuge zu übertragen – und wenn ja – welche Qualitätseinbußen können zugunsten der Performance in Kauf genommen werden?

4. **Schatten:**
   Wie sollte die Integration eines Schattenverfahrens in SPIDER realisiert werden?

Diese vier Fragestellungen beinhalten die wichtigsten Komponenten, die bei einer Integration in SPIDER berücksichtigt werden müssen. In den nachfolgenden vier Kapiteln wird daher im Einzelnen auf diese Kernpunkte eingegangen.

GPU Programmierung

Während der Implementierungsphase dieser Arbeit haben sich die Vertex- und Fragment-Shader als sehr mächtiges Werkzeug erwiesen, bei dessen Verwendung jedoch einige Punkte beachtet werden müssen.
Wie die Ergebnisse der Arbeit zeigen, ist eine Integration der Shader-Funktionalität in SPIDER möglich. Mit der Veröffentlichung der Performer Version 3.2 werden nun zwei neue Klassen zur Verfügung gestellt, die die Shader-Programmierung mittels GLSL unterstützen. Der Programmcode von Vertex- und Fragment-Shadern wird jeweils als Objekt der Klasse `pfShaderObject` in Form eines Strings gespeichert. Mehrere Shader Objekte können anschließend zu einem Objekt der Klasse `pfShaderProgram` zusammengefasst und über das `pfGeoStateState`-Attribut `PFSTATE_SHADPROG` als Zustandselement verwendet werden. Die GLSL-Funktionalität kann dadurch in SPIDER realisiert werden, ohne auf OpenGL zurückgreifen zu müssen. Außerdem ist eine Speicherung des Programm Objektes in Form eines Attributes nicht mehr nötig. Ein Problem entsteht dadurch, dass in GLSL keine Zugriffsmöglichkeit auf das `TEXTURE_ENV_MODE` Attribut besteht. Dadurch kann bei der Texturierung nicht entschieden werden, wie die Fragmentfarbe mit der Texturfarbe verrechnet werden muss, da kein Zugriff auf die Texturfunktion

besteht. Dies kann durch die Verwendung einer `attribute`-Variablen gelöst werden. Innerhalb der Applikation muss der Status des `TEXTURE_ENV_MODE` Attributes ermittelt und an den Shader gesendet werden, damit dieser Informationen über die auszuführende Texturfunktion hat. Unter Berücksichtigung der in Kapitel 5.2.3 beschriebenen Unterschiede in der Verwaltung von Matrizen und in der Interpretation der Koordinatensysteme zwischen Performer und GLSL, können die Shader entsprechend der prototypischen Realisierung in SPIDER eingesetzt werden.

Beleuchtungssimulation

Die in Form einer prototypischen Implementierung realisierte Beleuchtungssimulation hat sich, wie Kapitel 6 genauer erläutert, qualitativ zwar als sehr gut erwiesen, ohne drastische Einschränkungen ist dieses Vorgehen jedoch nicht für die Fahrsimulation zu realisieren. Vor allem die vorgestellte Beleuchtung auf Fragmentebene muss optimiert werden. Soweit möglich sollten Berechnungen vom Fragment- auf den Vertex-Shader verlagert werden, um die Frequenz der benötigten Berechnungen zu verringern.
Soweit auf eine ALC-Funktionalität verzichtet werden kann, bietet sich die Beschränkung auf eine Lichtquelle an. Dazu kann aus den Texturen der beiden Scheinwerfer eine gemeinsame Textur berechnet werden, mit der die Lichtbündelquerschnitte der Scheinwerfer gleichzeitig in die Szene projiziert werden kann. Dadurch ist nur eine projektive Texturkoordinate und eine Distanz zu berechnen. Die Überprüfung der homogenen Koordinate fällt zudem nur einmal an und die Beleuchtungswerte der beiden Scheinwerfer müssen weder addiert noch auf den Wertebereich $[0.0, 1.0]$ abgebildet werden, da sie schon korrekt in der Textur codiert sind. Wie die Performancetests gezeigt haben sind solche Einschränkungen auf jeden Fall notwendig, um die Bildwiederholrate von $60 fps$ erreichen zu können.

Integration Fremdverkehr

Zwar kann die realisierte Beleuchtungssimulation generell auf jedes Fahrzeug übertragen werden, der Aufwand der bereits für die Beleuchtungssimulation des Eigenfahrzeugs benötigt wird, lässt jedoch wenig Spielraum für deren Verwendung für den Fremdverkehr. Die Beleuchtungssimulation für den Fremdverkehr muss daher noch mehr auf die Performance ausgerichtet sein als die des Eigenfahrzeugs.
Besonders von Bedeutung ist in diesem Zusammenhang, für welche Fahrzeuge der Szene die Beleuchtungssimulation durchgeführt wird und für welche nicht. Dabei muss der Extremfall berücksichtigt werden, bei dem der Lichtkegel des Fahrzeugs sichtbar ist, obwohl sich dieses wiederum außerhalb des Sichtbereiches befindet. Durch das Clipping an den Seiten des View Frustums wird das Fahrzeug ignoriert und damit hat auch dessen Licht kein Einfluss mehr auf die Szene. Eine mögliche Lösung dieses Problems stellt das Anlegen einer *Bounding Box (BB)* dar, die den Lichtkegel des Fahrzeugs und den Scheinwerfer umgibt. Durch deren Integration in den Fahrzeugszenengraph wird das Clipping des Fahrzeugscheinwerfers in dem beschriebenen Fall verhindert. Dies stellt zwar ein Nachteil in der Performance dar, die Bounding Box wird jedoch nur so groß wie unbedingt nötig gewählt, damit dieser Fall nicht zu häufig auftritt. Ob ein Fahrzeug mit einer Beleuchtung versehen wird, kann nun anhand der Entfernung der Bounding

