

# Diplomarbeit

## Shaderwriting für Mental Ray® und Softimage®|XSI® mit Programmierbeispiel eines selbst definierten Shaders

Ausgeführt zum Zweck der Erlangung des akademischen Grades

**Dipl.-Ing. (FH) für Telekommunikation und Medien**

am Fachhochschul-Diplomstudiengang Telekommunikation und Medien St. Pölten

unter der Leitung von  
Mag. phil. Peter Adametz

Zweitbegutachterin  
Mag. Edith Huber

ausgeführt von

Rüdiger Manfred Karl Anton Raab  
Matrikelnummer 0110038091

# Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe. Diese Arbeit stimmt mit der vom Begutachter beurteilten Arbeit überein.

---

Ort, Datum

---

Unterschrift

# Inhaltsverzeichnis

0.1 Einleitung.....	7
A Theoretischer Teil.....	11
A.1 Renderer – die digitalen Fotografen.....	11
A.2 Features von Renderern.....	14
A.3 Klärung des Begriffes „Shader“.....	20
A.3.1 Material Shader.....	23
A.3.2 Texture Shader.....	23
A.3.3 Light Shader.....	25
A.3.4 Shadow Shader.....	26
A.3.5 Volume Shader.....	26
A.3.6 Photon Shader.....	27
A.3.7 Photon Volume Shader.....	28
A.3.8 Geometry Shader.....	28
A.3.9 Lens Shader.....	28
A.3.10 Output Shader.....	29
A.3.11 Contour Shader, Contour Contrast Shader, Contour Store Shader.....	29
A.4 Renderbegriffe.....	31
A.4.1 Raytracing vs. Scanline Rendering.....	32
A.4.1.1 Raytracing.....	32
A.4.1.2 Scanline Rendering.....	33
A.4.1.3 Raymarching.....	34
A.4.2 Sampling.....	35
A.4.2.1 Adaptive Sampling.....	36
A.5 Die Mental Ray-Architektur.....	37
A.6 Das Mental Ray Dateiformat.....	38
A.7 Die Mental Ray Fabrik.....	38
A.8 Wo man Programmierinformation über Mental Ray findet.....	43
A.9 Shader Assignment, Rendertrees, Shadergraphs, Phenomena.....	44
A.9.1 Shader Assignment und Shadergraphs.....	44
A.9.2 Der Rendertree.....	44
A.9.3 Shadergraphs.....	47
A.9.4 Phenomena.....	47
A.10 Unterschied zwischen "Texture" und "Material".....	50
A.11 MI-File Aufbau.....	50
A.12 Mental Ray und Softimage.....	61
A.13 SPDL (Software Package Delivery Library).....	61
A.13.1 Der SPDL-Kopf.....	62
A.13.2 Der PropertySet-Teil.....	62
A.13.3 Der Phenomenon-Teil.....	66
A.13.4 Der Defaults-Teil.....	66
A.14 Wie XSI auf Shader zugreift.....	68
B Praktischer Teil.....	70
B.1 Einleitung.....	70
B.2 Die Headerdatei "shader.h".....	70
B.2.1 Die Funktionen Strukturen und Variablen in shader.h.....	71
B.2.1.1 Datentypen.....	72

B.2.1.2 TAGs.....	73
B.2.1.3 Konstanten:.....	74
B.2.2 Die Struktur miState.....	74
B.3 Übergang von der Theorie zur Praxis – der erste Shader.....	80
B.3.1 Vorüberlegungen.....	84
B.3.1.1 Komfort.....	84
B.3.1.2 Optimierung.....	84
B.3.1.3 Was begründet das Schreiben des Shaders genau?.....	84
B.3.1.4 Wie präzise soll der Shader werden?.....	85
B.4 C-Coden, Teil 1 – Vorüberlegungen.....	85
B.5 Beispielshader „LumaReflect“.....	86
B.6 Der Shader-Wizard.....	89
B.6.1 Die Sicherheitseinstellungen von Java Applets.....	89
B.6.1.1 Opera:.....	90
B.6.1.2 Internet Explorer (Windows 2000).....	90
B.6.1.3 Mozilla Firefox:.....	91
B.6.2 Den Shader im Wizard zusammensetzen.....	91
B.6.2.1 Die Einstellungen im Shader Wizard.....	91
B.6.2.2 Fehler des Shader-Wizards ausbessern.....	96
B.7 C-Coden, Teil 2 – Learning by doing.....	96
B.7.1 Der Shadercode für lumaReflect.....	99
B.7.1.1 Erster Teil – Variablen, Shading und Glanzlicher.....	99
B.7.1.2 Zweiter Teil – Reflexionen, Falloffs und Gammakorrekturen.....	105
B.8 Einzubindende Libraries und Pfade.....	110
B.9 Kompilieren.....	110
B.10 SPDL-Logik.....	110
B.11 Kontrolle von SPDL-Dateien.....	115
B.12 Installation und Deinstallation von Shader.....	116
B.13 Shader testen.....	117
B.13.1 Die Testbilder.....	117
B.14 Zusammenfassung.....	119
B.14.1 Beantwortung der forschungsleitenden Fragestellungen:.....	119
B.14.2 Zusammenfassung der Resultate.....	121
B.14.3 Persönliche Interpretation.....	122
B.14.4 Diplomarbeitsthemen, die auf dieser Arbeit aufbauen können.....	123
C Anhang:.....	125
C.1 Aufrufsreihenfolge von Shader in Mental Ray.....	125
C.2 Quickstart zum Shaderwriting.....	126
C.3 Referenz zu wichtigen shader.h – Funktionen und Datentypen.....	127
C.3.1 Raytypes.....	127
C.3.2 Vektorrechnung:.....	128
C.3.3 Specular Highlights.....	129
C.3.4 Ray Direction verändernde Funktionen.....	129
C.4 Literaturverzeichnis:.....	134

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

**Mental Ray, Mental Images** sind eingetragene Markenzeichen von **mental images GmbH & Co. KG**  
**mental ray Phenomenon, mental ray Phenomena, Photon Map, mental ray Relay, Phenomenon, Phenomena** sind Handelsnamen von **mental images GmbH & Co. KG**

**Avid, SOFTIMAGE, XSI**, und das **XSI-Logo** sind entweder eingetragenen Markenzeichen oder Handelsnamen von **Avid Technology, Inc**

**Visual C++, Visual Studio** sind eingetragene Markenzeichen von **Microsoft Corporation**

**VERISIGN** ist ein eingetragenes Markenzeichen von **VeriSign**

**Nvidia Quadro, Nvidia** sind eingetragene Markenzeichen oder Handelsnamen von **NVIDIA Corporation**

**Renderman, PIXAR** sind eingetragene Markenzeichen von **Pixar**

## 0.1 Einleitung

*Softimage* ist eines der weltweit führenden 3D-Programme. Viele Filme, die man derzeit im Kino sehen kann wurden mit Effekten, die dieses Programm ermöglicht ausgestattet. *Softimage* ist nur ein Teil des Arbeitsprozesses, den man zur Berechnung digitaler Bilder benötigt. Der andere Teil der Arbeit wird von einer Rendersoftware namens *Mental Ray* erledigt. Diese Programme arbeiten im Prinzip voneinander unabhängig: Während der Arbeit mit *Softimage* benötigt man *Mental Ray* nicht und umgekehrt. Mit *Softimage* definiert man den Inhalt der zu berechnenden Bilder, wohingegen *Mental Ray* die mit *Softimage* erstellten Szenen in konkret darstellbare Bilder verarbeitet. Diesen Berechnungsvorgang nennt man *Rendering*.

Die Rendersoftware *Mental Ray* wird von der Firma *Mental Images* mit Firmensitz in Berlin, vertrieben. *Mental Ray* ist ein *Raytracer*, das heisst, das Programm verwendet virtuelle Strahlen um Farbwerte für einzelne Bildpunkte zu berechnen. Aufgrund dieser Fähigkeit kann *Mental Ray* Phänomene wie Lichtbrechung und Reflexionen darstellen.

*Mental Ray* ist nicht nur Bestandteil von 3D-Programmen, die Bilder für Film und Fernsehen herstellen, sondern wird in Architektursoftware eingebunden um virtuelle Bilder von Gebäuden und Innenräumen mit physikalisch korrekten Beleuchtungsverhältnissen zu berechnen. Diese Vielseitigkeit lässt erkennen, dass *Mental Ray* eine äußerst mächtige und komplexe Software ist. Dennoch wurden dem Anwender beim Programmieren der Software Möglichkeiten offen gelassen eigene Erweiterungen zu integrieren.

Diese Erweiterungen heißen *Shader* und werden als kompilierter Programmcode in Form einer .DLL-Datei in die Software integriert. (Begriffsdefinition siehe Kapitel A.3 ) In *Mental Ray* gibt es mehrere Typen von *Shadern*. Jeder Typ wird zu einem anderen Zeitpunkt und aus einem anderen Grund ausgeführt. Diese Zerlegung in kleinere Aufgaben ermöglicht dem Anwender einen *Shader* für einen ganz bestimmten Zweck, zum Beispiel zur Berechnung von Schatten zu schreiben. Dieser *Shader* wird danach für nichts anderes eingesetzt, als Schatten auf Objekten zu berechnen.

Softimage ist ein Werkzeug um Szenen am Computer digital zu erstellen. Natürlich ist es in Softimage auch möglich das Aussehen der Materialien einzustellen, welche Mental Ray später berechnen wird. Dies ist bequem über die grafische Benutzeroberfläche von Softimage möglich.

Das bedeutet, dass Softimage Kenntnis darüber besitzen muss, welche Shader in Mental Ray integriert sind und wie diese angesprochen werden, da Mental Ray selbst keine grafische Benutzeroberfläche hat. Sowohl Softimage als auch Mental Ray müssen über die Funktionsweise eines Shaders Bescheid wissen. Andernfalls kann Mental Ray das Objekt mit dem Shader nicht korrekt berechnen und stürzt im schlimmsten Fall ab.

Man erreicht dies, indem man nach dem Kompilieren eines Shaders in eine DLL-Datei eine zusätzliche Datei in Softimage registriert. Dadurch erhält Softimage die nötige Information über den Aufbau des Shaders. Da das selbstständige Tippen einer solchen Datei mühsam ist, gibt es mittlerweile Werkzeuge, die diesen Prozess automatisieren und den Vorgang der Shaderintegration erheblich beschleunigen.

Ein guter Shader liefert optisch eindrucksvolle Ergebnisse mit möglichst wenig Rechenaufwand. Ein weiterer Aspekt ist die Umsetzung surrealistischer Effekte für Film- und Fernsehproduktionen. Es ist in Softimage möglich Effekte mit Hilfe des Rendertrees umzusetzen (Siehe Kapitel A.9.2). Die volle Kontrolle über einen Effekt hat man aber erst, wenn man ihn von Grund auf selbst programmiert.

#### **Forschungsleitende Fragestellungen dieser Arbeit sind:**

- Wie sind die beiden Programme Mental Ray und Softimage miteinander verflochten?
- An welchen Schnittstellen muss man einen selbstprogrammierten Shader einbinden?
- Welche Tools stehen zum Shaderwriting zur Verfügung und was ist bei ihrer Verwendung zu beachten?
- Wie programmiert man Shader die über den Funktionsumfang der Standardshader hinausgehen?

#### **Hypothese:**

Es gibt Situationen in denen man mit den Basis-Shadern, welche mit Mental Ray und Softimage geliefert werden, nicht die gewünschten visuellen Effekte erzeugen kann. Mit Hilfe der Schnittstelle von Mental Ray und Softimage, welche es erlaubt eigenen Shadercode zu integrieren, ist es möglich solche Situationen zu umgehen und effektvolle Lösungen zu finden.

Die Diplomarbeit gliedert sich in 2 Hauptteile. Wie vorhin schon erwähnt ist ein Grundwissen in Mental Ray unabdingbar um für diese Software Shader schreiben zu können. Ebenso ist Erfahrung im Umgang mit dem Programm Softimage nötig. Die Arbeit baut somit auf bestehendem Wissen auf, erweitert es und demonstriert auf anschauliche Art und Weise an einem konkreten Beispiel die Anwendung in der Praxis.

Der erste Teil ab Seite 10 gibt einen allgemeinen Überblick über Renderer im generellen. Der Begriff “Shader” wird eingehend erläutert, wie auch auf den Hintergrund verschiedener Rendertechniken eingegangen wird. Anschließend werden die speziellen Fähigkeiten von Mental Ray beschrieben. Von besonderem Interesse ist die Szenenbeschreibungssprache von Mental Ray. Diese unterscheidet sich sehr von der Darstellung der Szenenelemente in Softimage. Viele Softimage Benutzer werden überraschend neue Informationen finden die zum tieferen Verständnis von bestimmten Mental Ray-spezifischen Problemen beitragen. Selbst der Leser der keinen Shader zu schreiben beabsichtigt, kann dem theoretischen Teil wertvolle Informationen entnehmen.

Im zweiten praktischen Teil ab Seite 67 wird das in der Theorie erworbene Wissen an einem Experiment erprobt. Die Themenbehandlung ist dabei nicht rein mathematisch, sondern zieht auch künstlerisch-ästhetische Aspekte der subjektiven Wahrnehmung in Betracht.

Unumgänglich dabei ist die programmiertechnischen Aspekte von Mental Ray zu behandeln. Daher wird zu Beginn die äußerst Komplexe Struktur der Headerdatei “shader.h” erklärt. In dieser knapp 40 Seiten langen Datei befindet sich alles, was man zum Schreiben eines Shaders benötigt. Hier werden nicht nur Funktionen deklariert sondern auch große Strukturen wie der *Shaderstate*, Strukturen mit Information über die Bildformate oder auch die *miOptions*. Auf jedes einzelne Detail einzugehen ist nicht das Forschungsziel dieser Diplomarbeit. Der Leser wird mit dem nötigen Basiswissen ausgestattet, mit welchem er die folgenden Kapitel nachvollziehen kann.

Ein weiterer Bestandteil des Softimagepaketes ist der *Shader Wizard*. Der Shader Wizard ist ein wichtiges Hilfsmittel um rasch die leeren Basisdateien, die zum Weiterarbeiten benötigt werden erzeugen zu lassen. Damit erspart man sich die Grundstruktur jedes Mal von neuem schreiben zu müssen. Man kann komfortabel eine Headerdatei mit allen benötigten Parameter generieren.

Abschließend wird der wichtigste Teil, das Schreiben eines Funktionscodes behandelt.

Programmiertechnische Aspekte wie *Lazy Evaluation* und mathematische Überlegungen für die Umsetzung eines *Falloff*, einer *Luminanzberechnung*, ect. werden angestellt und umgesetzt. Der Programmcode wird ausführlich erläutert, der fertige Shader kompiliert sowie in der Praxis getestet. Beispielbilder stellen die gerenderten Resultate einer Testszene dar.

Im Anhang findet der interessierten Leser Kurzanleitungen, Auszüge aus der Datei *shader.h* sowie Dokumentationen von Mental Ray welche beim eigenständigen Entwickeln neuartiger Shader behilflich sind. Der Einstieg in diese komplexe Materie ist mitunter schwierig, da man einiges an Vorwissen benötigt um überhaupt mit dem aktiven Programmieren beginnen zu können. Darin begründet sich die Gliederung der Diplomarbeit in einen theoretischen und einen praktischen Teil. Bei Verständnisschwierigkeiten in einem der Teile kann man im anderen Teil über die Theorie beziehungsweise Praxis nachschlagen. Somit bietet diese Arbeit eine Basis für alle, die sich mit dem Thema Shaderwriting eingehender beschäftigen wollen.

# A Theoretischer Teil

*Im ersten Teil dieser Diplomarbeit ist Information über generelle Renderbegriffe zu finden. Nach einer kurzen Einleitung wird auf verschiedene Rendertechniken eingegangen. Später wird der Mental Ray-Renderer genau erklärt und auf seine Besonderheiten eingegangen. Diese Inhalte sollen das Verständnis über diesen Renderer vertiefen. Damit fällt es einem leichter den Programmcode des zweiten Teils zu verstehen und daraus das Wissen über Shaderwriting zu vertiefen.*

## **A.1 Renderer – die digitalen Fotografen**

Beinahe jeder Mensch wird heutzutage mit künstlichen Bildern konfrontiert – in jeder Film- und Fernsehproduktion wird von Effekten gebrauch gemacht, welche ohne Computertechnologie nicht umzusetzen wären. Die Technik ist mittlerweile so ausgereift, dass man den Unterschied gar nicht bemerkt. Es wird dann erst nach einiger Überlegung klar, dass eine Kamerafahrt um ein Werbeprodukt mit einer realen Kamera gar nicht mehr möglich ist.

Der Bereich der 3D-Animation ist heute aus der Filmbranche nicht mehr wegzudenken. Was 1982 mit dem Film „Tron“ begann reifte über die Jahrzehnte zu einem gewaltigen, mittlerweile selbstständigen Filmsektor. Der Film „Tron“ setzte einen Meilenstein in der Computeranimationsgeschichte der Computergrafik:

Die erste Raytracing-Software wurde hergestellt und Ken Perlin erfand die heute nicht mehr wegzudenkende „Perlin Noise“. Die Perlin Noise ist ein Mathematisches Modell mit dem es erstmals möglich ist prozedurale Texturen<sup>1</sup> herzustellen. Diese Technik legte den Grundstein für unzählige neue Effekte, die von Eiskristallen, Wolkenformen bis zu züngelnden Flammen reichen.

Die wenigsten Menschen wissen, was im Computer passiert, wenn sie 3D-Spiele spielen oder mit 3D-Animationssoftware arbeiten. Jedem Menschen sagt der Begriff Polygon zwar etwas und man weiß, dass ein Polygon eine aus 3 oder mehr Eckpunkten aufgespannte Fläche ist (poly, griech.: dt. viel). Doch was danach kommt ist den meisten ein Rätsel. Welche mathematischen Berechnungen dahinter stecken, um auf einem 2-Dimensionalen Bildschirm die 3D-Objekte in perspektivischer oder orthografischer Ansicht abzubilden ist für den Endkunden ein Mystikum. Der Begriff lautet Matrizenrechnung:

---

<sup>1</sup> Prozedurale Texturen werden anhand von mathematischen Formeln berechnet und benötigen daher keinen Speicherplatz auf einem Datenträger. Ein weiterer Vorteil ist, dass sich ihr Muster in keiner Weise jemals wiederholt.

Eine Matrize ist für die Transformation der Punkte im 3D-Raum zuständig und mit Matrizenrechnung ist es auch möglich, Punkte so zu transformieren, dass man eine Projektion auf eine 2-Dimensionale Fläche erhält. Im 3D-Programm wird die virtuelle Kamera auf die Polygone gerichtet und dann werden diese Polygone auf eine Fläche projiziert. Diese Fläche ist in dem Fall eine Bilddatei von X mal Y Pixel Auflösung und die Projektion wird durch Multiplikation der 3D-Punkte mit einer Projektionsmatrix bewerkstelligt. [OGL\_R, 2003, s. 101ff.]

$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2n}{f-n} \\ 0 & 0 & 1 & \frac{f-n}{f-n} \end{pmatrix} \quad \text{und} \quad P^{-1} = \begin{pmatrix} \frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\ 0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -\frac{f-n}{2fn} & \frac{f+n}{2fn} \end{pmatrix}$$

Abbildung 1: Eine Projektionsmatrix welche eine Orthografische Projektion durchführt

In der realen Welt geschieht Projektion ganz einfach, indem man eine Kamera auf die zu fotografierende Szene richtet und den Auslöser betätigt. Dabei treffen Lichtteilchen auf das Objektiv der Kamera, welche von der Linse gebrochen werden und auf das Filmmaterial treffen. Die Projektionsfläche ist in diesem Fall das Filmmaterial. Für die Projektion sind keine mathematischen Berechnungen erforderlich. Die Lichtstrahlen gehorchen den Gesetzen der Physik.