Box zur Betrachterposition entschieden werden.
Nachdem die Fahrzeuge bestimmt wurden, deren Lichtkegel dargestellt werden soll, muss den Shadern deren Anzahl und Transformationsmatrizen übermittelt werden. Mit der Transformationsmatrix des Fahrzeugs kann die Beleuchtungssimulation analog der des Eigenfahrzeugs realisiert werden. Aufgrund der Performanceprobleme ist dies jedoch wohl nicht ohne weitere drastische Einschränkungen möglich. Dies könnten beispielsweise sein:

- Die Berechnung der Scheinwerferbeleuchtung weiter entfernte Fahrzeuge auf Vertex Ebene. Dadurch würde in diesen Fällen nur eine Berechnung innerhalb des Vertex-Shaders erfolgen. Die Beleuchtungswerte müssten als `varying` Variablen an den Fragment Shader gesendet werden, damit sie entsprechend interpoliert werden.
- Verwendung nur einer projektiven Texturkoordinate pro Fahrzeug. Dies bedeutet, dass die Beleuchtung nur für einen bestimmten Scheinwerfer erfolgt und dadurch Berechnungen eingespart werden können.
- Verzicht auf die Umrechnung in das Lichtkoordinatensystem. Dadurch kann eine Transformation der Koordinaten eingespart werden. Da die Berechnungen von Matrizen jedoch sehr performant erfolgt, ist der zu erwartende Effekt nicht besonders hoch.
- Keine Distanzberechnung, sondern Imitation der Lichtabnahme durch Absenkung der Scheinwerfer. Dadurch kann auf die kostspielige Berücksichtigung der Lichtabnahme verzichtet werden. Solange sich das Fahrzeug in relativ ebenem Gelände befindet, ist der optische Fehler der dadurch entsteht vertretbar. Alternativ kann, im Bezug auf das Lichtkoordinatensystem, die z-Koordinate des aktuellen Vertex durch die die homogene Koordinate w dividiert werden, um ein angenähertes Distanzmaß zu erhalten.

Ob diese Maßnahmen ausreichen, um die Beleuchtungssimulation auch für andere Fahrzeuge zu ermöglich, ist angesichts der ermittelten Frameraten fraglich. Realistisch abgeschätzt werden kann dies jedoch erst nach einer allgemeinen Optimierung der Beleuchtungssimulation. Der Einsatz der Lichtsimulation in Szenen mit Fremdverkehr ist generell nur in Verbindung mit Nebel zu empfehlen, der ab einer bestimmten Entfernung einzusetzen beginnt. Dadurch kann verhindert werden, dass das Umschalten zwischen verschiedenen LOD-Stufen durch plötzlich erscheinende Lichtkegel sichtbar ist.

Schatten

Wie bereits in Kapitel 4.4 beschrieben, bietet sich das Shadow Mapping als Verfahren zur Schattenberechnung in SPIDER an. Der Aufwand, der für eine Schattenbestimmung im Shader entsteht, ist minimal – die Erstellung der Shadow Map hingegen kostet einen zusätzlichen Renderingpass und erschwert dadurch die Integration eines Schattenverfahrens in SPIDER. Wird eine Schattensimulation gewünscht und kann ein zweiter Renderingpass in Kauf genommen werden, ist das Shadow Mapping wie folgt in SPIDER realisierbar:
Zunächst muss entschieden werden, welche Lichtquellen bei der Schattenberechnung berücksichtigt werden. Aus Performancegründen ist es nicht zu empfehlen, mehr als eine Lichtquelle

zu wählen, weil jede zusätzliche Lichtquelle auch einen zusätzlichen Renderingpass erfordert. Um den Schattenwurf der Scheinwerfer zu simulieren, ist es sinnvoll, die Shadow Map aus Sicht einer imaginären Lichtquelle in der Mitte der beiden Frontscheinwerfer zu erstellen.
Ist eine Shadow Map erstellt worden, wird im zweiten Schritt eine geringe Erweiterung des auf Seite 54 dargestellten Fragment-Shaders notwendig. Diesem muss ein zusätzlicher Sampler (`sampler2DShadow`) übergeben werden, wie im Codefragment in Abbildung 5.3 für den linken Scheinwerfer zu ersehen ist. Der Sampler kann genutzt werden, um auf Texturen mit dem internen OpenGL Format `GL_DEPTH_COMPONENT` zuzugreifen. In SPIDER muss sichergestellt werden, dass die erstellte Shadow-Map in diesem Format an den Shader gesendet wird, andernfalls ist der Zugriff auf den Sampler undefiniert. Um zu überprüfen, ob das aktuelle Fragment im Schatten liegt oder nicht, muss anschließend nur noch der Vergleich des Tiefenwertes der Shadow Map mit dem der projektiven Texturkoordinaten aus Sicht des linken Scheinwerfers erfolgen.
Die GLSL Methode `shadow2DProj()` übernimmt den Test und liefert als Ergebnis 0, falls das Fragment im Schatten des linken Scheinwerfers liegt. Durch Multiplikation des Wertes mit dem Beleuchtungsswert aus der Lichttextur geht der Einfluss des Scheinwerfers auf das aktuelle Fragment wie gewünscht verloren.
Sollten Schatten aus Performancegründen nicht für Fremdfahrzeuge realisiert werden, ist trotzdem nicht damit zu rechnen, dass dies einem Betrachter übermäßig störend auffällt. Diese Annahme liegt darin begründet, dass der von anderen Fahrzeugen verursachte Schattenwurf ohnehin nur im näheren Umfeld zu sehen ist. In der kurzen Entfernung jedoch, in der dem Betrachter ein fehlender Schatten störend auffallen könnte, liegt dieser bereits im Scheinwerferkegel des Eigenfahrzeugs und wird dadurch abgeschwächt.

```
uniform sampler2DShadow shadowMap;

float inShadow = shadow2DProj(shadowMap,projPosL).r;
float I_left   = lightL * measureDist2 * inShadow;
```

Abbildung 5.3: *Programmcode:* Erweiterung des Fragment-Shaders um einen Schattentest

Kapitel 6

Ergebnisevaluation

In dieser Diplomarbeit wurde die Beleuchtung der Umwelt durch Fahrzeugscheinwerfer untersucht und in Form einer prototypischen Realisierung implementiert. Die dabei erzielten Ergebnisse und die aufgetretenen Probleme werden nachfolgend genauer erläutert. Dazu wird die realisierte prototypische Beleuchtungssimulation zunächst auf deren Performance und auf die Qualität der Beleuchtungssimulation hin überprüft. Die Ergebnisse werden anschließend den in Tabelle 5.1 aufgelisteten Anforderungen gegenübergestellt, die vor der Realisierung festgelegt wurden. Die Erkenntnisse dieser Gegenüberstellung sind auch in die Erstellung eines Implementierungskonzeptes für SPIDER in Kapitel 5.3 eingeflossen.

6.1 Performance

Die Performance des realisierten Beleuchtungsverfahrens stellt im Bezug auf deren mögliche Integration in SPIDER die wohl wichtigste Messgröße dar. Aus diesem Grund werden in diesem Kapitel die Ergebnisse eines Performancetests vorgestellt.
Als Testkonfiguration diente dabei eine Graphik-Workstation mit 2 *Xeon* Prozessoren und einer *nVIDIA Quadro FX1000* Graphikkarte. Die Simulation wurde in einer 1-Channel Konfiguration betrieben und war entsprechend des verwendeten Bildschirms auf 60 Hz beschränkt. Die Szenen der Datenbasis bestanden durchschnittlich aus 4670 Dreiecken. Um für jede Testfahrt die gleichen Voraussetzungen zur Verfügung stellen zu können, wurde eine Streckenführung für das Eigenfahrzeug und bestimmte Fremdfahrzeuge festgelegt, der diese selbständig folgen konnten. Die Ermittlung der Performance erfolgte anhand der durchschnittlichen Bildwiederholungsrate, die pro Sekunde ermittelt wurde (*fps*). Die ermittelten Werte geben jedoch nur bedingt Auskunft über die Performance der Implementierung, da diese einerseits nur in prototypischer Form vorhanden ist, und die Werte andererseits nur anhand einer bestimmten Konfiguration getestet worden sind.
Getestet wurde mit dieser Konfiguration zum einen die Fahrsimulationssoftware selbst (SPIDER) als Referenzgröße, eine Beleuchtung mittels projektiver Performer-Beleuchtung sowie die während dieser Diplomarbeit entwickelte Realisierung (GLSL). Als Referenzgeschwindigkeit

für die Shader Funktionalität dient außerdem ein Shader, der nur die FFP nachbildet. Die Shader der prototypischen Implementierung wurden mit der Intention entwickelt, eine möglichst gute Qualität der Simulation zu erreichen und darauf basierend bei Bedarf bestimmte Einschränkungen zugunsten der Performance vornehmen zu können. Die Ergebnisse der Performancetests sind in Tabelle 6.1 dargestellt. Aus den Ergebnissen geht sehr deutlich hervor, dass der Fragment-Shader das Bottleneck der Beleuchtungssimulation darstellt. Durch die dort ausgelassenen Frames erreicht die komplette Simulation auf der GPU lediglich eine Bildwiederholrate von $22fps$. Die umfangreichen Matrizenberechnungen im Vertex-Shader und deren Ermittlung in SPIDER haben in dieser Versuchskonfiguration keinen unmittelbar feststellbaren negativen Einfluss auf die Performance. Kleine Vereinfachungen im Fragment-Shader haben, aufgrund der sehr viel größeren Anzahl an Fragmenten im Vergleich zu den Vertices, eine stärkere Auswirkung. Allein der Verzicht auf den Tone Mapper erbrachte in Versuchskonfiguration 8 beispielsweise einen Performancegewinn von bis zu $8fps$. Die Bemühungen, die Beleuchtungssimulation auf $60fps$ zu beschleunigen, beschränkten sich daher auch hauptsächlich auf den Fragment-Shader. Zwar ist diese Marke auch mit dem auf Seite 63 dargestellten vereinfachten Fragment-Shader nicht erreicht worden, obwohl dieser weder eine Lichtabnahme mit der Distanz noch das Umgebungslicht berücksichtigen, doch ist das Performancepotential nach oben erkennbar. Durch eine weitere Optimierung der Shader, aber auch der Programmierung in SPIDER, ist zu erwarten, dass eine Bildwiederholrate von 60 Frames erreicht wird. Einen weiteren Performancegewinn könnte zudem der Einsatz der neuen GLSL Unterstützung von Performer 3.2 mit sich bringen.
Die realisierte Beleuchtungssimulation erreicht zwar mit vergleichbarer Darstellungsqualität nicht die Performance der Performer-Implementierung, sie zeigt aber eine Möglichkeit ein vergleichbares Verfahren ohne Performer-Abhängigkeit zu realisieren und kann somit durch eine optimierte Implementierung zu einer Alternative werden.