Wie man sieht, sind die Abbildungsvorgänge zueinander analog. Lediglich die Techniken unterscheiden sich voneinander.

Das Rendering kann man als eine Art virtuelle Fotografie verstehen. Eine 3D-Szene wird analysiert, dabei wird ermittelt, welches Objekt das gerade darzustellende Pixel belegt. Wie das Pixel zu seiner Farbe kommt, ist die Arbeit des Renderprogrammes und wird im Laufe dieser Diplomarbeit noch erörtert.

Rendering ist der Berechnungsvorgang eines Programmes welches Information über eine Szene<sup>2</sup> verarbeitet und dabei jedes Pixel im auszugebenden Bild mit einem Farbwert füllt. Das Ergebnisformat kann dabei von anderer Natur als ein RGBA-Bild sein.

2 Unter dem Begriff "Szene" ist in dieser Arbeit folgende Definition zu verstehen: Information über 3D-Geometrie im Raum sowie der möglicherweise ebenso enthaltenen Information wie diese Geometrie von einem bestimmten Programm zu interpretieren ist.

In einer Szene befinden sich verschiedene Elemente welche das Ergebnis des Zielbildes beeinflussen. Dies können zum Beispiel Lichtquellen mit einer bestimmten Richtwirkung und Farbe sein oder Materialien, welche die Oberfächeneingenschaft einer 3D-Geometrie beschreiben. Dabei kommen je nach Renderer verschiedene Algorithmen für die Bildberechnung zum Einsatz. Je nach verwendeten Algorithmen kann das optische Ergebnis und die Renderzeit dabei stark variieren. Manche Renderer beherrschen die Berechnung von Effekten wie volumetrischen Nebel, Lichtbrechung, Reflexion, indirekter Beleuchtung und weiteren rechentechnisch aufwändigen Effekten. Die Bezeichnungen variieren dabei von Software zu Software wobei die Algorithmen dahinter sehr ähnlich sind. Die Bezeichnungen, wie sie in dieser Arbeit gemacht werden, halten sich alle an die Namensgebung von Mental Ray.

## A.2 Features von Renderern

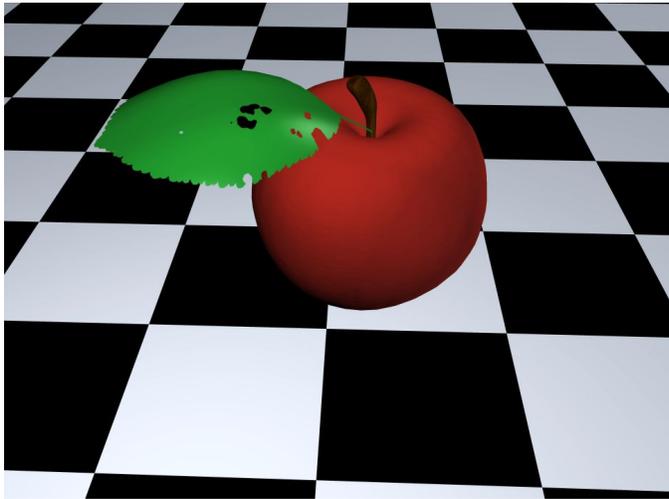
*Nicht alle Renderer funktionieren gleich. Die Fähigkeiten verschiedener Render unterscheiden sich voneinander. Die wichtigsten Fähigkeiten eines modernen Renderprogrammes werden in diesem Kapitel kurz erklärt*

Renderer sind Programme welche Bilder mit Hilfe von vordefinierter Informationen über eine 3D-Szene berechnen können. Die Fähigkeiten von Renderern unterscheiden sich oft untereinander. Aufgrund verschiedener Algorithmen entsteht je nach Renderer ein charakteristischer Look der für diesen Renderer charakteristisch ist. Das ist mitunter der Grund, warum Künstler oft eine ganz bestimmte Software verwenden. Diese Urteile sind meist subjektiver Natur. Es gibt derzeit auf dem Markt eine Unzahl hochwertiger Produkte die professionellen Ansprüchen gerecht werden. Erst im Detail werden die Unterschiede sichtbar.

Jeder Renderer analysiert die Information über eine 3D-Szene auf und erzeugt nach diesen Vorgaben eine Bilddatei in dem gewünschten Format. Die Palette der Bildformate ist breit. Meist werden die Bilder in verlustlosen Formaten wie *.tif* oder *.tga* abgespeichert. Aber auch andere Bildformate, die eine größere Farbtiefe erlauben oder sogar Helligkeitswerte, die über das „weiß“ des Computerbildschirmes hinausgehen, können eingesetzt werden. Solche Bildformate werden HDRI's [LUEF03] (High Dynamic Range Images) und werden gerne für Final Gathering<sup>3</sup> verwendet. Von dieser Rendertechnik wird später noch die Rede sein. Welche Bildformate man nun mit einer Rendersoftware abspeichern kann, entnimmt man am besten der Dokumentation des Renderers oder der 3D-Software welche den Renderer in das

---

<sup>3</sup> Final Gathering ist eine Technik die schnell realistische Beleuchtung nachempfinden soll. Dabei wird nicht für einzelne Pixel ein Farbwert berechnet sondern großflächig getestet, von wo an einer bestimmten Stelle die Bestrahlung durch Materialien stattfindet. Indem man eine Szene mit einem HDRI-Bild umgibt, erhält die Szene realistische Beleuchtung weil ein HDRI dank des hohen Dynamikumfangs die Szene in echtes Umgebungslicht taucht.



*Abbildung 2: Ein Apfel auf einem Schachbrettmuster. Der Apfel und der Boden werden mit einem Lambertshader berechnet.*

ihn sich vorstellt. Die Oberflächenmerkmale, welche charakteristische Eigenschaften der Beschaffenheit des Apfels beschreiben fehlen gänzlich.

Programm einbindet.

Die ersten Renderer waren nur in der Lage Objekte von Lichtquellen beleuchten zu lassen und dabei je nach Farbe des Lichtes, Farbe des Objektes und Winkel der Oberflächennormalen und dem Lichtstrahl einen Farbwert für das aktuelle Pixel zu berechnen. Diese Methode 3D-Geometrie zu *shaden* ist wohl eine der einfachsten, und sie wirkt beim Betrachten flach und unrealistisch. Dem Betrachter fällt sofort auf, dass dieses Bild nur die Idee eines Apfels vermittelt, aber keinen richtigen Apfel darstellt, so wie man



*Abbildung 3: Texturierter Apfel auf altem Parkettboden.*

Das zweite Bild zeigt denselben Apfel voll texturiert mit Specular Highlights, Schattenwurf und Bumpmapping. Es stellt sich wohl kaum die Frage welches der beiden Bilder dem Betrachter besser gefällt.

Der Mensch reagiert auf verschiedene optische Ausprägungen die es ihm innerhalb von Sekundenbruchteilen ermöglichen über die Beschaffenheit von physikalischen Medien zu urteilen. Renderer versuchen möglichst viele optische Phänomene nachzubilden. Dabei gibt es zwei verschiedene Ansätze:

- Die physikalisch korrekte Nachbildung. Diese ist vor allem für architektonische Renderings von großer Bedeutung. Mit Hilfe dieser Bilder kann man rechtzeitig erkennen kann ob in einem Raum ausreichend ausgeleuchtet wird oder ob zusätzliche Leuchtmittel in die Installation eingeplant werden müssen. Die Berechnung solcher Bilder dauert oft Stunden bis Tage, Präzision hat oberste Priorität.
- Die optisch überzeugende Nachbildung. In der Unterhaltungsindustrie zum Beispiel spielt Realismus eine untergeordnete Rolle: Bilder müssen überzeugen, sie brauchen oder dürfen nicht realistisch sein. Der Vorteil dabei ist, dass aufgrund dieser Tatsache manche Berechnungen zu Gunsten der Renderzeit eingespart werden können.

Optische Ausprägungen von Materialien, Gasen und Flüssigkeiten in der realen Welt sind vielseitig. Die für das menschliche Auge wohl wichtigsten sind folgende:

## **Materialfarbe**

Sie bestimmt die Farbe oder die Textur mit der das Objekt überzogen ist. Der Mensch reagiert auf Farbe subtil, weshalb das Gebiet der Farbpsychologie sehr groß ist. Anhand ihrer Farbe kann man reife Äpfel von Unreifen oder verfaulten allein durch das Ansehen unterscheiden. Vor allem die Textur ist in einer guten 3D-Szene wichtig. Dank der enormen Speicherkapazität von Computern ist es kein Problem Texturen mit der Größe von zig Megabyte an geometrischen Objekten anzubringen.

## **Glanzlichter**

Sind unechte Spiegelungen von Lichtquellen auf der Materialoberfläche. Sie geben einen zusätzlichen optischen Eindruck über die Oberflächenbeschaffenheit. Sind Glanzlichter breit und matt vermutet der Betrachter dahinter eine weiche, samtige Oberfläche, wohingegen schmale spitze Glanzlichter auf glatte, polierte Oberflächen hinweisen. Die Berechnung von Glanzlichtern ist im Gegensatz zu echten Reflexionen billig und schnell. Glanzlichter sind Teil der Berechnung von Material Shader. Es gibt Wissenschaftliche als auch nichtwissenschaftliche Modelle für die Berechnung von Glanzlichtern.

[BSR, 1999,

[http://www.siggraph.org/education/materials/HyperGraph/illumin/specular\\_highlights/blinn\\_model\\_for\\_specular\\_reflect\\_1.htm](http://www.siggraph.org/education/materials/HyperGraph/illumin/specular_highlights/blinn_model_for_specular_reflect_1.htm)]

## **Schattenwurf**

Anhand des Schattenwurfes erkennt man aus welcher Richtung Licht auf eine Oberfläche trifft. Schatten geben dem Bild mehr Tiefe. Der Betrachter des Bildes kann somit Gegenstände wahrnehmen, die er nicht direkt sieht, wie zum Beispiel den Schattenwurf eines Baumes. Bei der Berechnung von Schatten gibt es zwei verschiedene Verfahren: Raytraced Shadows und Shadow Maps: Beide Verfahren haben ihre Vor- und Nachteile, wobei Raytraced Shadows aufwändiger zu berechnen sind als Shadow maps. Allerdings haben beide Verfahren ihre Daseinsberechtigung. [MRDOC, 2003, node118.htm]

## **Spiegelungen, Transparenz und Lichtbrechung**

Transparenz und Lichtbrechung sind Phänomene, die bei durchsichtigen Objekten auftreten. Flüssigkeiten sind lichtdurchlässig und brechen das Licht, sobald es von einem Medium in das andere übergeht. Sehr oft tritt bei Flüssigkeiten auch das Phänomen der Spiegelung auf, sobald der Einfallswinkel des Lichtstrahls einen bestimmten Wert erreicht hat. Weiters tritt Spiegelung bei glatten, polierten Oberflächen auf. Wenn alle Lichtstrahlen in die gleiche Richtung reflektiert werden, spiegelt das Objekt perfekt.

Reflexion und Lichtbrechung erfordert den Einsatz von Raytracing. Raytracing ist vor allem im Verbund

mit Spiegelungen und Transparenzen sehr rechenintensiv da unzählige Rays berechnet werden müssen. Die einzig unaufwändige Alternative ist, bei Transparenten Objekten keinen anderen Brechungsindex als den Wert 1.0 zu verwenden. Reine Transparenz ohne Lichtbrechung auch mit Scanlinerendering berechnet werden.



**Abbildung 4 :** Lampe, die einen Raum ausleuchtet. Ohne Global Illumination wäre nur das beleuchtete Dreieck auf dem Boden zu sehen.

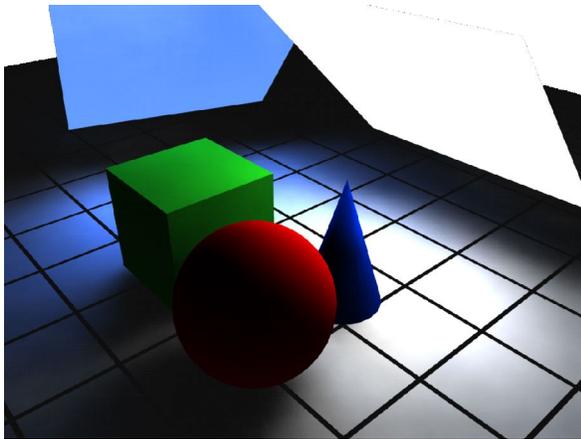
### **Global Illumination und Caustics**

In der normalen Welt wird Licht von Objekten mit einer gewissen Energie wieder reflektiert. Dabei muss es sich nicht um einen Spiegel handeln:

Nichtspiegelnde Objekten reflektieren das Licht diffus, das heisst in eine Vielzahl von Richtungen. Die reflektierte Lichtmenge- und Farbe hängt von der Farbe des Objektes ab. Dunkle Objekte reflektieren weniger Licht, helle Objekte mehr. Darum wird auch ein Zimmer erhellt, wenn man eine Lampe auf die Decke richtet. Im Normalfall ziehen Renderer diesen

nichtlokalen Effekt nicht in Betracht. Das heisst, würde das Szenario in einer 3D-Szene nachgestellt, wäre der Raum stockdunkel und auf der Decke ein beleuchteter Fleck.

Nachdem ein Renderprogramm nicht die abermillionen Photonen der realen Welt nachberechnen kann, wird dieser Effekt approximiert. Dazu wird begrenzte Anzahl an virtuellen Photonen in die Szene geschossen, welche sich wie reale Lichtteilchen verhalten. Die Teilchen werden reflektiert, gebrochen oder verschluckt. Diese Technik nennt sich "*Global Illumination*". Ein roter Würfel auf einer weißen Oberfläche wird seinerseits von weißem reflektiertem Licht der Oberfläche angestrahlt, während und die Oberfläche vom eingefärbten Licht des Würfels angestrahlt wird. Diese Erscheinung heisst "*Color Bleeding*". In dem Beispielbild ist gut zu sehen, dass der Boden von der roten Seitenfläche der Lampe rötlich eingefärbt wird.

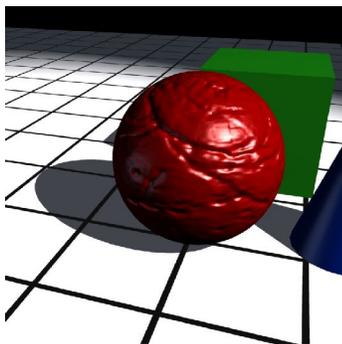


**Abbildung 5:** Geometrische Objekte von selbstleuchtenden Flächen mit der Final Gathering-Technik beleuchtet

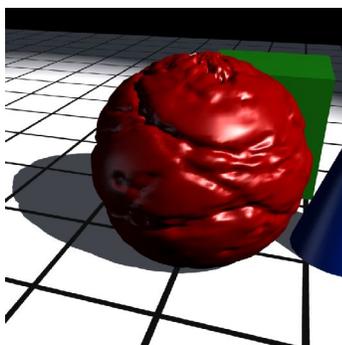
lediglich aus der Direkten Beleuchtung. Das bringt natürlich eine Geschwindigkeitssteigerung mit sich. Bilder die mit Final Gathering gerendert wurden sehen sehr weicher und geschmeidig aus. Man kann eine Szene wie im Beispielbild nur von strahlenden Oberflächen beleuchten lassen und bekommt auf diese Weise schon brauchbare Resultate. Final Gathering wird gerne zusammen mit Global Illumination verwendet, da der Effekt von Global Illumination durch das Final Gathering verstärkt wird.

## Final Gathering

Dieser Effekt ist ein Abkömmling der Global Illumination. Licht wird im Gegensatz zu Global Illumination nicht von Lichtquellen emittiert, sondern von der Geometrie selbst, welche von sich aus leuchtet. Somit ist dieses Objekt in einer völlig finsternen Szene dennoch sichtbar. Dies zieht Final Gathering in Betracht und berechnet auf den anderen Objekten von diesen Oberflächen abgestrahltes Licht. Die Lichtteilchen werden dabei nicht wie bei Global Illumination mehrfach reflektiert sondern stammen



**Abbildung 6:** Kugel mit Bump Mapping



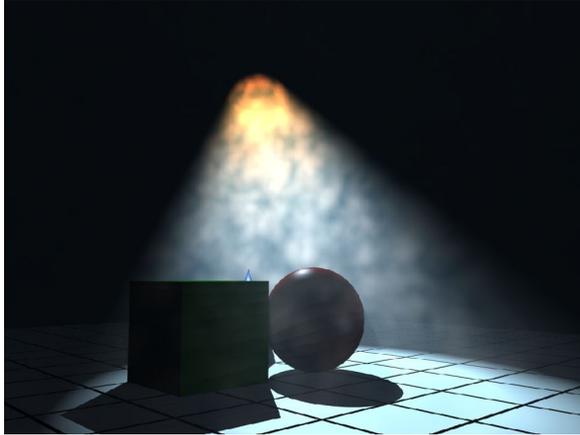
**Abbildung 7:** Kugel mit Displacement Mapping

## Bump Mapping und Displacement Shader

Die Struktur einer Oberfläche wird nicht nur von Glanzlichtern und Reflexionen bestimmt, sondern auch von kleinen Unebenheiten (im Englischen „Bumps“ genannt).

Bumpmapping zieht dies in Betracht und kann aus einer Bildvorlage ein Relief bilden. Tatsächlich ändert sich nichts an der Geometrie, dieses Relief täuscht lediglich eine unebene Oberfläche vor – deswegen werfen diese Unebenheiten auch keine Schatten. Eine Bump Map ist selbst kein eigener Shader. Es werden lediglich die Normalvektoren des Shaders verändert, was an mehreren Stellen in Mental Ray geschehen kann.

Displacement geht einen Schritt weiter und Verändert zur Renderzeit tatsächlich die Geometrie der Objekte. Das kostet natürlich Zeit da der Renderer diese neue Geometrie erst erzeugen muss. Dafür wird mit dieser Methode erzielt, dass diese Unebenheiten auch tatsächlich Schatten werfen. Im Gegensatz zum Bumpmapping handelt es sich bei Displacement um einen echten Shader.



*Abbildung 8: Szene mit Nebel. Die Strahlen reisen durch ein Volumen mit ungleichmäßiger Dichte*

## Volume Rendering

In den gängigen 3D-Programmen ist jede 3D-Geometrie innen hohl – durchdringt die Kamera zufällig die Außenhülle, so sieht man das leere Innere des Objektes. Um die Leere mit Staub, Rauch oder Partikeln zu füllen stehen dazu Volume Shader zur Verfügung. Ein Volume Shader verfolgt den Strahlengang des Lichtes durch das Innere und ist im Stande das Innere des Objektes auszufüllen. In den meisten Fällen bildet man Rauch, Nebel oder Gas mit Volumeeffekten nach. Einfache Volume Shader

simulieren gleichmäßigen Nebel, andere wiederum können komplexe fraktale Muster im Inneren des Objektes erzeugen. Die Volume Shader die von Mental Ray mit Softimage|XSI mitgeliefert werden erfüllen die Grundbedürfnisse und sind für Laien nicht einfach zu handhaben. Erfahrung über die Funktionsweise von Mental Ray ist vorteilhaft, denn durch bloßes Herumprobieren wird der Anwender schnell frustriert.