Nr.	Typ	Funktionalität/Einschränkungen	fps
1	SPIDER	Referenzkonfiguration ohne Beleuchtungssimulation	≥ 60
2	Performer	projektive Beleuchtung mittels Performer-Lichtquellen	≥ 60
3	GLSL	Die Shader übernehmen die Aufgaben der FFP: Transformation der Vertices in Clipkoordinaten, Texturkoordinatenbestimmung und Texture Mapping	≥ 60
4	GLSL	komplette Beleuchtungssimulation (vgl. Abb. 5.2.4 und 5.2.5)	≈ 22
5	GLSL	VS entsprechend Konfiguration 4, der FS übernimmt nur das Texture Mapping	≥ 60
6	GLSL	FS entsprechend Nr. 4, der VS übernimmt nur die Texturkoordinatenbestimmung und die ModelViewProjection Matrix und übergibt für benötgte `varyings` konstante Werte	≈ 22
7	GLSL	VS entsprechend Nr. 4, im FS keine Berücksichtigung des Sonnenlichts (`uniform tod`), die Überprüfung des Fragmentes auf die Position im Bezug zur Lichtquelle erfolgt nur auf Basis einer homogenen Koordinate	≈ 26
8	GLSL	komplette Beleuchtungssimulation entsprechend Konfiguration 4, das Tone Mapping wird jedoch durch Clipping der Werte auf den Bereich $[0, 1]$ ersetzt	≈ 30
9	GLSL	komplette Beleuchtungssimulation entsprechend Konfiguration 4, FS jedoch ohne Quadrierung der Distanz	≈ 25
10	GLSL	komplette Beleuchtungssimulation entsprechend Konfiguration 4, FS berücksichtigt jedoch bei der Berechnung nur die Distanz einer Lichtquelle	≈ 30
11	GLSL	Vereinfachte Beleuchtungssimulation: Der VS entspricht dem der Konfiguration 4, mit der Änderung, dass keine Distanz bestimmt wird und die emissive Farbe als `float`-Wert übergeben wird. Auch das Sonnenlicht wird ignoriert. Der Fragment-Shader ist in Abbildung 6.1 ersichtlich. Die Lichtabnahme ist nicht berücksichtigt, dies muss daher durch Absenkung der Scheinwerfer imitiert werden.	≈ 56

Tabelle 6.1: Ergebnisse des Performancetests

```
const GLcharARB *fragmentCodeSimple = {

        uniform sampler2D  decalMap, lightMapL, lightMapR;

        varying vec4 projPosL, projPosR;
        varying float emissiveCol;

        vec4 textureColor;
        float light = emissiveCol;

        void main(void){
                // Beleuchtungsmesswerte der Fragmente bestimmen,
                // falls diese vor rechter Lichtquelle liegt
                if (projPosR.w >= 0) {
                        light += texture2DProj(lightMapR,projPosR).r;
                        light += texture2DProj(lightMapL,projPosL).r;
                }
                // Texturfarbe des Fragmentes bestimmen
                textureColor = texture2D(decalMap,gl_TexCoord[0].st);
                gl_FragColor = textureColor * clamp(light,0.0,1.0);
        }
};
```

Abbildung 6.1: *Programmcode:* Vereinfachter Fragment-Shader

6.2 Qualität

Die visuelle Darstellung der realisierten Beleuchtungssimulation kann durch verschiedene Parameter verändert und an die jeweiligen Anforderungen angepasst werden. Durch die Integration eines Tone Mappers ist es gelungen, die visuelle Qualität der Beleuchtungssimulation zu erhöhen, indem die Darstellung der Beleuchtungswerte an die Leuchteigenschaften der Scheinwerfer angepasst werden kann. Durch die Wahl einer Referenzbeleuchtungsstärke ist es dadurch auch möglich, die Ausleuchtung der Straße durch verschiedene Scheinwerfertypen zu vergleichen. Die Abbildung 6.2 stellt eine typische SPIDER Simulationsszene bei Tageslicht dar. Die Beleuchtung einer solchen Szene bei Nacht wurde durch die prototypische Implementierung jeweils für Halogen- und Xenonscheinwerfer simuliert (vgl. Abbildungen 6.4 und 6.5). Durch die Angabe einer maximalen Beleuchtungsstärke als Referenzhelligkeit, ist es möglich, die zwei Scheinwerfertypen zu vergleichen. Der größte Vorteil des Tone Mappers ist, dass die berechneten Beleuchtungswerte bei Bedarf nicht-linear skaliert werden können. Dies kann die optische Qualität der Beleuchtungssdarstellung in den Fällen erhöhen, in denen die berechneten Beleuchtungswerte sehr weit auseinander liegen. Die in Kapitel 4.2.2 beschriebenen Probleme linearer Mappingverfahren können dadurch vermieden werden.
Das Tone Mapping erhöht jedoch andererseits die Komplexität der Realisierung, da es zwei weitere Parameter erfordert. Zudem kann es zu qualitativ weniger guten Ergebnissen kommen, wenn die maximale Beleuchtungsstärke für die Szene nicht richtig abgeschätzt werden kann. Die Qualität der Darstellung ist neben dem Tone Mapping vor allem von den verwendeten Texturen abhängig. Die Lichtbündelquerschnitte, die bei den Messungen ermittelt wurden, repräsentie-