## A.3 Klärung des Begriffes „Shader“

*Der Begriff "Shader" wird in dieser Diplomarbeit sehr oft fallen. Was sich genau hinter diesem Begriff verbirgt, wird auf den nächsten Seiten dargelegt. Es wird erklärt, dass es verschiedene Typen von Shadern gibt und welche das Renderprogramm Mental Ray bei seinen Berechnungen verwendet. Es wird jeder Shadertyp erklärt, wann dieser zum Einsatz kommt und was seine Aufgabe ist.*

Ursprünglich konnten Renderer nur Farbe und Beleuchtung auf Objektoberflächen berechnen. Mental Ray erlaubt es, fast jeden Aspekt des Renderings selbst in die Hand zu nehmen. Man kann seinen eigenen Programmcode für fast jede Funktionalität von Mental Ray programmieren und in den Renderer einbinden.

Der Rendervorgang gliedert sich grob unterteilt in 3 Stufen: Preprocessing, Pixelberechnung und Postprocessing. Diese Stufen werden noch einmal feiner während des Rendervorganges aufgebrochen: Für jede kleine Aufgabe existieren spezielle Funktionen die bestimmte Aufgaben erledigen. Diese kleinen Unteraufgaben sind zum Beispiel Berechnungen von Schatten, Auswertung der Farbe auf Oberflächen ect. Um zum Beispiel den Farbwert auf einer Oberfläche zu berechnen, muss man sich auf weitere

Funktionen verlassen: Eine Funktion welche die Eigenschaften der Lichtquelle wie Intensität, Farbe und Intensitätsabfall über die Distanz berechnet. Eine weitere die den Schattenwurf anderer Objekte berechnet, eine die berechnet, ob der Lichtstrahl durch ein Volumen (Nebel, Rauch) reist und so weiter. Eine Funktion, die auf eine bestimmte Aufgabe spezialisiert ist, nennt sich „Shader“ und ist nichts anderes als ein Programmcode welcher für Mental Ray in der Sprache C/C++ verfasst ist.

*Definition:* Ein Shader in Mental Ray ist ein Programmcode der in C/C++ geschrieben wurde. Dieser Programmcode schreibt Mental Ray vor, wie er eine Berechnung für einen bestimmten Rendervorgang durchzuführen hat.

Mental Ray implementiert viele Arten von Shader wie

- Material Shader
- Texture Shader
- Shadow Shader
- Photon Shader
- Volume Shader
- Photon Volume Shader
- Environment Shader
- Geometry Shader
- Displacement Shader
- Light Shader
- Light Emitter Shader
- Lens Shader
- Output Shader
- Contour Shader
- Contour Contrast Shader
- Contour Store Shader

Alle Shader auf das Genaueste zu beschreiben würde den Rahmen dieser Arbeit sprengen. Daher konzentriert sich diese Diplomarbeit auf die im Alltag am häufigsten eingesetzten Shader.

Shader arbeiten Hand in Hand. Meistens wird das Ergebnis der Berechnung des einen Shaders an den nächsten weitergereicht, der nun seinerseits seine Berechnungen durchführt. Was ein Shader nun berechnet, hängt von der Art des Shaders ab. Die Wirkungsweise von Shader ist oft völlig verschieden – manche Shader berechnen keine Farbwerte, sondern ändern lediglich den Strahlengang von Eye-Rays (siehe nächstes Kapitel) oder sie stellen Koordinaten für Texturelookups<sup>4</sup> zur Verfügung. Shader beschränken ihre Berechnungen nicht nur auf Farbwerte, sondern können auch andere

---

<sup>4</sup> Bei einem Texturelookup liefert ein Shader anhand von eingegebenen Koordinaten den Zahlen- oder RGB-Wert aus einer Bilddatei

Ergebnisdatentypen wie Vektoren, Skalare, Booleans (True/False-Werte) produzieren. Ein Shader muss keine sichtbaren Farbwerte als Ergebnis haben. Ein Shader kann intern Berechnungen durchführen und seine Ergebnisse anderen Shadern zur Verfügung stellen die diese Berechnungen benötigen. Solche Shader sind sogenannte Hilfs- oder Toolshader und können beim Bau von „*Render trees*“ von großem Nutzen sein. Auch ein „unsichtbarer“ Shader ist somit maßgeblich am Renderprozess beteiligt, denn ohne seine Ergebnisse könnte ein anderer Shader seine Berechnungen nicht durchführen.

Um einen Mental Ray Shader in Softimage|XSI zu implementieren, muss man den Shadercode auf Windows-Systemen in eine Dynamic Link Library (DLL-Datei) kompilieren. Auf Unix-Systemen werden Dynamic Shared Objects (DSO) erzeugt [vgl. BIB1].

Weiters muss man eine SPDL-Datei bereitstellen. In dieser Datei wird der Shader deklariert und sein grafisches Layout für in Softimage|XSI festgelegt. Grob gesprochen ähnelt eine SPDL-Datei einer HTML-Datei: Eine HTML-Datei definiert wie eine Webseite im Browserfenster angezeigt werden soll. Eine SPDL-Datei ist dasselbe für Softimage|XSI – Softimage liest die Datei und kann daraus die dazugehörige Propertypage (unter XSI-Animatoren auch als PPG bekannt) aufbauen. Mit dieser Information und der DLL- oder DSO-Datei, welche den Programmcode beinhaltet, ist es Softimage nun möglich den Mental Ray Renderer mit der nötigen Information zu versorgen. Man kann eine SPDL-Datei auch so gestalten, damit man einen Shader nicht falsch im Rendertree (siehe Kapitel “Shader Assignment, Rendertrees, Shadergraphs, Phenomena”) verwenden kann. Man kann einem Shader vorschreiben nur als Material und Shadow-Shader zu fungieren, nicht jedoch für alle anderen Arten. Solange der User keine Sicherheitsvorkehrungen trifft, kann nämlich weder Mental Ray noch Softimage erkennen, für welchen Zweck ein Shader geschrieben wurde. Es liegt in der Verantwortung des Shaderwriters in der SPDL oder schon direkt im C/C++-Code des Shaders Restriktionen zu setzen. Falls man dies unterlässt liegt es in der Verantwortung des 3D-Künstlers auchtzugeben, für welchen Zweck er den Shader einsetzt. Ein falsch verwendeter Shader kann bestenfalls schwarze oder weisse Pixelwerte erzeugen, im schlimmsten Fall stürzt Mental Ray ohne Vorwarnung ab.

Andererseits sollte man darauf achten, inwieweit ein Shader für mehr als nur einen Shaderzweck einsetzbar ist: Wenn man einen Shader für einen anderen Zweck gebrauchen *kann* ohne dass er den Mental Ray Renderer zum Absturz bringt, sollte man dem User unbedingt auch die Möglichkeit dazu lassen. Vielleicht ermöglicht dieser Missbrauch einen speziellen Effekt der sonst nicht möglich wäre. Viele Shader wurden gezielt so programmiert, dass sie als Material, Shadow und Photon Shader eingesetzt werden können.

Auf Seite 123 ist der Mechanismus erklärt, der einen Shader automatisch erkennen lässt, für welchen Zweck er gerade eingesetzt wird und sich dementsprechend verhält. Auf Seite 88 ist erklärt, wie man

einstellt, dass Shader im Softimage Rendertree nicht an die falschen Nodes angehängt werden können.

Was Anfängern wohl sehr verwirrend vorkommen mag ist, dass sich Shader gegenseitig aufrufen und je nach Bedarf andere Shader Shader in ihrem Aufruf involvieren. Vgl. dazu die Abb. im Anhang.

"Aufrufsreihenfolge von Shader in Mental Ray".

### **A.3.1 Material Shader**

Material Shader sind die wohl am meisten verwendeten Shader. Mental Ray schreibt vor, dass jedes geometrische Objekt einen Material Shader besitzen *muss*. Hat ein Objekt keinen Material Shader, weigert sich Mental Ray dieses Objekt zu rendern und übergeht es einfach. Ein Material Shader ist der erste Shader der aufgerufen wird, wenn ein Strahl (*Ray*) auf ein geometrisches Objekt trifft. Im Regelfall werden dabei weitere Shader involviert: Um die Beleuchtung von Lichtquellen auf dem Material zu erhalten, werden die Light Shader aufgerufen. Soll die Lichtquelle Schattenwurf erzeugen, werden die Shadow Shader, der sich im Strahlengang des Lichtes befindlichen Objekte aufgerufen.

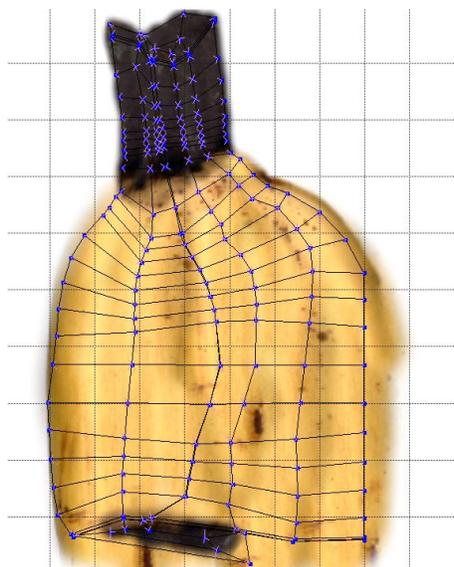
In Material Shader wird eine Vielzahl von Phänomene berechnet. Die Berechnung von Glanzlichtern ist Aufgabe des Material Shaders. Auch Reflexionen, Transparenz und Lichtbrechung wird in Material Shadern berechnet. Eine Oberfläche wirkt weitaus überzeugender, wenn sich andere Gegenstände darin spiegeln. Dies erzeugt einerseits einen edlen Look und trägt wie auch Glanzlichter dazu bei, dass man beim Betrachten einen raschen Eindruck über die Oberflächenbeschaffenheit bekommt. Wird die Reflexion mit der Originalfarbe des Objektes gemischt, kann man verschiedene Effekte damit erzielen: Je stärker die Reflexion, desto glatter und metallischer wirkt eine Oberfläche. Sanfte Reflexionen können Objekten mit Holztexturen einen guten „Lack“ verpassen.

### **A.3.2 Texture Shader**

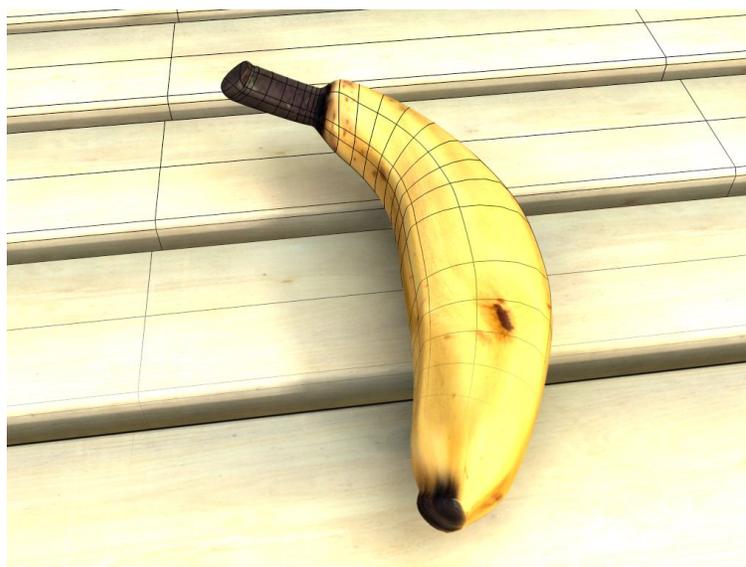
Texturen machen Oberflächen erst interessant. Sie verleihen ihnen ein abwechslungsreiches Aussehen. Texturen sind entweder Bilder, die von der Festplatte geladen werden oder prozedural erzeugte Muster. Prozedurale Texturen stammen nicht von Bilddateien, sondern ihnen liegt ein Algorithmus zugrunde der die Berechnung von Farb- und Helligkeitswerten beschreibt. Im Regelfall kann man gezielt einzelne Parameter der prozeduralen Textur verändern und so ihr Aussehen maßgeblich beeinflussen. Ein Standardwerk für alle Interessenten für dieses Thema ist die Arbeit „An Image Synthesizer“ von Kenneth Perlin oder die Website [www.noisemachine.com/talk1/](http://www.noisemachine.com/talk1/), auf der Perlin die Basics der Prozeduralen Texturgenerierung erklärt.

Nachdem Objekte, sofern es sich nicht um glatte Fußböden oder Wände handelt, zumeist nicht in einer

Ebene liegen ist das Anbringen einer Textur auf einem solchen Objekt keine triviale Sache. Es spezielle Verfahren, wie man 2-Dimensionale Texturen auf 3-Dimensionalen Objekten anbringt. Diesen Vorgang nennt man Projektion. Es gibt automatische Projektionsmethoden welche die Textur durch zum Beispiel zylindrische Projektion um das Objekt wickeln. Die andere Methode und bei komplexen 3D-Objekten unumgängliche Methode ist, die Koordinaten manuell einzurichten. Dabei werden in mühevoller Kleinarbeit alle Polygone händisch auf der zweidimensionalen Textur zurechtgerückt. Die Punkte erhalten dabei einen Koordinatenwert. Dieser Koordinatenwert wird als UV-Koordinate bezeichnet.



**Abbildung 9:** UV-Map auf einer vorbereiteten Textur



**Abbildung 10:** Texturiertes Objekt mit sichtbaren Polygonen.

Das UV-Koordinatensystem entspricht dem Karthesischen Koordinatensystem mit X und Y. Der Unterschied ist, dass die Texturkoordinaten zwischen 0.0 und 1.0 liegen müssen. Dies hat den Vorteil, dass Texturkoordinaten unabhängig von der Auflösung des darunter liegenden Bildes sind. Lauten die UV-Koordinaten für einen Punkt 0.2 und 0.75, so wird dieser Punkt auf einer 512x512-Textur auf die Pixelkoordinaten 102.4 und 384 umgerechnet. Wird diese Textur gegen eine Textur mit anderem Seitenverhältnis ausgetauscht, werden die Koordinaten neu berechnet und wieder auf die entsprechenden Pixelwerte gemappt<sup>5</sup>.

Man kann Texturen nicht nur zur Steuerung von Farbwerten verwenden, sondern auch um andere Parameter eines Shaders damit zu speisen. So kann etwa die Reflexionsfarbe oder –intensität mit einer

<sup>5</sup> Mapping ist der Prozess, bei dem feste Koordinatenwerte durch Skalieren an Bilder oder geometrische Objekte angepasst werden. Mapping bedeutet, dass eine Vorlage in ein anderes System umgelegt wird, in diesem Fall UV-Koordinaten auf XY-Positionen in der Bildvorlage.

Textur gesteuert werden.. Mehr dazu im Kapitel "Shader Assignment, Rendertrees, Shadergraphs, Phenomena", Seite 41.

### **A.3.3 Light Shader**

Es gibt in Softimage 3 Arten von Lichtquellen: Parallele Lichter, Punktlichter und Spotlights. Diese 3 Lichtquellentypen unterscheiden sich in einigen Merkmalen, die hier genannt sind.

Eine parallele Lichtquelle simuliert eine unendlich weit entfernte Lichtquelle – und erzeugt so den Eindruck von Sonneneinstrahlung. Parallele Lichtquellen besitzen daher keinen Ursprung, sondern haben nur eine Richtwirkung. Es ist also wichtig, dass man diese Lichter richtig rotiert, damit sie in die korrekte Richtung zeigen. Es ist in Softimage egal, wo man eine Parallele Lichtquelle in der Szene positioniert da Mental Ray die Position ignoriert. Ausschlaggebend ist nur ihre Richtung in die sie zeigen.

Punktlichter strahlen das Licht in alle Richtungen gleichmäßig ab, besitzen daher keine Richtwirkung. Dafür ist es aber wichtig, wo die Lichtquelle im Raum positioniert ist.

Spotlights haben sowohl eine Richtwirkung, als auch einen Ursprung der für Berechnungen von Bedeutung ist.

Im Light Shader wird diesen Eigenschaften Rechnung getragen. Soll ein Material Shader auf seiner Oberfläche Lichteinstrahlung mit in Betracht ziehen, ruft er den Light Shader der entsprechenden Lichtquelle(n) auf. Der Light Shader muss seinerseits erkennen, um was für eine Lichtquelle es sich handelt und seinen Farbwert am Schnittpunkt mit der entsprechenden Oberfläche zurückgeben [MRDOC1, 2003, node120.htm]. Light Shader bieten unter anderem folgende Parameter: Farbe, Intensität, Falloff über die Entfernung, ob die Lichtquelle Schattenwurf erzeugen soll oder nicht und welche Restlichtstärke ein Schatten mit voller Intensität aufweisen soll. Spotlights besitzen zusätzlich eine innere Breite, bei der man zusätzlich den Falloff zum Rand hin einstellen kann. Es ist auch möglich Bilder ähnlich einem Diaprojektor in die Szene zu projizieren. Dies kann man allerdings auch im Rendertree von Softimage bewerkstelligen.

Neue Light Shader zu entwickeln ist eine eher überflüssige Sache, da es wohl kaum innovative Ideen gibt die man auch ohne Programmieraufwand umsetzen kann. Eine Möglichkeit wäre ein Light Shader, bei dem man selber die Lichtintensität über die Entfernung einstellen kann. Das kann aber auch im Rendertree von Softimage einfach nachgestellt werden.

### A.3.4 Shadow Shader

Diese Art von Shader ist nichts anderes, als ein abgespeckter Material Shader. Soll eine Lichtquelle Schatten werfen, sucht Mental Ray nach Objekten im Strahlengang des Lichtes zum Schnittpunkt mit der Materialoberfläche. Wird dabei festgestellt, dass sich ein Objekt im Strahlengang befindet, wird der Shadow Shader des Objektes aufgerufen. Dieser gibt zurück ob und wie durchsichtig das Objekt ist. Ein Shadow Shader macht nichts anderes als die Intensität des Lichtes zu vermindern. Ist das Objekt gänzlich undurchsichtig, kann die Schattenberechnung für die Lichtquelle abgebrochen werden. Der Shadow Shader gibt an den Light Shader zurück, dass das Licht nicht bis zum Schnittpunkt vordringen konnte. Der Light Shader seinerseits gibt den Wert "schwarz", also  $RGB = 0.0$  an den aufrufenden Material Shader zurück. Alternativ dazu kann der Light Shader den Wert des Restlichtes, welcher in den Shader von Softimage|XSI *Umbr*<sup>6</sup> genannt wird, zurückgeben.

Shadow Shader kommen nur zum Einsatz wenn Schattenwurf für die entsprechende Lichtquelle aktiviert wurde.

### A.3.5 Volume Shader

Diese Shader simulieren Partikel, die sich in der Luft (dem Raum innerhalb eines geometrischen Objektes) befinden. Um Volume Shader zu sehen, muss man die Transparenz des Material Shaders erhöhen da man sonst nicht in das Innere des Objektes sieht. Alternativ können Volume Shader auch an die Kamera angehängt werden. In diesem Fall ist keine Umgebende Geometrie erforderlich, sondern es wird von jedem Strahl der die Kamera verlässt angenommen, dass er sich in einem Volume befindet. Volume Shader bedienen sich oft der Technik des „Ray Marching“ (mehr dazu im nächsten Kapitel) um nichthomogene Volumes zu simulieren – zum Beispiel in der Luft wabernde Nebel- oder Rauchschwaden. Diese Schwaden werden ähnlich wie prozedurale Texturen berechnet. Meistens handelt es sich dabei um die 3-Dimensionale Perlin-Noise. Will man diese Schwaden auch über die Zeit animieren, ist eine vierte Dimension nötig. Man kann sich gut vorstellen, dass solche Shader etwas langsamer sind, besonders wenn man weiß, dass die Raymarchingtechnik das Raytracing zusätzlich verlangsamt, weil ein Ray zusätzlich in noch feinere Segmente zerlegt wird. Dafür ermöglicht diese Technik Effekte, die anders nicht machbar wären.