ren jede gemessene Beleuchtungsstärke in Form eines `Float`-Wertes in Lux. Die zur Verfügung stehenden Lichttexturen, die daraus ermittelt wurden, liegen jedoch in Form von `Intensity`-Texturen mit einer Tiefe von 8 Byte (256 Grauwerte) vor. Daraus resultiert ein Verlust von Präzision, da nicht alle Werte in der vorliegenden Genauigkeit der Messwerte bei der Erstellung der Texturen berücksichtigt werden können. Um den verfügbaren Wertebereich so gut wie möglich zu verarbeiten, können die Werte entsprechend skaliert werden.
Die verwendeten Intensity-Texturen führen durch deren geringe Präzision und durch die niedrige Auflösung von 512 x 128 Pixeln (74dpi), in der sie vorliegen, zu Aliasingeffekten. Die Vernachlässigung des Lichteinfallswinkels wirkt sich nicht auffallend negativ auf die Ergebnisse der Beleuchtungssimulation aus, da die Scheinwerfer meist nur die Straße beleuchten und sich die Lichteinfallswinkel dort nicht stark voneinander unterscheiden.
Neben der Simulation der Beleuchtung müssen die Shader zusätzlich die Aufgaben der FFP übernehmen. Dies hat sich bei der Texturierung der Objekte als schwierig herausgestellt, weil die Shader, wie bereits in Kapitel 5.3 beschrieben, keinen Zugriff auf das OpenGL Attribut `TEXTURE_ENV_MODE` haben, welches die Texturfunktion spezifiziert. Dadurch kann nicht entschieden werden, wie die Fragmentfarbe mit der Texturfarbe verrechnet werden muss. Dies ist in den verwendeten Datenbasen nur selten zum Problem geworden, da dort fast alle Objekte texturiert sind, muss jedoch berücksichtigt werden, falls Datenbasen mit untexturierten Objekten verwendet werden.
Insgesamt liefert das realisierte Beleuchtungsverfahren qualitativ hochwertige Ergebnisse, wie die Abbildungen 6.3, 6.4, 6.5 und 6.7 zeigen. Auch das adaptive Kurvenlicht kann integriert werden, wie in Abbildung 6.8 angedeutet. Wie der Vergleich mit einer mittels projektiver Beleuchtung in Performer implementierten Simulation zeigt (vgl. Abbildungen 6.6 und 6.7), weist der Scheinwerferkegel bei der implementierten Beleuchtung eine realistischere Lichtverteilung auf und die Farben werden besser dargestellt. Die beschriebenen Aliasingprobleme lassen sich durch hochauflösendere Texturen mit größerer Farbtiefe beseitigen oder zumindest verringern.

Abbildung 6.2: SPIDER: Strassenszene bei Tageslicht

Abbildung 6.3: Beleuchtungssimulation aus Sicht des Gegenverkehrs

Abbildung 6.4: Beleuchtung durch Halogen Scheinwerfer

Abbildung 6.5: Beleuchtung durch Xenon Scheinwerfer

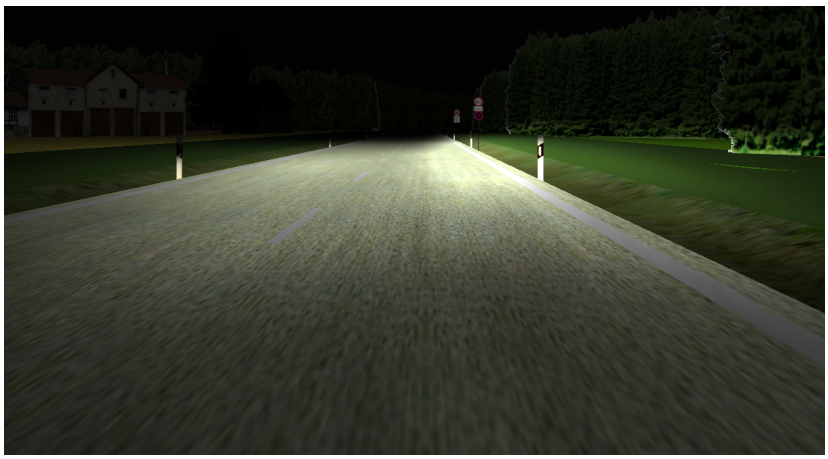


Abbildung 6.6: Beleuchtungssimulation mittels Performer Lichtquellen

Abbildung 6.7: Prototyische Realisierung der Beleuchtungssimulation

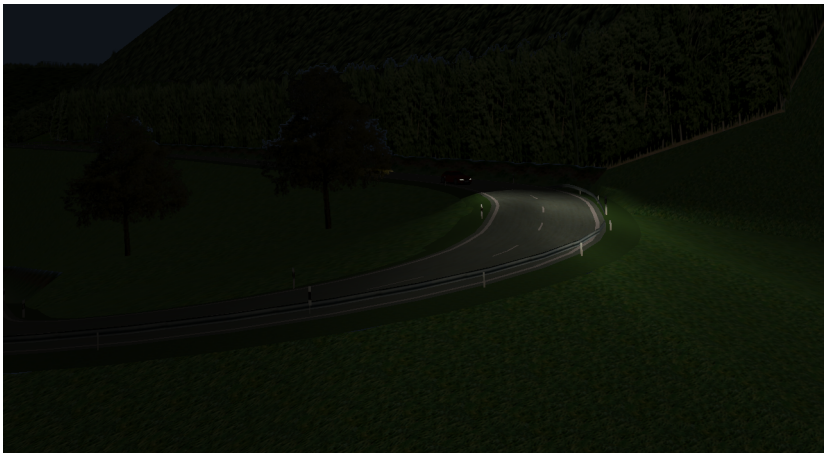


Abbildung 6.8: Integrationsmöglichkeit ALC

6.3 Erreichte Ziele

Als Ausgangspunkt für die prototypische Realisierung der Beleuchtungssimulation diente die Erstellung einer Anforderungsliste. Zum Abschluss wird diese nun zur Ergebnisevaluation herangezogen, um die geforderten Ziele den erreichten Ergebnissen gegenüberzustellen.