<sup>6</sup> Je nach eingestelltem Umbr-Wert gelangt ein bestimmter Bruchteil des Lichtes zur Materialoberfläche und erhellt diese, selbst wenn die dazwischen liegenden Objekte undurchsichtig sind.

## A.3.6 Photon Shader

Wie schon im vorigen Kapitel erwähnt, ermöglicht Photon Tracing die Berechnung von indirekter Beleuchtung. Dazu bedarf es zweier Dinge:

- Einer Lichtquelle, welche definiert wie viele Photonen von dieser Lichtquelle in die Szene gefeuert werden, welche Intensität und welche Farbe sie haben.
- Materialien, die mit diesen Photonen interagieren. Diese Interaktionen sind Reflexion, Absorption, Brechung oder das Speichern der Photonenenergie in einer Dreidimensionalen Datenstruktur, einer Photon Map. Diese Shader werden Photon Shader genannt. [MRDOC, node142.htm, node122.htm]

Photonen werden erst ab der zweiten Reflexion in der Photon Map gespeichert. Das liegt daran, dass die Material Shader die direkte Beleuchtung berechnen.

Wie ein Photon Shader die Lichtteilchen handhabt, liegt wie bei allen selbstprogrammierten Shadern, in der Verantwortung des Shaderwriters. Es liegt auch am Shaderwriter zu entscheiden ob und wann der Photon Shader Photonen abspeichern soll.

Der Vorgang beim Photon Tracing sieht folgendermaßen aus: Erst werden Photonen einer Lichtquelle in die Szene gefeuert und die beteiligten Photon Shader reflektieren und speichern die Photonen. Wenn die in der Lichtquelle eingestellte Anzahl an Photonen erreicht ist, ist der Vorgang des Photontracings abgeschlossen und der nächste Renderschritt kann beginnen. Stößt nun ein Material Shader eines Objektes auf gespeicherte Photonen die der Photon Shader ermittelt hat, kann er diese gespeicherten Photonen in seine Berechnungen mit einbeziehen – ob er das tatsächlich tut kann der Material Shader selbst entscheiden und wird nicht automatisch erledigt.

Die Anzahl der Photonen, die in eine Szene geschossen werden ist im Vergleich zur den Milliarden, die pro Sekunde in einem Raum permanent reflektiert, absorbiert und gebrochen werden, gering. Vernünftige Werte bewegen sich zwischen 10000 und 100000 Photonen pro Lichtquelle, wobei für hochqualitative Renderings der Photonenanzahl nach oben keine Grenze gesetzt ist – der Renderzeit allerdings auch nicht. Dafür erzielt man mit Photon Shadern einen Look, der sonst kaum zu erreichen ist.

Photonen werden übrigens mit ihrer vollen Energie abgespeichert die sie beim Auftreffen auf die Fläche hatten. Dennoch spielt der Winkel mit dem sie auf der Oberfläche auftrafen wie beim Shading in Material Shadern eine maßgebliche Rolle: Je steiler der Einfallswinkel, desto schwächer ist der Beleuchtungsbeitrag des Photons.

### **A.3.7 Photon Volume Shader**

Im Englischen werden diese Shader auch „participating media“, also am Lichttransport beteiligte Medien bezeichnet. Während Photon Shader die Interaktion von Photonen mit Materialoberflächen berechnen, erledigen Photon Volume Shader Interaktion von Photonen in Volumes. Wie man aus der Bezeichnung erkennen kann vereint ein Photon Volume Shader die Merkmale sowohl von Photon Shader als auch von Volume Shader. Ein Photon Volume Shader simuliert Licht, das durch kleine Partikel in der Luft gestreut wird und so mit diesen Partikeln interagiert. Solche Partikel können Staub, Rauch oder kleine Wassertröpfchen sein. Einen Photon Volume Shader zu schreiben ist wohl eines der komplexesten Themen auf dem Sektor Shaderwriting.

### **A.3.8 Geometry Shader**

Normalerweise modelliert man Geometrische Objekte selber. Man bearbeitet sogenannte Primitives wie Kugeln, Würfel, Tori, Zylinder. Natürlich muss man auch jeden einzelnen Eckpunkt und jedes Polygon einer Szene speichern. Das benötigt bei komplexen Formen sehr viel Speicherplatz. Diesen Speicherplatz kann man sich ersparen, indem man Geometry Shader verwendet: Diese erzeugen Geometrie prozedural, also mit Hilfe bestimmter Algorithmen. Der Shader wird mit der Basisinformation versorgt und erzeugt die Geometrie automatisch. Ein Geometry Shader für eine Kugel benötigt beispielsweise die Angabe für den Radius und mit wie vielen Polygonen die Oberfläche angenähert wird. Je größer dieser Wert ist, desto mehr Polygone werden zur Renderzeit erzeugt.

Geometry Shader spielen eine untergeordnete Rolle in XSI und Mental Ray, aber in einigen Sonderfällen können sie ganz nützlich sein.

Gerade bei Geometry Shader wird klar, wie freizügig der Begriff „Shader“ in der 3D-Branche verwendet wird, da ein Geometry Shader primär nichts mehr mit der Berechnung von Farbwerten während des Renderingprozesses zu tun hat. außerdem werden Geometry Shader in der Preprocessingphase berechnet, also bevor das aktive Rendering beginnt.

### **A.3.9 Lens Shader**

Das sind Shader welche auf die virtuelle Kameralinse Einfluss nehmen. Normalerweise besitzt die virtuelle Kamera keine Brennweite. Die Belichtung findet daher an einem einzigen Punkt statt. Vorteil dabei ist, dass man nie unscharfe Bilder erhält. Doch manchmal ist es durchaus erwünscht das Verhalten einer realen Kamera nachzubilden. Echte Kameras haben eine Optik die das Bild etwas verzerrt (jedem sind die Fischaugenaufnahmen von Kameras bekannt). Durch die Brennweite des Objektivs können einzelne Gegenstände gezielt fokussiert werden während die Objekte davor und dahinter unscharf zu

sehen sind.

Mit Lens Shadern kann man dieses Verhalten simulieren. Dem nicht genug – man kann mit Lens Shadern auch reales Aufnahmematerial simulieren. Gerade hier liegt noch großes Potenzial, denn eine der Stärken und paradoxerweise gleichzeitig Schwächen der digitalen Bilderzeugung ist, dass Renderer mathematisch korrekt arbeiten und 100% gestochen scharfe Bilder produzieren. Das ist manchmal gar nicht erwünscht. Bei vielen Kinoproduktionen muss künstlich hergestelltes Videomaterial mühsam an den Look des Zelluloidfilmes angepasst werden: Bildrauschen, Gradationskurven, verschiedene Empfindlichkeiten auf Farbe sind Kennzeichen echten Videomaterials [WHP053, 2002, s. 1-15]. Mit Lens Shadern kann man diese Schritte von vornherein umgehen, indem man einen Lens Shader programmiert, der das Belichtungsverhalten von echtem Filmmaterial genau nachbildet.

### **A.3.10 Output Shader**

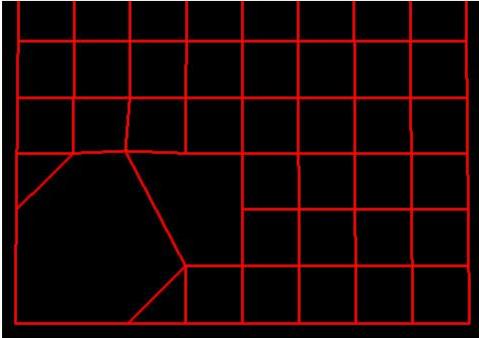
Diese Shader sind Postprocessing-Shader: Sie werden ausgeführt, nachdem der aktive Rendervorgang abgeschlossen ist und das gesamte Bild gerendert vorliegt. Output Shader kann man sich grob gesprochen als Effektfiler vorstellen, wie sie Zeichenprogramme mit sich bringen. Zum Beispiel könnte man ein Bild nach dem Rendern in Schwarzweiß-Werte umrechnen, oder das Bild weichzeichnen lassen. Das ist allerdings noch nicht die Stärke von Output Shadern – denn diese Funktion wird von allen Bildbearbeitungsprogrammen schon geboten. Output Shader haben Zugriff auf spezielle Informationen, die während des Renderns in sogenannten „Buffers“ gesammelt werden. Ein Framebuffer ist ein weiteres Bild mit den selben Pixeldimension des Zielbildes. Doch in einem Framebuffer lassen sich Informationen abspeichern, die grafisch nicht dargestellt werden können. Beispielsweise kann in einem Framebuffer der Tiefenwert eines Pixels abgespeichert werden, welches Objekt das Pixel belegt oder wie die Oberflächennormalen an diesem Pixel lauten. Auf diese Zusatzinformation haben Output Shader Zugriff und können dadurch spezielle Effekte erzielen: Zum Beispiel kann ein Bild abhängig von dem Tiefenwert der Pixel weichgezeichnet werden. Dieser Post-Effekt wird gerne verwendet, um rasch die Tiefenunschärfe von Kameras zu simulieren.

### **A.3.11 Contour Shader, Contour Contrast Shader, Contour Store Shader**

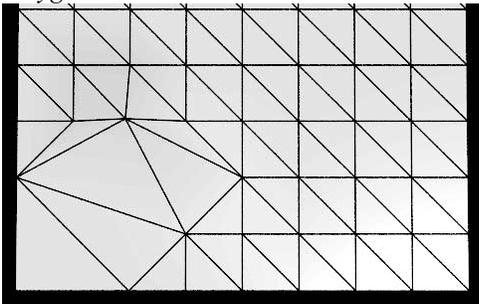
Dies sind Shader die einen cartoonartigen Look produzieren. Dieser werden wir im Rahmen dieser Arbeit aus zwei Gründen nicht durchnehmen: Erstens ist der Bereich des Contour Renderings sehr umfangreich, zweitens sind diese Shader schon sehr ausgereift, sodass es wenig Bedarf gibt neuartige Contour Shader zu schreiben.

## A.4 Renderbegriffe

Die folgenden Seiten erklären die gängigsten Renderbegriffe, und worin sich die verschiedenen Techniken unterscheiden. Die Unterschiede zwischen dem Raytracing- und dem Scanline-Algorithmus werden dargestellt. Die Technik des Raymarchings, welche gleichzeitig mit dem Raytracing angewendet werden kann, wird ebenfalls dargestellt. Danach wird das Sampling veranschaulicht, welches dem Aliasing vorbeugt.



**Abbildung 11:** In Softimage erstellte Geometrie mit noch nicht triangulierten Polygonen.

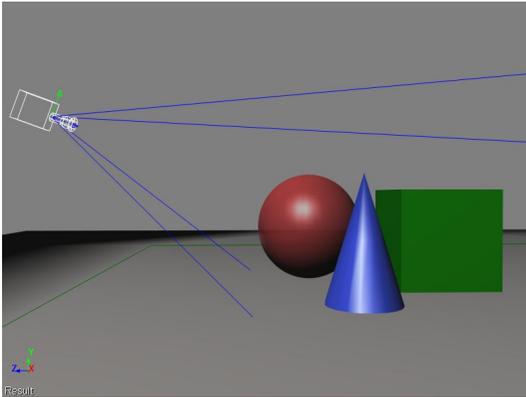


**Abbildung 12:** Dieselbe Geometrie, von Mental Ray trianguliert

Der Vorgang des Renderings ist der Prozess, bei dem alle ausgewerteten Daten in ein sichtbares Bild verarbeitet werden. Mental Ray kennt eine Vielzahl an Optionen die zur Renderoptimierung eingestellt werden können. In Softimage kann man diese Optionen über die Propertypage (PPG) „Render Options“ erreichen. Zuerst wertet Mental Ray die eingestellten Optionen aus und beginnt dann mit dem Preprocessing der Szene. Das sind Arbeiten vor der aktiven Pixelberechnung erledigt werden müssen. Diese Arbeiten umfassen Aufgaben wie das Auswerten der Geometry- und Displacement Shader. Danach wird die Geometrie in lauter Dreiecke aufgebrochen. Mental Ray kann keine Polygone mit mehr als 3 Ecken handhaben und besitzt daher interne Algorithmen, die vor dem Rendervorgang alle Flächen mit mehr als 3 Eckpunkten in Dreiecke zerlegen. Auf der seitlichen Abbildung ist dies deutlich zu erkennen.

Erst wenn die Geometrie verarbeitet und in den Arbeitsspeicher verfrachtet wurde, kann mit dem Rendering begonnen werden.

Weitere Preprocessing Schritte sind das Erstellen von Shadowmaps (welche in dieser Arbeit nicht behandelt werden), Photontracing und die Berechnung von Final Gathering. Danach erfolgt das, was die meisten Menschen unter Rendering verstehen: Die aktive Berechnung des Bildes, bei der man zusehen kann, wie das Bild Schritt für Schritt errechnet wird. Mental Ray besitzt, wie der Name der Software schon erkennen lässt, die Fähigkeit des Ray-Tracings.



**Abbildung 13:** Virtuelle Kamera, die auf einige Objekte gerichtet ist. Die blauen Linien zeigen die Sichtpyramide an.

## A.4.1 Raytracing vs. Scanline Rendering

### A.4.1.1 Raytracing

Raytracing ist ein Renderingalgorithmus, der die Farbwerte für ein Pixel errechnet, indem er Strahlen von der Kamera aus in die Szene schießt. Interessant dabei ist, dass dies genau umgekehrt zur Realität ist: Dort treffen Lichtstrahlen auf die Optik der Kamera, welche den Film belichtet. Beim

Raytracing ist es genau umgekehrt. Die Strahlen werden von der Kamera aus in die Szene geworfen. Trifft ein

Strahl auf ein geometrisches Objekt, so heisst der Punkt an dem der Strahl auf die Oberfläche trifft

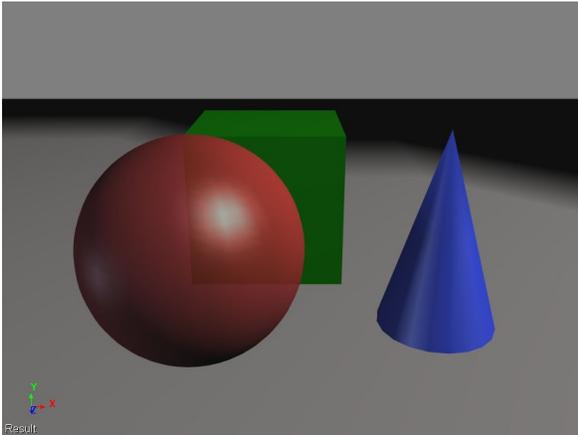
*Intersectionpoint*. Das Objekt wertet an dieser Stelle seinen Oberflächenshader aus. Der

Oberflächenshader ruft weitere Shader ins Spiel, wie zum Beispiel Light Shader. Light Shader rufen, wenn angefordert, Shadow Shader auf. Shadow Shader werden für alle Objekte aufgerufen, die sich zwischen der Lichtquelle und dem Intersectionpoint befinden.

Ein Oberflächenshader kann auch weitere Strahlen, sogenannte Reflexions oder Brechungsstrahlen erzeugen, die tiefer in die Szene vordringen und denselben Vorgang von neuem starten. Hat ein Shader weitere Shaderaufrufe zur Folge, muss dieser Shader auf die Ergebnisse dieser Shader warten. Somit ist ein Lens Shader der erste Shader der aufgerufen wird und zugleich der letzte der mit seiner Arbeit fertig wird, weil davor alle anderen Shader abgearbeitet werden müssen.

So wird ein Baum aufgebaut, der nach und nach durchlaufen wird, bis der Renderer wieder beim allerersten Shader angelangt ist. Danach kann der nächste Ray in die Szene geworfen werden. Eine anschauliche Grafik ist dazu auf Seite 121 zu sehen.

Aufgrund der Tatsache, dass Strahlen von der virtuellen Kamera und nicht von der Lichtquelle erzeugt werden, wird diese Technik mit "Backwards Raytracing" bezeichnet. Strahlen die von der Kamera aus gehen werden "Primary Rays" genannt. Aller weiteren, durch Lichtbrechung und Reflexion erzeugten Strahlen heissen "Secondary Rays". Diese Funktionalität kommt natürlich mit gewissen Kosten, da die Berechnungen, ob nun ein Strahl ein Objekt trifft oder nicht einen gewissen Rechenaufwand erfordern.



**Abbildung 14:** Blick aus der virtuellen Kamera auf die Objekte. Aus dieser Perspektive wird Mental Ray das Bild berechnen

#### A.4.1.2 Scanline Rendering

Aus diesem Grund gibt es eine zweite Technik namens "Scanline Rendering". Diese projiziert zuerst alle Objekte auf die abzubildende Fläche und sortiert diese danach entsprechend ihren X und Y-Koordinaten. Das Rendering wird dadurch beschleunigt, indem Mental Ray jetzt nur mehr diese sortierte Liste verfolgt, was in der Regel schneller ist. Doch diese Technik ist auch mit Nachteilen behaftet: Scanline Rendering unterstützt keine Reflexionen oder Lichtbrechung – lediglich einfache Transparenzen lassen sich mit dem Scanline Renderingalgorithmus verwirklichen. Dafür

ist eine 3D-Szene ohne Spiegelungen oder Lichtbrechungen viel schneller zu berechnen, als eine Szene mit. Aufgrund dieser Tatsache verwendet Mental Ray immer Scanline Rendering für Primary Rays – erst bei Richtungsänderungen durch Reflexion oder Lichtbrechung wird auf Ray Tracing umgeschaltet.

[MRB, 2002, s. 14f]

Ein weiterer Grund für den höheren Zeitaufwand von Raytracing ist, dass die Zahl der Strahlen manchmal explodieren kann. Dies sei anhand folgenden Beispiels demonstriert: Eine leicht transparente und spiegelnde Kugel bricht einen Teil des Lichtes, der auf ihre Oberfläche trifft, den anderen Teil reflektiert sie.

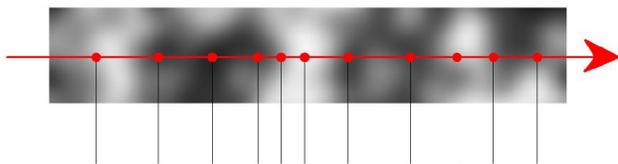
Trifft nun ein Primary Ray die Oberfläche der Kugel, passiert folgendes: Es werden 2 neue Rays erzeugt von denen der erste reflektiert und der andere gebrochen wird. Diese beiden neuen Rays sind nun Secondary Rays die beide unabhängig voneinander agieren. Trifft der Reflexionsstrahl nun auf ein solides Objekt, kann dort die Farbe des Objektes berechnet werden und der Ray hat sein Ende gefunden. Trifft der Ray jedoch auf ein weiteres, transparent/spiegelndes Objekt, werden weitere Rays erzeugt, die nun ihrerseits berechnet werden müssen. Damit dieses Spiel nicht ad Infinitum weitergeht, kann man eine maximale Strahltiefe in den Rendersettings einstellen. Wurde ein Ray nun zu oft reflektiert oder gebrochen, wird der Berechnungsvorgang abgebrochen und als Ergebnis der Farbwert "schwarz" zurückgegeben. Natürlich können andere Strahlen mehr Erfolg gehabt haben und so werden die Ergebnisse in den entsprechenden Verhältnissen zueinander gewichtet und zusammengemischt.