Anforderungen		**Ergebnisse**
1 Renderingschritt	✔	Durch Ausnutzung des Multitexturings kann die Beleuchtungssimulation in einem Renderingschritt berechnet werden.
60 fps	(✘)	Bei einer Beleuchtungssimulation mit qualitativ hochwertiger Darstellung konnten nur $22 fps$ erreicht werden. Durch verschiedene Einschränkungen kann die Performance auf bis zu $56 fps$ gesteigert werden.
Datenbasen weiter nutzbar	(✔)	Die Beleuchtungssimulation ist unabhängig von der verwendeten Datenbasis. Datenbasen die nicht durchgehend texturiert sind, werden jedoch noch nicht richtig dargestellt.
Nicht Performer spezifisch	✔	Die Shader sind nicht abhängig von Performer. Ein anderes System müsste lediglich die gleichen Informationen an die Shader senden.
Channel- & Kameraunabhängigkeit	✔	Durch die Berechnung der projektiven Texturkoordinaten auf Basis der Weltkoordinaten, ist keine entsprechende Abhängigkeit mehr gegeben.
Auf Fremdfahrzeuge übertragbar	✔	Die Beleuchtungssimulation ist für jedes beliebige Fahrzeug realisierbar, solange dies die Performance erlaubt.
Realitätsnahe Lichtkegel	(✔)	Die Lichtverteilung auf der Strasse wird gut dargestellt, die Darstellung kann jedoch, abhängig von den verwendeten Texturen, durch Aliasingeffekte beeinträchtigt werden.
ALC Integration möglich	✔	Die Ansteuerung der Scheinwerfer ist von SPIDER aus möglich.

Ziel erreicht?

✘ = nein, (✘) = teilweise, (✔) = weitestgehend, ✔ = ja

Tabelle 6.2: Gegenüberstellung der Anforderungen und Ergebnisse der Realisierung

Kapitel 7

Fazit und Ausblick

Ziel dieser Arbeit war die Entwicklung eines Modells zur Integration der Umweltbeleuchtung durch Fahrzeugscheinwerfer in die BMW Fahrsimulationssoftware SPIDER. Als Basis diente dazu sowohl die Recherche über echtzeitfähige Beleuchtungs- und Schattenberechnungsverfahren, als auch eine Betrachtung der photometrischen Hintergründe. In Form einer prototypischen Implementierung konnte die Beleuchtungssimulation mittels projektiver Texturen auf der Graphikkarte realisiert und daraus ein Integrationskonzept für SPIDER entwickelt werden.
Während der prototypischen Implementierung sind einige Probleme bei der Integration der Graphikkartenprogrammierung innerhalb von SPIDER aufgetreten, die jedoch durch OpenGL-Callbacks gelöst werden konnten. Dies war die erforderliche Grundlage, um die Beleuchtungssimulation auf der GPU umzusetzen und damit eine von Performer unabhängige Lösung zu erhalten.
Den größten Nachteil der realisierten Beleuchtungssimulation stellt die erreichte Performance dar. Wie die Tests in Kapitel 6.1 zeigen, liegt die Performance der Realisierung in der kompletten Funktionalität nur bei etwa 22 Frames pro Sekunde in der Testkonfiguration. Die geforderten 60 Frames konnten somit nicht erreicht werden. Einschränkungen im Bereich der Darstellungsqualität der Beleuchtungssimulation führen jedoch, wie Tabelle 6.1 zeigt, zur Beschleunigung der Shader und zu Frameraten von etwa 58 fps bei einer vereinfachten Beleuchtung. Durch die beschriebenen Vereinfachungen der Beleuchtungsberechnung und durch Optimierung der Anbindung der Shader an SPIDER, ist eine weitere Performancesteigerung zu erwarten. Ferner bleibt abzuwarten, ob und wieviel die Performance durch die Nutzung von Performer 3.2 gesteigert werden kann.
Die geforderte Darstellungsqualität konnte – losgelöst von den Performanceproblemen – erreicht werden. Die angesprochenen Aliasingeffekte können durch die Erstellung geeigneter Texturen behoben beziehungsweise vermindert werden. Obgleich der Lichteinfallswinkel in der Beleuchtungsberechnung nicht berücksichtigt wird, liefert das Verfahren, abgesehen von den genannten texturbedingten Artefakten, qualitativ hochwertige Resultate und stellt eine flexiblere und realistischere Beleuchtungssimulation dar, als sie die Nutzung der projektiven Performer-Lichtquellen bieten.
Wie schon die Recherche, zeigt auch die prototypische Realisierung, dass realistische und echtzeitfähige Licht- und Schattensimulationen – trotz leistungsfähiger und vernetzter Hardware – noch immer eine der größten Herausforderungen im Bereich der Computergraphik darstellen.

Deren Realisierung stellt immer eine Abwägung zwischen Performance und Qualität dar und daher sind auch mit der realisierten Implementierung ohne Einschränkungen keine echtzeitfähigen Geschwindigkeiten zu erreichen.
Die durch Recherche und Implementierung gewonnenen Erkenntnisse sind unter Berücksichtigung der Performance- und Qualitätsbetrachtungen in die Erstellung eines SPIDER-Integrationskonzeptes eingeflossen. Es konnte gezeigt werden, wie projektive Beleuchtung unter GPU Nutzung Performer-unabhängig realisiert werden kann und welche Beschleunigungsmöglichkeiten in Frage kommen.

Eine Beleuchtungssimulation für eine komplette Fahrsimulationsszene, inklusive Schatten und Beleuchtung durch den Gegenverkehr, mit dem vorgestellten Verfahren zu realisieren, erscheint auf Basis der gewonnenen Erkenntnisse eher unwahrscheinlich. Die Scheinwerferkegel des Eigenfahrzeugs in dieser Art zu realisieren ist jedoch sehr realistisch, wenn entsprechend optimierte Algorithmen zum Einsatz kommen. Gleichzeitig ergeben sich andere Möglichkeiten die GPU für die Fahrsimulation zu nutzen. Eine sinnvolle Anwendung wäre zum Beispiel die Ergänzung der Beleuchtungssimulation um eine generelle Fragmentbeleuchtung durch das Umgebungslicht.