### A.4.1.3 Raymarching

Raymarching ist wohl die rechenintensivste Operation die ein Shader ausführen kann. Wie der Name schon sagt, wird ein Ray auf seiner gesamten Länge durchwandert und noch einmal in feinere Elemente unterteilt. An jedem dieser Teilungspunkte wird sein Wert berechnet und zum Gesamtergebnis hinzuaddiert. Raymarching ist keine Funktion welche die Rendersoftware anbietet sondern muss eigenständig als Algorithmus in einem Volume Shader implementiert werden. Diese Technik kommt in allen Volume Shader, welche inhomogene Volumen simulieren, zum Einsatz. Man muss beim Einstellen dieser Volume Shader extrem vorsichtig sein. Zu grobe Einstellungen flackern in Animationen stark und zu feine Einstellungen verlängern die Renderzeit beträchtlich.

Den Vorgang in einem Volume Shader muss man sich folgendermaßen vorstellen: Das Objekt durch den der Strahl reist, ist mit einer 3-Dimensionalen prozeduralen Textur versehen die an jeder Stelle in dem Objekt andere Werte für die Dichte an der entsprechenden Position aufweist. Der Raymarching-Algorithmus testet in regelmäßigen Abständen entlang des Rays die Werte und summiert diese auf. Ist in dem Shader die Interaktion mit Lichtquellen ebenfalls definiert, muss die Intensität des Lichtes an der Stelle ebenfalls getestet werden.

Unterscheiden sich die Helligkeitswerte zwischen zwei Messpunkten zu stark, wird die Distanz noch einmal halbiert und dazwischen ein weiterer Messwert genommen, bis entweder eine bestimmte Rekursionstiefe erreicht ist, oder der Unterschied kleiner als der Threshold ist.



**Abbildung 15:** Ein Ray reist durch ein Volumen mit inhomogener Dichte. Die roten Punkte zeigen die Unterteilungen des Rays an

Die Raymarching-Technik kann also nur auf begrenztem Raum stattfinden. Rays welche die Szene verlassen und ins Unendliche gehen kann ein Raymarcher nicht verarbeiten. Deswegen wird der Effekt entweder in Volumen von Objekten verwendet oder man definiert im dem

Shader eine Boundingbox innerhalb derer der Effekt ausgewertet wird. Die Grafik zeigt einen Ray der durch ein Volumen reist und die Messpunkte entlang des Rays.

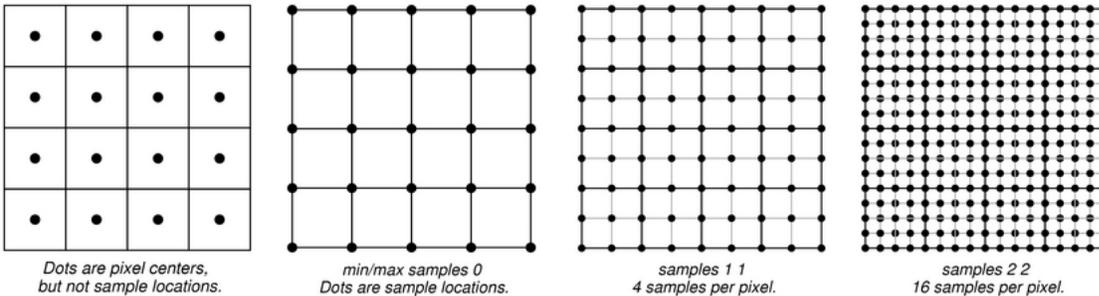
Dieser Mechanismus wurde in folgender Datei untersucht, die von dem FTP-Server von Mental Images bezogen werden kann:

<ftp://ftp.mentalimages.com/data/pub/shaders/base.zip> Die Datei mit dem Raymarching-Code heisst "baseraymarch.c".

In diesem Archiv befindet sich übrigens der Shadercode sämtlicher Mental Ray Base-Shader. Diese dürfen zum Studium und zur Weiterentwicklung verwendet werden.

## A.4.2 Sampling

Wenn Mental Ray einen Pixelwert berechnet, so schießt er meistens nicht nur einen, sondern gleich mehrere Rays in die Szene – und bekommt so mehrere Werte aus denen das Programm den Durchschnitt berechnet und den gewonnenen Farbwert in das Pixel schreibt.



**Abbildung 16:**  
Pixelraster mit  
gekennzeichneten  
Pixelmittelpunkten

**Abbildung 17:**  
Samplingpunkte liegen an  
den Ecken der Pixel

**Abbildung 18:**  
Oversampling mit  
Minimum-Maximum 1 1

**Illustration 19:**  
Oversampling mit  
Minimum-Maximum 2 2

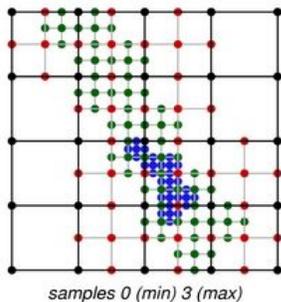
Ein Sample ist eine Farbwertberechnung für eine bestimmte Position im Bild wobei diese Position ist nicht das genaue Zentrum des Pixels ist. Unter Sample ist der gesamte Berechnungsvorgang eines Strahls von der Kamera zum Objekt, inklusive aller Secondary Rays zu verstehen [BPSMP, 2005]. Ein Pixelfarbwert wird meist aus mehreren Samples gemittelt. Dies nennt sich Oversampling.

Wie man in der Abbildung deutlich erkennen kann, liegen die Samplepunkte an den Eckpunkten des Pixels. Durch diesen Trick beugt man dem Aliasing vor. Aliasingfehler äußern sich bei Animationen dadurch, dass das Bild grieselt oder feine Strukturen im Bild zu flackern beginnen.

Je feiner die Samplingeinstellungen für den Renderer getroffen werden, desto weichere, genauere Bilder bekommt man. Die Samplinggenauigkeit kann man bei -3 ansetzen (das ist ein Sample für 8x8 Pixel) und bis 6 hochschrauben. Doch hohe Samplingwerte haben ihren Preis: Die Anzahl der berechneten Samples wächst exponentiell mit der Formel  $2^{2 \times N}$  mit N gleich dem Samplingwert.

<i>Samplingstufe</i>	<i>Berechnete Samples pro Pixel</i>
-3	1/64
-2	1/16
-1	1/4
0	1
1	4
2	16
3	64
4	256
5	1024
6	4096

Wie man in der Tabelle gut erkennen kann, hat die Erhöhung der Samplingeinstellungen eine rapide Zunahme der zu berechnenden Samples zufolge.



**Abbildung 20:** Samples bei Adaptive Sampling mit Minimum-Maximum 0 3

#### A.4.2.1 Adaptive Sampling

Mental Ray nutzt auch eine spezielle Samplingtechnik namens „Adaptive Sampling“. Der Renderer nimmt für das aktuelle Pixel erst weitere Samples, wenn der Farbunterschied zwischen 2 Samples einen gewissen Schwellwert, den *Threshold* überschreitet. So werden einfarbige Flächen ohne scharfe Kanten schneller berechnet da weniger Samples genommen werden müssen als Flächen, auf denen die Farben schnell kontrastreich sind.

Deshalb kann man in Mental Ray 2 Samplingwerte einstellen: Einen unteren und einen oberen. Der erste Wert gibt die Untergrenze, der zweite die Obergrenze an. Lauten die Werte 0 und 3, so entscheidet Mental Ray je nach Kontrastwerten zwischen 2 Samples ob er den Bereich noch feiner unterteilen soll, das heisst, noch mehrere Samples nehmen muss. Ist der Kontrast schwächer, werden keine weitere Samples mehr genommen. Dies ist in der seitlichen Abbildung gut zu erkennen.

Die Zahl der Samples die berechnet wurde ist um ein vielfaches kleiner als ohne dieser Einstellung da nur nach Bedarf eine höhere Samplingdichte verwendet wird.

Ein Bild besteht aus X mal Y Pixel: Ein Pixel berechnet sich aus einer Menge von Samples die Ihrerseits pro Sample einen Primary Ray auslösen. Die Berechnung nur eines einzigen Rays hat eine Vielzahl an Shaderausruufen zur Folge (Material, Light, Shadow,

Volume...). Ein Primary Ray löst in Szenen mit Transparenzen und Spiegelungen eine Vielzahl an Secondary Rays aus die nun ihrerseits weitere Shader aufrufen und möglicherweise weitere Secondary Rays erzeugen bis sie entweder auf ein nichttransparentes, nichtspiegelndes Material stoßen oder sie ihre maximale Strahltiefe erreichen und somit abgebrochen werden.

## A.5 Die Mental Ray-Architektur

*In diesem Kapitel wird Allgemeines über Mental Ray berichtet*

„Es gibt keine langen Renderzeiten sondern nur User, die schlechte Rendereinstellungen treffen.“

Die meisten Anwender sehen in Mental Ray nur ein Mittel zum Zweck. Mental Ray ist dafür verantwortlich, dass eine 3D-Szene passend gerendert wird. Es wird dem Renderer oft die Schuld gegeben, lange zu rendern. Der Grund dafür ist nicht dass die Software schlecht programmiert ist, sondern dass Mental Ray oft mit schlechten Einstellungen versorgt wird. Eine feine Adjustierung der Shader bringt manchmal einen enormen Zeitgewinn bei gleichbleibender Bildqualität mit sich.

Viele Benutzer haben nur eine ungefähre Ahnung von Rendereinstellungen und äußern sich negativ über lange Renderzeiten, ohne sich einen Gedanken zu machen was der Grund dafür sein könnte. Durch feines Justieren der Rendersettings können mit Mental Ray Szenen bei gleichbleibender Qualität schneller gerendert werden. Es macht sich bezahlt einige Minuten damit zu verbringen, die Samplingeinstellungen und den Threshold zu optimieren. Was Sampling genau ist, konnte man im vorigen Kapitel erfahren. Genauso zahlt es sich aus bei Flächen mit gekachelten Texturen die Multitextures (besser als MIP-Maps<sup>7</sup> bekannt) bei gleichzeitigem Elliptical Filtering zu aktivieren. Sorgsam getroffene Shadereinstellungen und genau justierte Raytracing Depths werden zwar in der Literatur erwähnt, doch es ist dennoch einige Erfahrung nötig, um ein Gefühl dafür zu bekommen. Das gilt eigentlich für fast alle Einstellungen die man in Mental Ray treffen kann: Die Mental Ray Dokumentation zu lesen und mit den Settings herumzuexperimentieren ist die beste Möglichkeit das Verhalten von Mental Ray zu beobachten. Dabei eröffnen sich so manche Tricks und Kniffe.

Mental Ray bietet eine Vielzahl an Techniken zur Optimierung von Renderergebnissen und jedem Softimage|XSI-Benutzer sei die Lektüre „Rendering with Mental Ray®“ ans Herz gelegt. Dort wird auf die verschiedenen Techniken bis ins kleinste Detail eingegangen sodass keine Fragen mehr offen bleiben. Das Buch ist keine Anleitung um gute Rendereinstellungen zu treffen. Die Rendereinstellungen hängen zu

<sup>7</sup> Der Ausdruck MIP steht für "multum in parvo" (lat), also "viele im Kleinen".

stark vom Bildinhalt ab, als dass man eine Standardprozedur angeben könnte. Allerdings gibt es gewisse "Do's" und "Dont's", die einem durch beim Lesen dieses Buches näher gebracht werden. Mit dem Wissen das man aus dem Buch erhält, wird es leichter rasch gute Einstellungen zu treffen.

Nachdem in den vorigen Kapiteln einiges zum Grundverständnis von Renderern im Allgemeinen und deren Techniken beigetragen wurde, wird nun im weiteren auf Mental Ray eingegangen. Mental Ray ist in eine Vielzahl von Programmen eingebunden. Das bedeutet, es ist nicht mehr nötig Mental Ray von der Kommandozeile aus zu bedienen sondern die jeweilige Software bietet programmintern die Schnittstelle zu Mental Ray. Die Vorteile liegen dabei klar auf der Hand: Man muss nicht zwischen zwei Programmen hin- und herwechseln sondern kann direkt in der 3D- oder CAD-Software Bilder rendern lassen. Es ist Aufgabe der implementierenden Software Mental Ray anzusteuern und mit korrekten Daten zu versorgen.

## **A.6 Das Mental Ray Dateiformat**

Das native Mental Ray-Format sind .mi Dateien. In einer .mi Datei ist eine Szene mit all ihren geometrischen Objekten, Lichtern, Kameras, Shader und sonstigen Informationen in solcher Weise beschrieben, dass Mental Ray aus dieser Datei ein Bild berechnen kann. Der Aufbau einer .mi Datei ist kompliziert – eine Szene händisch zu tippen ist theoretisch zwar möglich, aber von ganz einfachen Testszenen abgesehen, zu aufwändig. Dieser Job wird von der implementierenden Software erledigt. Interessierte können sich in Softimage in den Renderoptions eine .mi-Datei erzeugen lassen. Unter den Renderingoptionen existiert ein Reiter mit der Aufschrift "Export mi2". Dort muss man Dateinamen angeben und die Option "Ascii-Output" anhaken und die Szene als .mi-Datei auf den Datenträger schreiben lassen.

Man wird schnell feststellen, dass es äußerst schwierig ist, eine mit diesem Kommando erzeugte Datei zu verstehen, geschweige denn zu bearbeiten. Die gebotene Information ist zu umfangreich und nur für geübte User, die nach unerwarteten Renderergebnissen die Szene genau inspizieren wollen, macht es Sinn sich so eine Datei näher anzusehen.

## A.7 Die Mental Ray Fabrik

Das Modell der Mental Ray Fabrik wurde vom Autor extra für diese Diplomarbeit ausgearbeitet. Es veranschaulicht die Arbeitsstufen, die Mental Ray während des Rendervorgangs durchläuft. Besonderes Augenmerk soll darauf gelegt werden, welche Operationen zu welchem Zeitpunkt ausgeführt werden. Nicht alle Shader kommen im gleichen Arbeitsgang zum Einsatz. Shader wie Displacement und Geometry Shader werden sogar vor der Sampleberechnung eingesetzt.

Die Arbeitsweise von Mental Ray ist sehr verworren. Der Großteil geschieht im Hintergrund ohne dass der Shaderwriter, geschweige denn der Standarduser etwas davon bemerkt. Eine gute Methode die Arbeitsweise von Mental Ray zu demonstrieren ist, dies anhand der „Mental Ray-Fabrik“ zu tun. Diese Illustration kommt der Realität ziemlich nahe – denn Mental Ray macht im Grunde genommen nichts anderes, als den Rohstoff, also die Szenenbeschreibung im .mi-Format, in ein Bild zu verarbeiten. Dabei werden wie in einer Fabriksfertigung verschiedene Fertigungsstufen durchlaufen bis das fertige Produkt, nämlich ein gerendertes Bild, herauskommt.

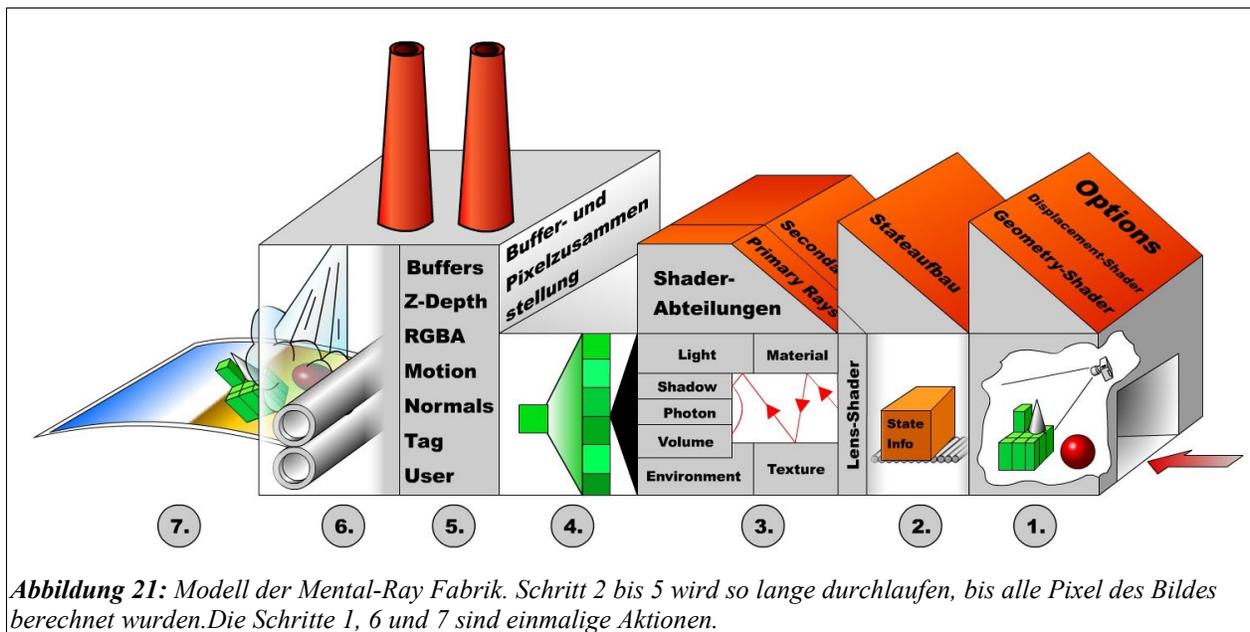


Abbildung 21: Modell der Mental-Ray Fabrik. Schritt 2 bis 5 wird so lange durchlaufen, bis alle Pixel des Bildes berechnet wurden. Die Schritte 1, 6 und 7 sind einmalige Aktionen.

Jeder hier beschriebene Schritt entspricht demselben Schritt, den Mental Ray tatsächlich im Computer ausführt. Der einzige Unterschied ist, dass das Beispiel stark abstrahiert ist, um den logischen Zusammenhang rascher begreifbar zu machen. Die Verarbeitungsrichtung verläuft von rechts nach links. Natürlich entspricht dieses Modell nicht 1:1 der Mental Ray Architektur, doch veranschaulicht es die

wichtigsten Vorgänge recht gut. Spezialfälle werden im Laufe dieser Arbeit noch aufgezeigt und behandelt.

1.) Der Rohstoff mit dem Mental Ray arbeitet, ist eine Datei im .mi-Format. In dieser Datei ist nicht nur die Information über die gesamte Szene enthalten, sondern auch Daten über den Auftrag, das heisst Rendereinstellungen. Diese können zwar von außerhalb überschieben werden (zum Beispiel durch Kommandozeilenargumente), doch nachdem Mental Ray kaum noch von der Kommandozeile aus bedient wird, ist dies selten der Fall.

Der Renderer lädt die Renderoptionen in den Speicher und beginnt die Szene nach Objekten zu durchsuchen, die entweder durch Geometry-Shader prozedural hergestellt werden müssen, oder von Displacement Shadern noch ihr Relief erhalten. Dabei werden alle Objekte so bearbeitet, dass sie nur mehr aus Dreiecken bestehen.

Es können in einer Szene zwar mehrere Kameras definiert sein, es kann jedoch nur eine dazu bestimmt werden, aus ihrer Perspektive das Bild zu rendern. Diese Kamera wird herausgesucht und als „Renderkamera“ aufgesetzt. Die angeforderten Buffer werden angelegt, um Informationen in sie hineinzuschreiben. Buffer liegen zur Schonung des Speichers auf dem Datenträger. Nun kann das Rendering fortgesetzt und die Samples berechnet werden.

2.) Für den Rendervorgang wird eine Datenstruktur, die sich „State“ nennt, hergerichtet. Diese Datenstruktur ist ein Informationspaket das während des gesamten Rendervorganges intensiv eingesetzt wird. In dieser Datenstruktur werden unzählige Information abgespeichert. In Kapitel B.1.2 auf Seite 71 wird auf den State noch genau eingegangen. Im State sind Informationen über Schnittpunkte, Koordinaten, Raylängen und so weiter festgehalten.

3.) Nun wird der erste Ray, der Primary Ray für das zu berechnende Sample in die Szene geschickt. Noch bevor der Ray die Kamera verlässt, wird darauf geachtet ob die Kamera eventuelle Lens Shader aufweist oder ob sie sich in einem Volume befindet. Vier Arten von Shader können der Kamera zugewiesen werden. Output-, Lens-, Volume- und Environment Shader.

Die Information über den ersten Ray steckt in dem eben vorbereiteten State-Paket. Wie schon erklärt wurde, bestimmt der Lens Shader über eventuelle Farbempfindlichkeit bzw. Form der Linse. Somit kann der Lens Shader schon beeinflussen, in welche Richtung der Strahl geschossen wird um Linsenverzerrungen zu simulieren. Diese Richtungsänderung wird im State festgehalten. Wurde in der Szene kein Lens Shader definiert, entfällt dieser Schritt.

Trifft nun der erste, von der Kamera ausgesendete Strahl auf ein Objekt, so wird an diesem Schnittpunkt der Material Shader dieses Objektes aufgerufen. All die Information die der Material

Shader braucht, steht im State: Wie die Koordinaten des Schnittpunktes lauten, woher der Strahl kommt der die Oberfläche trifft, wie lang er ist, wie die Oberflächennormale an dem getroffenen Punkt ausgerichtet ist und so weiter. Die Datenstruktur des State ist zwei gedruckte Din-A4 Seiten lang – dementsprechend viele Variablen sind dort zu finden.

Der State ist als ein Informationspaket zu betrachten, das zwischen den Shaderabteilungen hin- und hergereicht wird. Jede Shaderabteilung kann die Hilfe anderer Shaderabteilungen anfordern, um mit gewissen Problemen fertig zu werden. Wird die Oberfläche von Licht bestrahlt, fordert der Material Shader die Hilfe des Light Shaders an, der die Lichtfarbe und -intensität berechnet. Der Light Shader seinerseits möchte Schatten auf dem Objekt erzeugen. Das kann der Shader allerdings nicht selber, sondern fordert dazu die Hilfe eines Shadow Shaders an. Der Shadow Shader benötigt Information, ob er durch ein Volumen reist (also einer mit Partikeln gefüllten Atmosphäre) - und fordert nun seinerseits die Hilfe eines Volume Shaders an.

Bevor ein Shader die Hilfe eines weiteren Shaders anfordert, richtet Mental Ray eine neue Statestruktur für den aufzurufenden Shader her. Dies kann derselbe State des Shaders sein oder eine Kopie. Das heisst, in einigen Fällen wird genau derselbe State des vorigen Shaders weitergereicht oder eine Kopie weitergegeben. Shader, die von Mental Ray einen nichtkopierten State erhalten sind solche, von denen erwartet wird, dass sie das Ergebnis eines anderen Shaders verändern. Solche Shader sind zum Beispiel Lens, Shadow oder Volume Shader [MRDOC, 2003, node103.html].

Kopien werden weitergereicht, wenn zum Beispiel Reflexionsstrahlen oder Brechungsstrahlen erzeugt werden. In einem solchen Fall wird der neue State mit anderer Information gefüllt, wie zum Beispiel neuen Koordinaten für die Ray-Richtung, Ursprung und neuer Raylänge. Der neue Shader arbeitet nun mit diesem neuen State bis er mit seiner Arbeit fertig ist und gibt sein Ergebnis an den Shader, welcher ihn aufgerufen hat, zurück.

Je nach Typ des Shaders sind nicht alle Variablen des States für diesen definiert. Beispielsweise ist im State eines Light Shaders der ein Infinite Light<sup>8</sup> berechnen soll die Raylänge und der Ursprung der Lichtquelle nicht definiert. Fehlt einem Shader eine Information, so kann er den State des Shaders, der ihn beauftragt hat, nach Information durchsuchen – denn auf jedem Statepaket steht der Absender oder Auftraggeber. Ist der Shader mit seiner Aufgabe fertig, gibt er seine Ergebnisse wieder an den ihn aufrufenden Shader zurück. Dieser kann nun die erhaltenen Werte weiter verwerten und damit seine vorgeschriebenen Berechnungen durchführen.

---

8 In Infinite Light ist eine Lichtquelle die als unendlich weit entfernt angenommen wird.

- 4.) Ist der letzte Shader mit seinen Berechnungen fertig, ist das Sample fertig und wird in die Samplelist geschrieben. Jetzt das nächste Sample kann berechnet werden. Wenn alle Samples für ein Pixel vorliegen, werden diese zueinander gewichtet und man hat den fertigen Farbwert für das Pixel vorliegen. Bevor das Endprodukt, die Datei, hergestellt werden kann, müssen *alle* Pixel berechnet werden.
- 5.) Dazu werden die berechneten Werte kurzfristig in einem Buffer zwischengespeichert. Buffer existieren so lange, bis der Rendervorgang abgeschlossen wurde, oder sie mit einem expliziten Befehl auf die Festplatte geschrieben werden. Es existieren auch Buffer für nichtsichtbare Informationen, die beim Renderingprozess zum Teil automatisch generiert und nach dem Rendern, sofern nicht anders angegeben, wieder gelöscht werden. Weiters können User eigene Buffer anlegen, in welche sie selbst definierte Inhalte schreiben und abspeichern können.
- 6.) Bis das Bild komplett fertig gerendert wurde, kamen fast alle Shader zum Einsatz mit Ausnahme der Output Shader. Diese verpassen dem Bild den finalen Schliff, indem sie mit Hilfe der verschiedenen Buffer das Bild noch einmal nachbearbeiten und daran Änderungen vornehmen. Zum Beispiel kann ein Output Shader an Stellen, an denen der Raytracer kein Ergebnis brachte, weil ein Strahl die Szene verließ ohne auf ein Objekt zu treffen, eine Farbe anbringen.
- 7.) Nachdem der letzte Shader seine Arbeit verrichtet, hat ist das Bild fertig und kann im gewünschten Format abgespeichert werden.

Zusammenfassend kann das Folgende festgehalten werden:

Die Datenstruktur des States wurde so angelegt, dass alle Shader damit arbeiten können. Doch nicht jeder Shader definiert dieselben Variablen für den State. Zum Beispiel haben Geometry- oder Displacement-Shader KEINE Information über Rays, weil diese Shader mit dem Raytracingprozess nichts zu tun haben. Es ist besser zu wissen, welche Shader auf welche Variablen Zugriff haben als für jeden Shadertyp eine eigene Statestruktur zu definieren.

Der State wird zwischen den Shadern als Einheit zum Informationsaustausch genutzt. Benötigt ein Shader die Hilfe eines anderen Shaders, wird je nach Shadertyp der State zuerst kopiert und leicht verändert an den neuen Shader weitergegeben. Einzige Ausnahme sind Volume Shader. Sie bekommen keine Kopie des States, sondern gleich den letzten State, da sie dazu gedacht wurden, das Ergebnis des letzten Shaders nachhaltig zu verändern.

Rays die von der Kamera ausgehen, werden Eye/Lens-Rays genannt. Doch Rays sind in den seltensten Fällen vom Typ Eye/Lens, es gibt viele Typen von Rays, die im Anhang (s. 123) nachzuschlagen sind. Um welchen Typ von Ray es sich handelt, ist ebenfalls im State festgehalten.

## A.8 Wo man Programmierinformation über Mental Ray findet

Will man Shader für Mental Ray programmieren, ist der Einstieg in diesen Fachbereich ziemlich schwer. Meistens steht an Anfang die eine INTERNETRECHERCHE mit den Schlagworten „shader programming OR writing mental Ray“. Dabei stößt man auf interessante Tutorials die einem zeigen was mit Mental Ray möglich ist. Nimmt man sich die Zeit und Geduld, stellt man nach sturem Lesen und Abtippen fest, dass der gebotene Programmcode tatsächlich funktioniert. Deswegen hat man aber keinen tieferen Einblick in die Funktionsweise von Mental Ray bekommen. In den Online-Tutorials werden zwar vernünftige Beispiele gezeigt. Es fehlt allerdings eine genaue Beschreibung. Man kommt nicht darum herum, sich mit den Mental Ray Handbüchern über längere Zeit auseinanderzusetzen. Man bekommt sonst nirgends die genaue Funktionsweise von Funktionen wie `mi_eval_color()` erklärt.

Im ersten Teil dieser Arbeit wird noch nichts über die Datei "shader.h" gesagt. Eines sei jedoch vorweggeschickt: In dieser Datei sind alle Variablen und Funktionen für Mental Ray definiert. Sie muss in jeden Code mit eingebunden werden, da sonst ein selbstprogrammierter Shader nicht kompiliert werden kann.

Der erste Teil informiert nur darüber, welche Funktionalitäten Mental Ray bietet, wie das Interface dieses Renderers aussieht und wie bestimmte Funktionalitäten definiert wurden. Es wird nicht darauf eingegangen wie Shader selbst geschrieben werden. Dies wird in Teil 2 erörtert.

## A.9 Shader Assignment, Rendertrees, Shadergraphs, Phenomena

*Lange Zeit war es nur möglich, für ein Material einen einzigen Shader zu definieren, der einige Funktionalitäten bot. Hatte ein Shader eine gewisse Funktionalität nicht, so war es unmöglich den Shader zu erweitern. Man musste stattdessen einen anderen Shader verwenden, der die gesuchte Funktion bot. Konnte man in einem Material Shader keine Textur festlegen, so musste man auf diese Funktionalität verzichten. Das machte den Einsatz von Shadern sehr unflexibel. Die neuen Methoden, welche die bisherigen Probleme umgehen, sind in diesem Kapitel aufgelistet.*

### A.9.1 Shader Assignment und Shadergraphs

Mental Ray hatte die geniale Idee, den Einsatz von Shader flexibler zu gestalten. Die neue Technik nennt sich „Shaderassignment“. Was beim Shaderassignment passiert, ist nichts anderes als Wertzuweisung. Das Shaderassignment ermöglicht, dass ein Shader sein Berechnungsergebnis an einen weiteren Shader übergibt der mit diesem Ergebnis weitere Berechnungen durchführt.

Dies geschieht indem man dem Eingang des einen Shaders den Ausgang eines anderen Shaders zuweist. Auf diese Weise können viele kleine Shader zu einem großen und mächtigen Konstrukt kombiniert werden das eine spezielle Aufgabe erfüllt.

Zum Beispiel kann man einem Phong-Shader einen Texture Shader zuweisen. Der Vorteil dabei ist, dass sich der Phong-Shader nicht selbst um die Texturierung kümmern muss, sondern einen fertigen Farbwert erhält und nur mehr die Helligkeit am der betreffenden Stelle berechnen muss. Solche Zuweisungskonstrukte können enorme Größen erreichen, wenn man damit gezielte Effekte umsetzen will. Schreibt man ein derartiges Konstrukt in einer .mi-Szene, so bezeichnet man dies als "Shadergraph".

Das Shaderassignment ist in der 3D-Welt weit verbreitet und nicht mehr wegzudenken. Die Namensgebungen variieren zwischen den Programmen. Selbst Softimage, das ja mit Mental Ray arbeitet, hat eine eigene Bezeichnung dafür eingeführt. Dort heissen diese Konstrukte nicht mehr „Shadergraph“ sondern „Rendertree“.

### A.9.2 Der Rendertree

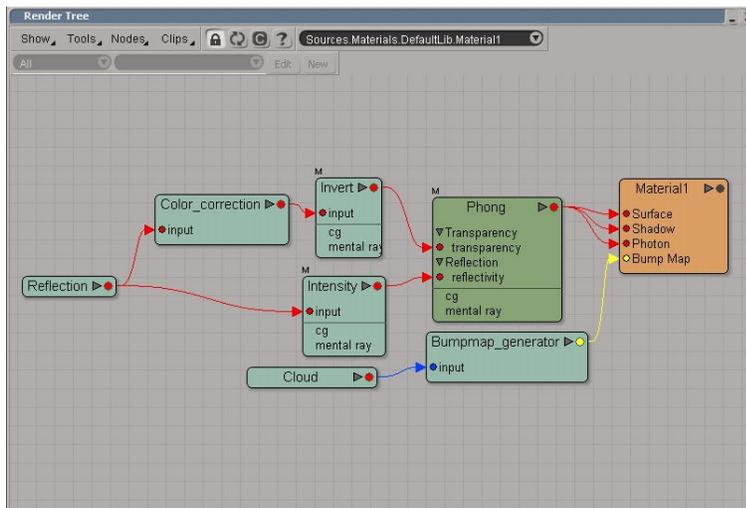
In Kapitel „**Klärung des Begriffe Shader**“ wurde der Begriff „Shader“, wie hier an dieser Stelle wiederholt definiert: „Ein Shader in Mental Ray ist ein Programmcode der in C/C++ geschrieben wurde. Dieser Programmcode schreibt Mental Ray vor, wie er eine Berechnung für einen bestimmten

Rendervorgang durchzuführen hat. „

Funktional unterscheiden sich Rendertrees und Shadergraphs nicht voneinander. Beide bieten dieselben Möglichkeiten und Funktionen. Einen Shadergraph in einer Textdatei nachzuverfolgen ist wesentlich mehr Aufwand, als dies in grafischer Form im Rendertree zu tun. Der Rendertree bietet eine umfassende grafische Benutzeroberfläche die den Prozess des Zusammensetzens wesentlich erleichtert.

Ein solcher Rendertree aus Softimage ist in Abbildung 22 zu sehen. Die Kästchen werden im Englischen als „Nodes“ bezeichnet und werden im Folgenden nach diesem Wort benannt.

Jeder Node, der in der Abbildung zu sehen ist, stellt mit Ausnahme des ganz rechten einen eigenständigen Shader dar. Das heisst, jeder Node repräsentiert einen Shader – und Shader repräsentieren Funktionen. Der Orange Node auf der rechten Seite stellt die Verbindung zum Material dar. In Mental Ray muss jedes geometrische Objekt zumindest einen Shader für „Surface“ definiert haben. Die weiteren Eingänge im Material-Node können optional belegt werden. Shader können für mehrere Shaderzwecke programmiert werden. Es gibt viele Shader, die man als Surface, Shadow und Photon Shader verwenden kann. Je nach Aufgabentyp, für welchen ein Shader programmiert wurde, steht in der SPDL-Datei an welche Materialeingänge man diesen Shader anhängen darf und an welche nicht (Siehe Kapitel B.5.2.1 auf Seite 88). Geometrische Objekte verarbeiten nicht alle Arten von Shadern sondern nur eine Untergruppe. Shader, die dort nichts verloren haben, wird es erst gar nicht ermöglicht an den Materialnode anzudocken. Daher bietet der Material-Node nur folgende Eingänge an:



- Surface
- Volume
- Shadow
- Environment
- Photon
- Photonvolume
- Contour
- Displacement
- Bump

**Abbildung 22:** Beispiel eines Rendertrees in Softimage

Aufgrund der lockeren Definition der Shaderschnittstelle ist es zwar möglich Shader falsch zu verwenden, doch solche Missbräuche enden meist mit einem Programmabsturz. Es gibt genug Sicherheitsmaßnahmen, die versehentlichem Missbrauch vorbeugen. Diese Sicherheitsmechanismen

können an verschiedenen Stellen angebracht werden. Entweder in der Shader deklarierenden SPDL-Datei oder direkt im C-Code des Shaders. Mehr dazu wird in TEIL 2 erläutert.

An die Eingänge im Material-Node können verschiedene Shader angeschlossen und kombiniert werden. Dass Shader nicht immer Farbwerte berechnen, wurde in Kapitel A.3 erläutert. Damit man immer weiß, welchen Ergebnistyp ein Shader produziert, wurde im Rendertree von Softimage ein Farbcode eingeführt.

<i>Farbe</i>	<i>Zahlentyp</i>
Rot	RGB(A)-Wert
Gelb	Vektor
Orange	Boolean
Hellgrün	Skalar
Dunkelgrün	Integer
Blau	Dateiquelle

Anhand dieses Farbcodes kann man rasch erkennen, welchen Typ die Eingänge des Shaders akzeptieren und von welchem Typ der Ausgang eines Shaders ist. Dies ist beim Zusammensetzen von Rendertrees sehr hilfreich. Ein RGBA-Wert besteht aus 4 Zahlenwerten. Weist man dem Skalareingang eines Shaders den Farbausgang eines anderen Shaders zu, weiß der Shader nicht, wie er diesen Wert umsetzen muss. Doch auch hierfür gibt es eine Lösung, sogenannte Type-Conversion Shader. Man einen derartigen Shader zwischenschalten um einen Zahlentyp in einen anderen umzuwandeln. Die Shader sind meist so programmiert, dass man auswählen kann, auf welche Art und Weise die Konvertierung stattfindet. So kann man sich aus 3 Skalarwerten einen RGB-Wert zusammensetzen lassen.

Solche Hilfsshader sind universelle Shader. Sie können das Ergebnis jedes beliebigen Shaders weiterverarbeiten und überall eingesetzt werden, wo man Werte einfach bearbeiten oder umwandeln möchte. Man kann damit sowohl das Ergebnis eines Lichtshaders verändern als auch mit Skalaren oder Vektoren rechnen. Der Einsatz ist breit gefächert. Daraus erkennt man, dass der Begriff Shader mit einer gewissen Freizügigkeit verwendet wird, da nicht sofort erkennbar ist welche Berechnungen tatsächlich dahinterstecken. Um jedoch die Bezeichnungen konsistent zu halten, werden Nodes unabhängig von ihrer Komplexität Shader genannt. Zudem dienen Shader dazu den Renderprozess maßgeblich zu beeinflussen. Selbst der einfachste Node (=Hilfsshader) kann für das Aussehen des gerenderten Bildes ausschlaggebend sein.

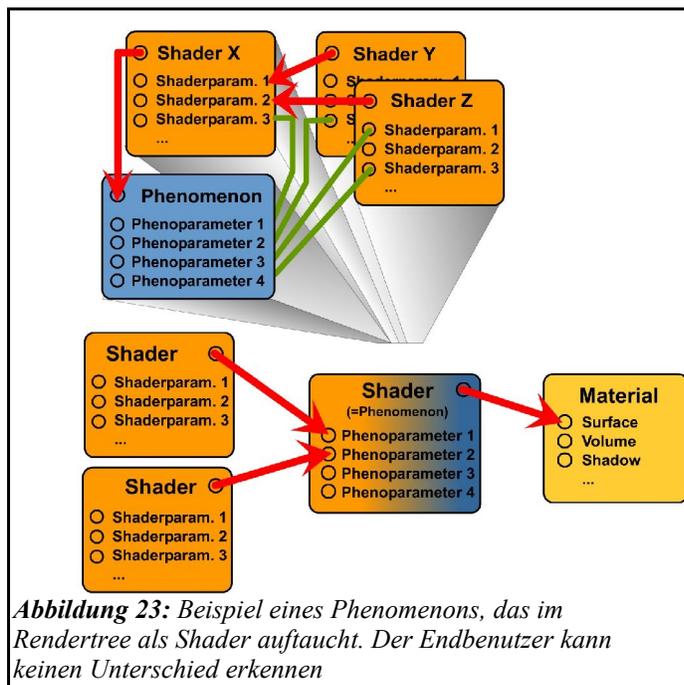
## A.9.3 Shadergraphs

In Softimage werden Rendertrees eingesetzt um mehrere Shader zu kombinieren. Übersetzt man eine Szene in eine .mi-Datei, nennt man diesen Rendertree dort Shadergraph. Es gibt keine funktionellen Unterschiede zwischen den Konstrukten. Ein Rendertree kann in der geschriebenen Variante mühelos nachverfolgt werden.