Neben den Lichtquellen haben zahlreiche Umweltfaktoren Einfluss auf die Beleuchtung: Nebel, Regen oder Schnee, die die ohnehin beeinträchtigte Sicht bei Nacht weiter verschlechtern und dadurch das Fahrverhalten beeinflussen. Die Betrachtung der Interaktion des Lichts mit diesen Wetterbedingungen könnte die Beleuchtungssimulation qualitativ aufwerten. Durch Erweiterung der vorgestellten Beleuchtungssimulation um den bisher ignorierten Lichteinfallswinkel könnte die Fahrsimulation außerdem einen Beitrag zur Entwicklung neuer Scheinwerfer bieten, indem deren Ausleuchtung der Straße schon vor Serienproduktion in einer realitätsnahen Umgebung subjektiv erprobt werden könnte. Vor allem für Fahrerassistenzsysteme, die auf der Lichttechnik basieren, könnte die Fahrsimulation so zu einer wichtigen Plattform werden.
Aufgrund der Komplexität einer realistischen Beleuchtung wird es wohl in naher Zukunft kein Verfahren geben, dass eine mit der realen Beleuchtungssituation vergleichbare Beleuchtungssimulation in Echtzeit berechnen kann. Sowohl die Nutzung der projektiven Beleuchtung, als auch die Shader-Programmierung, stellen jedoch, wie diese Arbeit zeigt, vielversprechende Möglichkeiten dar, sich diesem Ziel weiter anzunähern.

Literaturverzeichnis

[AMH02] Thomas Akenine-Möller and Eric Haines. *Real-Time Rendering - Second Edition*. A K Peters, Ltd., Natick, Massachusetts, 2002.

[Ban99] Siegfried Banda. *Die Lichttechnischen Grundgrößen*. Expert-Verlag GmbH, Renningen, 1999.

[Bat99] Batagelo, Harlen Costa and Junior, Ilaim Costa. Real-Time Shadow Generation Using BSP Trees and Stencil Buffers. In *In Proceedings of SIBGRAPI, Volume 12*, pages 93–102, October 1999.

[BS02] Balozs Benedek and Laszlo Szecsi. Performance improvements of rendering caustics using photon maps in interactive ray tracing. In *I. Hungarian Conference on Computer Graphics and Geometry*, 2002.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 242–248, August 1977.

[CW93] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, San Diego, 1993.

[DBB03] Philip Dutre, Kavita Bala, and Phillippe Bekaert. *Advanced Global Illumination*. A K Peters, Ltd., Natick, Massachusetts, 2003.

[DBMS02] Kirill Dmitriev, Stefan Brabec, Karol Myszkowski, and Hans-Peter Seidel. Interactive global illumination using selective photon tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 25–36, June 2002.

[EK02] Cass Everitt and Marc Kilgard. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. published online: www.developer.nvivia.com, 2002. 20.11.2004.

[FvDFH96] James Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics. Principles and Practice. 2nd Edition in C*. Addison-Wesley, 1996.

[GBMP03] Jürgen Gausemeier, Jan Berssenbrügge, Carsten Matysczok, and Klaus Pöhland. Real-time representation of complex lighting data in a nightdrive simulation. In

Proceedings of the Immersive Projection Technology and Virtual Environments 2003, Zürich, 2003.

[Hei91] Tim Heidmann. Real shadows, real time. *Iris Universe*, 18(2):23–31, 1991.

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. *Computer Graphics Forum*, 22(4):753–774, Dec. 2003. State-of-the-Art Reviews.

[Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 277–284, August 1986.

[Kil01] Marc Kilgard. Shadow Mapping with Today's OpenGL Hardware. published online: www.developer.nvivia.com, 2001. 21.11.2004.

[LBR+99] J. P. Löwnenau, J. H. Bernasch, H. Rieker, P. Venhovens, J. Huber, W. Huhn, and F. Reich. Adaptive light control - a new light concept contolled by vehicle dynamics and navigation. *SAE Transactions*, (980007), 1999.

[LKK99] P. Lecocq, J-M. Kelada, and A. Kemedy. Interactive headlight simulation. In *Driving Simulation Conference 1999, Paris, France*, 1999.

[LSB+01] Jan P. Löwnenau, Martin H. Strobl, Jost H. Bernasch, Franz M. Reich, and Alexander H. Rummel. Evaluation of adaptive light control in the BMW driving simulator. In *Driving Simulation Conference 2001, Sophia Antipolis, France*, 2001.

[Mat97] Kresimir Matkovic. *Tone Mapping Techniques and Color Image Difference in Global Illumination*. Dissertation, Technische Universität Wien, Technisch-Naturwissenschaftliche Fakultät, 1997.

[MBG+99] Tom McReynolds, David Blythe, Brad Grantham, Scott R. Nelson, and Mark J. Kilgard. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 99 course notes*, 1999.

[Mül03] Stefan Müller. Photorealistische Computergrafik - Materialien zur Vorlesung. http://www.uni-koblenz.de/~cg/veranst/ws0304/pcg_material.html, 2003. Stand: 01.04.2004.

[PBMP02] Timothy J. Purcell, Ian Buck, William R. Mark, and Hanrahan Pat. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

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

[RIE02] S. Roettger, A. Irionar, and Th. Ertl. Shadow Volumes Revisited. In *Proc. WSCG '02*, pages 373–393, 2002.

[SA03] Kurt Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. Silicon Graphics, Inc., Version 1.5 edition, 2003. Editor: Jon Leech.

[SBL99] Martin H. Strobl, Jost H. Bernasch, and Löwnenau. Realzeit-Licht-Simulation im BMW Fahrsimulator. In *GI-Workshop „Sichtsysteme - Visualisierung in der Simulationstechnik", Bremen, Deutschland*, 1999.