## A.9.4 Phenomena

Phenomena (sg. Phänomenon) sind eine Erweiterung der Shadergraphs. Phenomena sind nicht dazu gedacht Shadergraphs zu ersetzen sondern die Bedienung von großen Shadergraphs zu erleichtern. Phenomena werden in Softimage|XSI via SPDL's eingebunden und sind für den Benutzer völlig unsichtbar.

Ein Phänomenon ist ein Shadergraph, bei dem man nur auf ausgewählte Parameter zugreifen kann. Nach außen hin sieht ein Phänomenon exakt wie ein Shader aus, auch wenn dahinter wieder ein eigenständiger Shadergraph steckt.



Ein Phänomenon ist beinahe gleich aufgebaut wie ein Shadergraph mit dem einzigen Unterschied, dass ein Phänomenon sogenannte Interface Parameters bietet. Jeder einzelne Shader in einem Shadergraph besitzt eine bestimmte Anzahl an Parametern, die man für diesen Shader einstellen kann. Manche Werte brauchen für einen Shader nur einmal eingestellt und müssen danach nicht mehr adjustiert werden. Deswegen werden nur solche Parameter als Interfaceparameter definiert, deren Änderung Endergebnis in vernünftigen Grenzen variiert.

Das Phänomenon zeigt nach außen nur mehr diese Parameter und verbirgt seine innere tatsächliche Struktur. Ein Phänomenon ist im Rendertree von Softimage nicht als ein solches zu erkennen, sondern wird genauso als „Node“ dargestellt wie andere Shader. Es ist auch nach wie vor möglich, einem Parameter des Phänomenon-Interfaces einen Wert per Shader-Assignment zuzuweisen. Wohin der Wert im Phänomenon gelangt, hängt natürlich vom Aufbau

des Phänomenons ab [MRB, 2001, 223-239], wie man in der Abb. deutlich erkennen kann.

Zum Beispiel sind sämtliche Fehlermodelle von Softimage Phenomena (außer die "simple"-Versionen). Dies wird erst klar, wenn man sich eine Szene als .mi-File ausgeben lässt und die Datei genauer inspiziert. Sowohl der Aufbau von Phenomena als auch von Shadern wird in SPDL's beschrieben. Anhand dieser SPDL's kann Softimage den Phenomenon-Code für die .mi-Datei erzeugen.

Der Unterschied zu Shadergraphs ist der, dass nicht mehr alle Parameter sichtbar sind, sondern nur jene, die für das Aussehen des gerenderten Bildes relevant sind. In Shadergraphs oder Rendertrees hat man normalerweise Zugriff auf alle Parameter, was in der Regel besonders für unerfahrene Anwender verwirrend ist. Zumeist sind von diesen unzähligen Parametern nur wenige für sinnvolle Veränderungen relevant. Man braucht also nur mehr wenige Werte einzustellen um etwas an dem Material zu verändern. Also wird ein Interface mit ausgewählten Parametern definiert. Danach hat man nur mehr auf die Parameter des Interfaces Zugriff. Alle anderen Parameter die nicht im Phenomenon-Interface aufscheinen, werden mit den jeweils definierten Standardwerten versorgt.

Interface Parameters lassen ein Phenomenon nach außen hin wie einen einfachen Shader aussehen, auch wenn nach innen das Phenomenon aus einer komplizierten Struktur von verketteten Shadern besteht.

Es fehlt oft an Basiswissen über die Arbeitsweise von Mental Ray, da man hier indirekt mit dem State und allen seinen Variablen arbeitet. Viele Anwender wissen nicht wie Mental Ray im Hintergrund arbeitet was zur Folge hat, dass selbsterstellte Rendertrees manchmal ineffizient arbeiten.

Ein Phenomenon kann in Softimage|XSI folgendermaßen nachgestellt werden:

Nachdem man einen mehr oder weniger komplexen Rendertree zusammengesetzt hat, gibt es darin nur ein paar Parameter die zur maßgeblichen Änderung des Looks verwendet werden. Diese Parameter werden in einer Property-Page zusammengefasst:

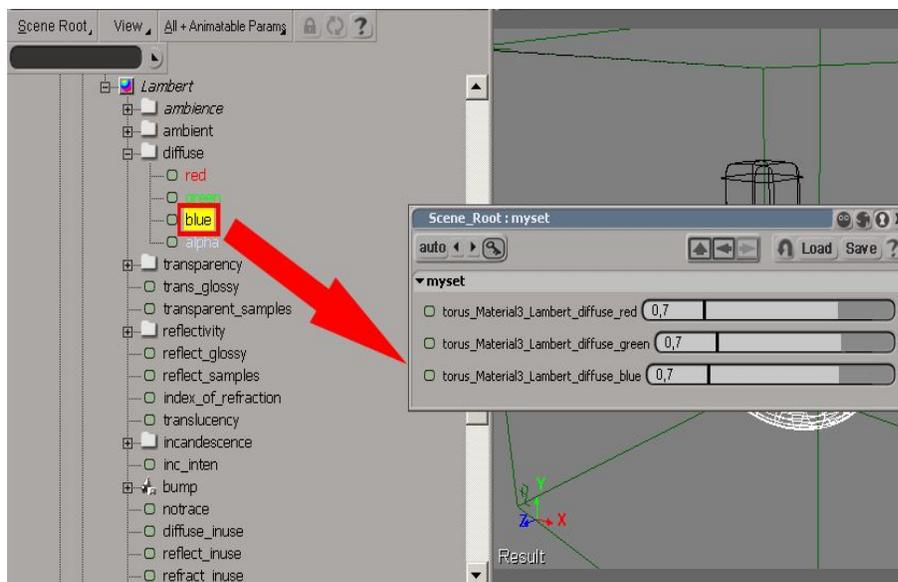
Man selektiert das Objekt, das mit dem entsprechenden Rendertree versehen wurde. Unter „Animate>Create>Parameter>New Custom Parameter Set“ kann man eine leere Property-Page (PPG) erstellen. Beim Aufruf des Kommandos kann man der Property-Page einen Namen geben, zum Beispiel

“Metall\_shader\_Params“. Die PPG wird gelockt (das Schlüsselloch-Symbol rechts oben), damit diese beim Öffnen einer anderen PPG bestehen bleibt.

Nun öffnet man der Reihe nach die PPGs der Shadernodes des Rendertrees und zieht per Drag and Drop die einzelnen Parameter, die man separiert bearbeiten möchte, in die Propertytype. Diese Parameter werden automatisch auf die entsprechenden Werte gelinkt.

Will man die Bezeichnung der Parameter ändern, kann man dies, indem man links des Parameters einen Rechtsklick auf dem grünen Button ausführt und „Edit Parameter Definition“ wählt. Dort stellt man Namen und Geltungsbereich des Reglers ein. Aber Achtung: Will man Farbwerte über eine Propertytype steuern, ist dieser Prozess wesentlich schwieriger. Farbreger lassen sich nämlich nicht in einer Propertytype unterbringen. Dafür gibt es einen Workaround:

Man expandiert im entsprechenden Objekt den Explorer, sucht sich aus dem Material den Farbparameter heraus und zieht diesen per Drag and Drop auf die Propertytype. Das Resultat sind drei Slider für die Farbwerte.



**Abbildung 24:** Zuweisen der RGB-Werte an eine Propertytype

Auf nebstehender Abbildung ist dieser Schritt dokumentiert. Hat man alle Parameter in einer Propertytype zusammengefasst, muss man nicht mehr zwischen den einzelnen Nodes im Rendertree hin- und herwechseln, um einen Parameter zu ändern. Diese Arbeitsweise kommt dem Verhalten eines

Phenomens sehr nahe, auch wenn es sich dabei um kein echtes Phänomen handelt.

## A.10 Unterschied zwischen "Texture" und "Material"

Die Begriffe *Material* und *Texture* sollten keinesfalls verwechselt werden. Die Aufgabe des Texture Shaders ist es, anhand von UV-Koordinaten auf der Oberfläche einer Geometrie einen entsprechenden Farbwert zurückzugeben. Dieser Farbwert kann entweder aus einer Bilddatei stammen oder von einem prozeduralen Algorithmus berechnet worden sein. Ein reiner Texture Shader ist für nichts anderes gedacht. Material Shader hingegen können zwar Texture Lookups eingebaut haben, zeichnen sich aber vor allem dadurch aus, dass sie weitere Rays in die Szene werfen können. Texture Shader hingegen liefern nur aufgrund der gegebenen UV(W) Koordinaten ein passendes Ergebnis, weshalb ein Texture Shader auch als Shadow Shader eingesetzt werden kann.

Texture Shader stellen nur die Funktionalität des Texturelookups zur Verfügung und erzeugen im Gegensatz zu Material Shader nie neue Rays.

## A.11 MI-File Aufbau

*Mental Ray hat eine eigene Szenenbeschreibungssprache. Dateien, die in der korrekten Mental Ray Syntax verfasst wurden, haben die Dateiendung ".mi". Mental Ray akzeptiert solche Dateien als Input und kann daraus ein Bild berechnen. Die wichtigsten Eigenschaften dieses Szenenbeschreibungsformats werden hier erläutert.*

Jedes 3D-Programm hat sein eigenes Szenenbeschreibungsformat. In diesem Format ist festgelegt, auf welche Art und Weise es geometrische Objekte darstellt, wie Shader definiert und Lichter verwaltet werden usw. Das heisst, dass das Programm seine interne Szenendarstellung zuerst für Mental Ray übersetzen muss bevor Mental Ray überhaupt zu arbeiten beginnen kann. Deswegen ist es oft auch keine einfache Sache einen Renderer in ein 3D-Programm zu integrieren. Softimage zum Beispiel arbeitet seit seinem ersten Erscheinen mit Mental Ray – die Software wurde so ausgelegt, dass die Szenen möglichst schnell und einfach für Mental Ray übersetzt werden können. Doch selbst das funktioniert nicht immer einwandfrei. Das ist auch der Grund, warum in Softimage bislang kein anderer Renderer zum Einsatz kam. Erst ab der Softimage Version 4.0 konnten neben dem Software Renderer Mental Ray auch Hardwarerenderings mittels handelsüblicher Grafikkarten hergestellt. Mittlerweile ist Softimage auf die Spieleentwicklungsschiene aufgesprungen und bietet die Möglichkeit, im Programm interaktiv Cg-Shader

zu schreiben und sofort zu testen. Cg ist eine Shadersprache für von Nvidia entwickelte Grafikkhardware. Im Gegensatz zu Mental Ray Shader werden die Cg-Shader nicht in C/C++ geschrieben, sondern können direkt in Softimage geschrieben und getestet werden. Weitere Information zu dieser Sprache kann unter [„http://developer.nvidia.com/page/cg\\_main.html“](http://developer.nvidia.com/page/cg_main.html)

gefunden werden. Hardwarerendering wird von Mental Ray in Ansätzen unterstützt. Der Einsatz wird sich im Laufe der Jahre noch weiter verstärken. Im Frühling 2004 kündigte Mental Ray an, verstärkt mit Nvidia zu kooperieren. Gemeinsam mit der neuen QuadroFX 4000, einer High-End Grafikkarte von Nvidia, soll es jetzt möglich, sein Softwarerendering durch Hardware weiter zu beschleunigen und Berechnungen dank der höheren Rechengenauigkeit der neuen Grafikkarten von diesen erledigen zu lassen. Man darf gespannt auf diese Entwicklung sein.

In dieser Arbeit wird nicht auf die Cg-Unterstützung in Softimage eingegangen, da sich diese Arbeit auf die Integration von Mental Ray beschränkt.

Die ganze .mi-Szenenbeschreibungssyntax zu beschreiben ist zu umfangreich. Es soll gezeigt werden, welche Informationen Mental Ray verarbeitet und in welcher Form die Szeneninformation aufbereitet sein muss. Softimage|XSI ist dafür verantwortlich, dass die Szene für Mental Ray richtig aufbereitet wird. Das Interessante ist, dass Mental Ray fließend in Softimage eingebaut ist und über ein Integration Application Interface gesteuert wird. Es gibt kein .MI-Format in das die Szene zuerst übersetzt wird, obwohl Softimage .MI-Dateien erzeugen und abspeichern kann. Das ist auch der Grund weshalb nicht alle Mental Ray Features von Softimage |XSI aus zu gesteuert werden können. Das Integration API wird ständig erweitert und verbessert, aber ein paar Dinge wurden verabsäumt in das XSI-Interface integriert zu werden. Das sind zwar keine essentiellen Features, aber doch Funktionalitäten, die versierte Benutzer gerne implementiert haben würden. Schließlich geht es darum, dass Mental Ray in seiner vollen Leistungsfähigkeit unterstützt wird. Ein Grund dafür ist, dass Mental Ray wie jede andere gute Software ständig erweitert und verbessert wird. Kommt eine neue Version von Mental Ray heraus, muss nun das Softimage|XSI-Interface entsprechend an die neuen Funktionen angepasst werden. Daher wird nach dem Erscheinen einer neuen Mental Ray Version meist ein Update von Softimage|XSI angeboten welches die neuen Features unterstützt.

Das Format, mit dem Mental Ray umgehen kann, sind .mi-Dateien. Eine .mi Datei besteht aus einem Set von Elementen, die in mannigfaltiger Weise verschachtelt werden. Diese Elemente heißen Top-level Elemente. Ein Top-level Element wird mit seinem entsprechenden Schlüsselwort deklariert, gefolgt von einigen Angaben, die das Element genauer beschreiben. Aus der Kombination dieser Top-level Elemente kann jede nur denkbare Szene zusammengestellt werden. Diese Elemente sind [MRB, 2001, s. 30f]:

<code>options</code>	Die Renderoptionen bestimmen wie Mental Ray arbeitet. Darin werden wichtige grundlegende Einstellungen getroffen, welche die Geschwindigkeit beim Rendern und das resultierende Bild maßgeblich beeinflussen. Options können von außen, also von der Kommandozeile überschrieben werden. Das heisst, wenn in der Kommandozeile eine Funktionalität aktiviert wurde, die in den Options in der <code>.mi</code> -Datei deaktiviert wurde, so gilt die Definition der Kommandozeile.
<code>camera</code>	Definiert eine Kamera, von deren Perspektive aus die Szene gerendert wird.
<code>light</code>	Definiert eine Lichtquelle
<code>object</code>	Definiert ein geometrisches Objekt
<code>instance</code>	Eine Instanz platziert ein Element in der Szene. Dies kann wahlweise ein Element vom Typ <code>light</code> , <code>object</code> oder <code>camera</code> sein.
<code>instgroup</code>	Bündelt mehrere Instanzen in einer Gruppe. Instanzgruppen können selbst wiederum instantiiert werden. Alle Instanzen und Instanzgruppen müssen in einer Instanzgruppe, der <code>rootgroup</code> enthalten sein. Mit dem <code>render</code> -Befehl gibt man an, welche Instanzgruppe (die selbst wieder aus anderen Instanzen und Instanzgruppen bestehen kann) mit welcher Kamera und welchen Renderoptionen die Szene gerendert wird.
<code>declare</code>	Bevor ein Shader verwendet werden kann, muss er deklariert werden. Der Shadername muss dabei gleich lauten wie die C/C++-Funktion, die diesen beschreibt. In der Deklaration steht, wie die Parameter heissen und von welchem Typ (Color, Vector, ect...) sie sind.
<code>shader</code>	Spezifiziert einen Shader genau. Die Parameter werden mit Werten versehen und der Shader erhält einen eindeutigen Namen. Mit diesem Namen kann man den Shader mit seinen Einstellungen jederzeit in der Szene einsetzen.
<code>material</code>	Dieses Element fasst mehrere Shader zusammen. Es definiert für geometrische Objekte auf jeden Fall seinen Surface Shader und optional dazu Displacement-, Shadow-, Volume-, Environment-, Contour-, Photon- und Photon Volume-Shader. Ein Material wird

danach einem Objekt oder einer Instanz angehängt. Dieses Element entspricht dem Material Node aus dem XSI-Rendertree. Einzig der Bumpmap-Eingang ist eine Spezialität des Rendertrees in Softimage, da Bumpmapping kein eigener Shader Type ist, sondern lediglich die Orientierung der Oberflächennormalen verändert. Softimage lässt den User allerdings glauben, dass dies ein eigener Shadereingang sei, was nicht der Fall ist.

`phenomenon` Definiert ein Phänomen. Wie schon gezeigt wurde, bündelt ein Phänomen eine Menge von Shader, die zu einem Shadergraph zusammengebaut wurden. Man hat nur auf ausgewählte „Interface Parameter“ Zugriff, der Rest ist nach außen hin unsichtbar. In einem Phänomen-Konstrukt kann auch angegeben werden, welche Art von Shader vorliegt (Light, Shadow, Displacement...). Weiters sind Phänomene praktisch, weil man bestimmen kann, dass sie vor dem Rendern der Szene an anderen Top-level-Elementen Shader anbringen. [MRB, 2001, s. 238f]

`texture` Ein Texture-Element verweist auf eine Textur auf der Festplatte. Auf diese Textur wird mittels einer Pfadangabe verlinkt.

Ein Top-level Element wird erzeugt indem man die Bezeichnung des Elements tippt, gefolgt von einem frei wählbaren Identifikationsnamen der unter Anführungszeichen angeführt wird.

Die Syntax mit der Elemente definiert werden ist einfach. Top-level Elemente werden durch ihr Schlüsselwort, gefolgt von einem frei wählbaren Identifikationsnamen (der am besten der C-Syntax entspricht) deklariert. Anhand dieses Identifikationsnamens können diese Elemente dann in weiteren Top-Level Elementen verwendet werden.

```

declare shader
    color "lumaPhongRbx2" (
        ...)
end declare

shader "Luma_defined1" "LumaPhongRbx2" (
    ...)

declare shader
    color "rbx_displace" (
        ...)
end declare

shader "displaced_sh" "rbx_displace" (
    ...)

material "myLumaMat"
    = "Luma_defined1"
    displace = "displaced_sh"
end material

```

*Abbildung 25: Ein Material in einer .mi wird aus Shadern zusammengestellt, die zuvor deklariert und definiert wurden. Die Namen im Declare-Statement müssen dem Funktionsnamen in der C/C++-Datei entsprechen. Der Name bei der Definition darf frei gewählt werden, soll aber der C-Syntax entsprechen.*

Hier ist das Ganze farblich genau veranschaulicht. Im Declare-Statement werden die Shader erst mit ihrem Namen deklariert. Dieser Name *muss* mit dem Funktionsnamen wie er im Shadercode lautet übereinstimmen. Wenn also der Funktionsname der C-Funktion `lumaPhongRbx2` lautet, muss der Shader mit `declare shader color "lumaPhongRbx2"` deklariert werden. Zwischen den runden Klammern werden die einstellbaren Parameter des Shaders deklariert. Diese Definition wurde in der seitlichen Abbildung weggelassen und durch drei Punkte ersetzt.

Die Shaderdeklaration ist eine einmalige Deklaration und deshalb wird sie auch normalerweise in einer separaten .mi-Datei

gespeichert. Diese Datei wird per `$include`-Befehl in die eigentliche Szenendatei eingebunden.

Das Shaderstatement sieht so aus: Das erste Wort unter Anführungszeichen ist der neue Name für den definierten Shader. Das zweite Wort gibt an, welchen deklarierten Shader man mit Werten versieht. Schlussendlich werden die Shader in einem Materialstatement zusammengefasst. Verpflichtend ist die Angabe eines Surface-Shaders für die Materialoberfläche. Alle anderen Angaben von Shader sind optional. In unserem Fall wurde an das Material auch noch ein Displacement Shader angefügt.