[SD02] Marc Stamminger and George Drettakis. Perspective shadow maps. In John Hughes, editor, *Proceedings of ACM SIGGRAPH 2002*. ACM Press/ ACM SIGGRAPH, July 2002.

[Shi02] Peter Shirley, editor. *Fundamentals of Computer Graphics*. A K Peters, Ltd., Natick, Massachusetts, 2002.

[Shi03] Peter Shirley, editor. *Realistic Ray Tracing, Second Edition*. A K Peters, Ltd., Natick, Massachusetts, 2003.

[SKvW+92] Mark Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadows and lighting effects using texture mapping. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 249–252. ACM Press, 1992.

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

[SS00] Philipp Slusallek and Marc Stamminger. Photorealistische Bildsynthese - Materialien zur Vorlesung. http://graphics.cs.uni-sb.de/Courses/ss00/pbs/, 2000. Stand: 10.08.2004.

[TPWG02] Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg. Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics*, 21(3):537–546, July 2002.

[Wat93] Alan Watt. *3D Computer Graphics*. Addison-Wesley, 1993.

[WBS03] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 270–274. ACM Press, 1978.

[WJ01] Henrik Wann Jensen, editor. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, Ltd., Natick, Massachusetts, 2001.

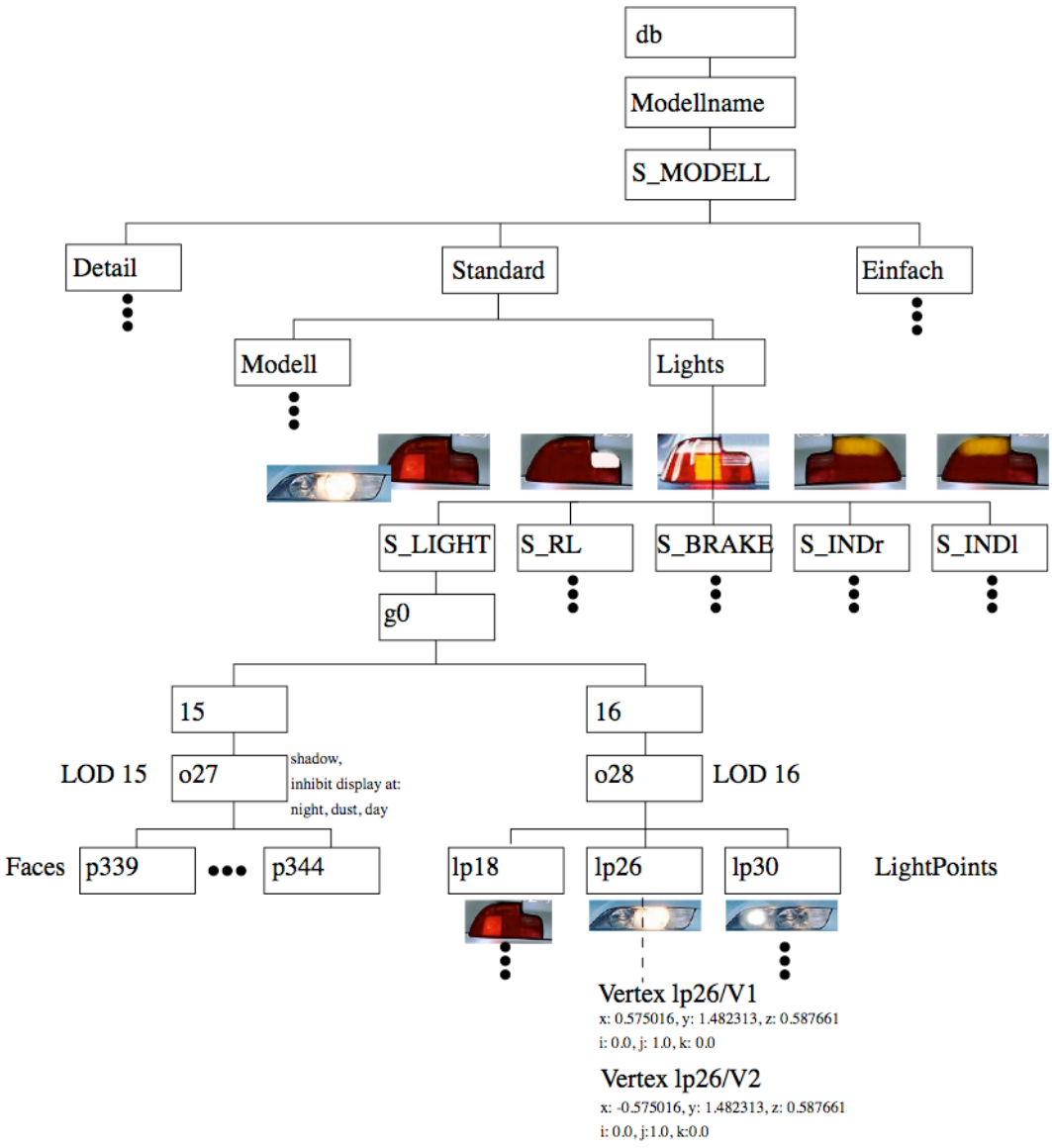
[WPKB02] Thomas Weber, Christian Plattfaut, Michael Kleinkes, and Jan Berssenbrügge. Virtual nightdrive. In *Driving Simulation Conference*, Paris, France, 11 - 13 September 2002.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, volume 20. Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/~wald/Publications.

[WSP04] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light space perspective shadow maps. In H. W. Jensen and A. Keller, editors, *Eurographics Symposium on Rendering (2004)*. The Eurographics Association, 2004.

Index

Anhang: Fahrzeug-Szenengraph



Erklärung:

Hiermit versichere ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

München, den 27. Februar 2005

Simon Knebel