Man kann Materialien an mehreren Stellen anbringen. Eine Möglichkeit ist das Material in einer Objektdefinition unterzubringen. Eine andere Möglichkeit ist, zugleich mit dem `instance`-Befehl für ein Objekt das Material anzugeben. Der Grund, warum es mehrere Möglichkeiten gibt, liegt in der Funktionalität der Materialvererbung [MRB, 2001, s. 307-315]: Wenn in einer Objektdefinition kein Material angegeben wurde, so erben Objekte die Materialien, die bei der Instantiierung (`instance`, `instgroup`) angegeben wurden.

Es folgt nun der Inhalt einer .mi-Datei. In der Struktur findet man sich schnell zurecht und man erkennt rasch, dass eine Szene aus gegliederten Elementen, den Top-Level Elementen besteht. Eine Szene wird ausschließlich aus diesen Komponenten zusammengestellt. Elemente wie Licht, Shader, geometrische Objekte, Phenomena, Kameras, Optionen werden erst definiert. Definieren bedeutet nichts anderes die Objekte mit Werten zu versehen.

*Definition:* Ein Shader wird deklariert, indem er mit seinem Funktionsnamen des C-Codes angeführt wird und man seinen Rückgabedatentyp sowie die Datentypen und Namen seiner Interfaceparameter bekannt macht.

*Definition:* Phenomena müssen wie Shader ebenfalls erst deklariert werden. Ihnen liegt meist ein Shadergraph zugrunde. Ein Phänomenon definiert ebenfalls wie ein Shader Interfaceparameter. Diese Interfaceparameter sind jedoch ausgewählte Parameter einzelner Shader des zugrunde liegenden Shadergraphs. Man kann von außen nur diese definierten Parameter steuern. Parameter, die zwar im Phänomenon zu finden sind, jedoch nicht als Interface-Parameter öffentlich zugänglich gemacht wurden, bleiben dem User verborgen. Nach außen hin verhält sich ein Phänomenon gleich wie ein Shader.

*Definition:* Definieren eines Elements in einer .mi Datei bedeutet, für alle Parameter, die das Element beschreiben, Werte (Zahlen, Strings,...) bereitzustellen. Ein definiertes Element ist noch nicht aktiv an der Szene beteiligt, sondern muss erst in der Szene verwendet werden.

Diese Shader müssen in Material-Statements untergebracht werden. Materials definieren, welche Shader für welche Shaderzwecke verwendet werden. Verpflichtend ist wie oben schon erwähnt, dass ein Shader für die Oberfläche der Geometrien angegeben wird, alle anderen Eingänge können optional mit anderen Shader oder eben Phenomena belegt werden.

Danach können diese Elemente instantiiert oder in einem Kommando aufgerufen werden. In .mi-Files ist es erlaubt, mehrere `option`-Blöcke zu definieren. Das liegt daran, dass mit dem `render`-Statement derjenige Optionsblock angegeben wird, dessen Einstellungen gelten sollen.

Nach der Objektedefinition, werden diese instantiiert und für den Renderer in der Szene sichtbar gemacht. Die Definition allein ist nicht ausreichend. Dort wird lediglich das Objekt mit seinen grundlegenden Eigenschaften beschrieben, aber noch nicht als aktives Objekt in die Szene integriert. Erst nach der Deklaration ist es möglich eine Instanz daraus zu erstellen.

Dies wird im Folgenden genauer untersucht:

Hier liegt eine .mi-Datei im ASCII-Format vor [MRSMP, 2005, scn\_example.mi]. In dieser Datei sind alle Information enthalten, die Mental Ray benötigt um ein Bild zu erzeugen. Alle 3D-Programme die an Mental Ray angebunden sind, müssen Mental Ray mit diesem Format versorgen.

```
#-----  
# Copyright 1986-2000 by mental images GmbH & Co.KG, Fasanenstr. 81, D-10623  
# Berlin, Germany. All rights reserved.  
#-----  
#  
# Chapter 2.9      : Scene Example  
# Description     : Example scene creates two different images of a cube,  
#                  each of which with a different camera and light  
#  
#-----  
  
verbose on  
link "base.so"  
$include <base.mi>  
  
options "opt"  
  samples      -1 1  
  contrast     .1 .1 .1 .1  
  trace depth  2 2  
end options  
  
camera "cam1"  
  frame        1  
  output       "rgb" "x.rgb"  
  focal        100  
  aperture     144.724029  
  aspect       1.179245  
  resolution   500 424  
end camera  
  
instance "caminst1" "cam1" end instance  
  
light "light1"  
  "mib_light_point" (  
    "color" 1 1 1,  
    "shadow" on,  
    "factor" 1  
  )  
  origin      141.375732 83.116005 35.619434  
end light  
  
instance "lightinst1" "light1" end instance  
  
declare shader  
color "mib_illum_phong" (  
  color      "ambience",  
  color      "ambient",  
  color      "diffuse",  
  color      "specular",
```

```

        scalar          "exponent", # phong exponent
        integer         "mode",     # light selection mode 0..2
        array light     "lights"
    )
    version 2
    end declare

    shader "definedPhong"
        "mib_illum_phong" (
            "ambience" .3 .3 .3,
            "ambient"   .5 .5 .5,
            "diffuse"   .7 .7 .7,
            "specular"  1 1 1,
            "exponent"  50,
            "lights"    [ "lightinst1" ]
        )

    material "mtl" opaque
        = "definedPhong"

    end material

    object "obj1"
        visible shadow trace
        group "mesh"
            -7.068787  -4.155799  -22.885710
            -0.179573  -7.973234  -16.724060
            -7.068787  4.344949   -17.619093
            -0.179573  0.527515   -11.457443
            0.179573   -0.527514   -28.742058
            7.068787   -4.344948   -22.580408
            0.179573   7.973235    -23.475441
            7.068787   4.155800    -17.313791

            v 0  v 1  v 2  v 3  v 4  v 5  v 6  v 7

            c "mtl"  0 1 3 2
            c       1 5 7 3
            c       5 4 6 7
            c       4 0 2 6
            c       4 5 1 0
            c       2 3 7 6
        end group
    end object

    instance "inst1" "obj1" end instance

    instgroup "world"
        "caminst1" "lightinst1" "inst1"
    end instgroup

    render "world" "caminst1" "opt" # render frame 1

```

Diese Datei ist keine von Softimage erzeugte Datei. Softimage|XSI würde sämtliche existierenden Parameter des Optionblocks ausfüllen um vorzubeugen, dass Mental Ray eigene Standardwerte verwendet. Das soll weiter nicht stören. Zuerst werden mit „verbose on“ sämtliche Mental Ray Statusmeldungen aktiviert, die das Programm während des Renderns in der Kommandozeile erzeugt. In den Statusmeldungen sind Informationen über die jeweiligen Arbeitsschritte enthalten. Dadurch kann man beim Abbruch des Renderers durch einen Fehler erkennen, an welcher Stelle und mit welcher Fehlermeldung Mental Ray den Rendervorgang abgebrochen hat.

Mit dem `link`-Kommando wird auf benötigte `.dll`- bzw `.so`-Dateien verwiesen, die Mental Ray zum Rendern benötigt. In der Datei „`base.dll`“ ist der Kompilierte Shadercode für alle Basissshader die mit Mental Ray geliefert werden, enthalten.

Die Includeanweisung `$include <base.mi>` weist Mental Ray an, den Inhalt dieser Datei ebenfalls zu laden. Es reicht nämlich nicht aus, lediglich auf die `.dll`-Datei zu verweisen, sondern man muss auch die Shader mit all ihren Parameternamen deklarieren. Diese Deklarationen stehen in der Datei `base.mi`. Mit weiteren Includeanweisungen ist es möglich zusätzliche `.mi`-Dateien zu laden. Zum Beispiel können in anderen `.mi`-Dateien schon Geometrien deklariert worden sein, die im Folgenden nur mehr instantiiert werden müssen.

```
camera "cam1"
  frame      1
  output     "rgb" "x.rgb"
  focal      100
  aperture   144.724029
  aspect     1.179245
  resolution 500 424
  clip       0.100000 32768.000000
end camera

instance "caminst1" "cam1" end instance
```

Jedes Top-Level Element hat bestimmte Parameter, die für dieses Element eingestellt werden müssen. Manche Parameter sind mit Werten zu versehen, andere Angaben sind optional. Werden Angaben weggelassen, nimmt Mental Ray bestimmte Standardwerte für diese Parameter an. Hier wurde die Kamera definiert und jeder ihrer relevanten Parameter mit Werten versehen.

- `frame`: Gibt an das wievielte Bild einer Animation die `.mi` -Datei beschreibt.
- `output`: Definiert welches Bildformat erzeugt wird. Mental Ray unterstützt eine breite Palette von Formaten.
- `focal`: Abstand der Kamera von dem Kamera-Ursprung zur Viewing-Plane. Wird dieser Wert mit „infinity“ versorgt, bedeutet dies, dass die Kamera unendlich weit vom Objekt entfernt ist und keine perspektivische Verzerrung stattfindet. Dies bezeichnet man als orthografische Kamera.
- `aperture`: Gibt die Breite der Viewing-Plane in Kamerakoordinaten an.
- `aspect`: Beschreibt das Seitenverhältnis der Bildbreite zur Bildhöhe. Das Verhältnis lautet standardmäßig 4:3. Stimmt dieser Wert nicht mit dem Verhältnis von Breite und Höhe die in „resolution“ angegeben wurden überein, bedeutet das, dass die Pixel nichtquadratisch sind.

`resolution`: Die Auflösung des zu rendernden Bildes

`clip`: Abstand der Clippingplanes von der Kamera. Alles was sich außerhalb der Clippingplanes befindet, also davor oder dahinter, wird nicht mitgerendert.

Hier wird der Unterschied zu Softimage|XSI deutlich. Dort wird das Bildformat und der Dateiname mit seinem Pfad in den Renderoptionen eingestellt. Aber in Wirklichkeit wird diese Information in die Scene-Entity der Kamera geschrieben. Das macht das Übersetzen einer Szenenbeschreibung für verschiedene Renderer kompliziert, denn ein anderer Renderer deklariert diese Angaben an anderer Stelle oder benötigt Angaben, die ein anderer Renderer nicht benötigt.

In ein Kamerastatement können auch Shader integriert werden. Mögliche Shader sind Output-, Lens-, Volume- und Environment-Shader. In Softimage|XSI werden Lens Shader im Rendertree der Kamera eingestellt. Output-, Volume- und Environment Shader werden in den Pass-Renderoptions definiert. Hinter den Kulissen sind sie aber alle Teil des Kamerastatements.

Während das `camera`-Statement die Kamera lediglich beschreibt, muss die Kamera für die Szene auch noch sichtbar gemacht werden. Dies geschieht, indem man die Kamera instantiiert, was im Beispielcode unmittelbar nach der Definition der Kamera mit `instance "caminst1" "cam1" end instance` instantiiert.

Im Folgenden wird eine Lichtquelle definiert und instantiiert:

```
light "light1"
  "mib_light_point" (
    "color" 1 1 1,
    "shadow" on,
    "factor" 1
  )
  origin      141.375732 83.116005 35.619434
end light

instance "lightinst1" "light1" end instance
```

Durch die Zeile `origin 141.375732 83.116005 35.619434` wird der Ursprung der Lichtquelle im Raum verändert. Eine zweite Möglichkeit das zu tun ist, zum Zeitpunkt der Instantiierung eine `Transform-Matrix` anzugeben, was beispielsweise so aussehen würde:

```

instance "lightinst1" "light1"
  transform 0 0 0 0
            0 0 0 0
            0 0 0 0
            141 83 35 1
end instance

```

Eine Transformationsmatrix besteht aus 4 Mal 4 Werten – Die 3x3 Werte oben links beschreiben die Rotation des Objektes im Raum. Der jeweils letzte Werte der ersten drei Zeilen repräsentiert die Skalierung in x, y und z. Die drei ersten Werte in der vierten Zeile repräsentieren die Translation entlang der x-, y- und z-Achsen. Der letzte Wert in der vierten Zeile lautet immer "1". Transformationsmatrizen können mehrfach hintereinander angewandt werden: Werden Instancegroups innereinander verschachtelt, so kann jede Instancegroup mit einer Transformationsmatrix frei im Raum gedreht werden. Alle instantiierten Objekte in der Gruppe werden mittransformiert. Wurden diese Objekte schon ebenfalls durch eine Matrix transformiert, akkumulieren sich die Transformationen.

```

declare shader
color "mib_illum_phong" (
  color      "ambience",
  color      "ambient",
  color      "diffuse",
  color      "specular",
  scalar     "exponent", # phong exponent
  integer    "mode",     # light selection mode 0..2
  array light "lights"
)
version 2
end declare

material "mtl" opaque
  "mib_illum_phong" (
    "ambience" .3 .3 .3,
    "ambient"   .5 .5 .5,
    "diffuse"   .7 .7 .7,
    "specular"  1 1 1,
    "exponent"  50,
    "lights"    [ "lightinst1" ]
  )
end material

```

Das Declare-Statement wurde aus der base.mi-Datei aus Übersichtlichkeitsgründen in die MI-Datei eingefügt. Normalerweise werden Shader in einer externen Datei deklariert und mit dem \$include-Befehl wie er auch am Kopf dieses Beispiels zu sehen ist, eingebunden. In Softimage sind im Verzeichnis "\Application\phenolib\include" einige MI-Dateien mit shaderdeklarationen zu finden.

## A.12 Mental Ray und Softimage

XSI verwendet eine eigene Integrations-API für Mental Ray um die Szene für Mental Ray zusammenzubauen. Danach kann XSI Mental Ray auffordern die Szene nach .MI zu exportieren, aber selbst ist Softimage|XSI nicht fähig .mi – Dateien zu lesen geschweige denn zu verstehen. Das mag paradox klingen, doch wurde Mental Ray auf diese Weise implementiert. Die gesamte Interaktion zwischen XSI und Mental Ray baut auf dieser Integrations-API auf. Interessierte Leser können sich im Netz nach der Datei "mirelay.h" auf die Suche machen. In dieser Headerdatei ist die ganze Integration beschrieben. Sie wird an und für sich nicht dem einzelnen User zugänglich gemacht. Eine Anfrage direkt an Softimage mag da mehr Erfolg bringen.

Es gibt auch eine Standalone Version des Mental Ray Renderers die das Rendern von .MI – Dateien ermöglicht.

## A.13 SPDL (Software Package Delivery Library)

*SPDLs sind ASCII-Dateien die den Aufbau von Propertytypes für Objekte, Operatoren, Plugins oder Shader beschreiben. Jeder Operator und jedes geometrische Objekt hat eine eigene SPDL. SPDLs definieren nicht die Implementation sondern definieren die Parameter die in der Propertytype eingestellt werden können. In der SPDL-Datei ist auch das Layout der Propertytype bestimmt, sowie interne Logik, also welchen Einfluss das die Änderung eines Parameters auf andere hat. Die SPDLs die in dieser Arbeit zum Einsatz kommen werden automatisch vom Shader Wizard erzeugt. Dieses Kapitel ist daher nur eine Ergänzung zu dem Thema SPDL-Dateien und stellt keine komplette Auflistung aller Features dar. Der volle Funktionsumfang von SPDLs kann in der Dokumentation von Softimage|XSI eingesehen werden.*

Jede Propertytype in Softimage|XSI wird durch eine Propertytype definiert. Einzige Ausnahme ist, wenn Propertytypes über das C++-API generiert werden, was aber für Propertytypes für Shader nicht gilt. Shader werden durch sich vom herkömmlichen SPDL-Layout leicht unterscheidende SPDLs beschrieben. Der Aufbau ist einfach und rasch zu verstehen.

## A.13.1 Der SPDL-Kopf

Den Kopf einer SPDL bilden diese 3 Zeilen:

```
SPDL
Version = "2.0.0.2";
Reference = "{guid}"
```

Ein GUID ist ein "globally unique identifier" der einzigartig ist. Er wird mittels eines Speziellen Algorithmus' erzeugt, der aus der Uhrzeit und Netzwerkadresse die GUID generiert. Dies gewährleistet, dass GUIDs einzigartig sind. Die Mit der Visual C Entwicklungsumgebung kommen zwei Tools, die GUIDs erzeugen können. `Guidgen.exe` und `uuidgen.exe`. Erstere ist die Kommandozeilenversion. Brauch man eine größere List von GUIDs kann man diese mit folgendem Aufruf in eine Datei schreiben lassen:

```
uuidgen -n10 > myguids.txt
```

## A.13.2 Der PropertySet-Teil

Danach werden die Ausgangs- und Eingangsparameter des Shaders genau beschrieben. Dies geschieht im PropertySet-Teil. Deklaration eines Propertysets lautet

```
PropertySet "<shaderparameter_Bezeichnung>"
{
    Parameter "bezeichnung" ...
    {
        ...
    }
}
```

eingeleitet. Die `shaderparameter_Bezeichnung` ist der Name der Struktur in der Headerdatei welche die Shaderparameter beschreibt. Der Name muss exakt gleich lauten, sonst können die Parameter des Shaders später nicht angesprochen werden. Die einzelnen Parameter werden in dem Parameterblock deklariert. Darin wird definiert um welchen Typ es sich handelt, ob es Input- oder Outputparameter sind und einiges mehr. Die Definition für einen Parameterblock lautet wie folgt.

```
Parameter "property_name" input | output
{
```

```

guid = "guid" ;
title = "label_text" ;
flags = 0|1;
writable = on|off ;
readable = on|off ;
animatable = on|off ;
texturable = on|off ;
persistable = on|off ;
inspectable = on|off ;
type = boolean | color | filename | integer | lens
      | light | material | matrix | model | scalar | shader
      | string | texture | texturespace | vector | reference
      | rtrendercontext;
type = array { Parameter {...} } ;
type = struct { Parameter {...} ... } ;
value = value ;
range = min_value to max_value
ui "keyword" = "combo" | "rgb" | "rgba" |
              "LightList" | "ImageBrowser";
}

```

input | output gibt an ob es sich um einen Eingabeparameter oder einen Ausgabeparameter handelt.

guid Ein Shader hat nur einen Ausgabeparameter, der je nach Ausgabebetyp eine vorgeschriebene GUID [vgl. SPDLREF] aufweisen muss. Die GUID für die möglichen output Parameter sind:

```

ColorOutputGUID    = "4C6879FF-7EC8-11D0-8E3B-00A0C90640EC";
VectorOutputGUID   = "1D58FA86-A96A-11D1-90DA-0000F804EB21";
ScalarOutputGUID   = "1D58FA87-A96A-11D1-90DA-0000F804EB21";
IntegerOutputGUID  = "1D58FA88-A96A-11D1-90DA-0000F804EB21";
MatrixOutputGUID   = "1D58FA89-A96A-11D1-90DA-0000F804EB21";
BooleanOutputGUID  = "1D58FA8D-A96A-11D1-90DA-0000F804EB21";

```

Alle anderen Parameter müssen eine zufällig generierte GUID besitzen.

Title ist eine kurze Beschreibung des Parameters und kann frei gewählt werden

flags 0|1 gibt an, ob der Parameter incremental Rendering unterstützt. Der Wert kann 0 oder 1 lauten. Incremental Rendering sollte generell deaktiviert werden, weil es ohnehin nur für Previewzwecke in der Render Region von Softimage verwendet wird.

writable = on|off Definiert ob der Parameter von außerhalb verändert werden kann, beispielsweise durch einen Script. Wird die Angabe weggelassen wird als

Defaultwert ON angenommen.

`readable = on|off` Definiert ob der Parameter ausgelesen werden kann. Wird die Angabe weggelassen wird als Defaultwert ON angenommen.

`animatable = on|off` Definiert ob der Wert animiert werden darf. Wird die Angabe weggelassen wird als Defaultwert ON angenommen.

`texturable = on|off` Definiert ob dem Parameter andere Shader zugewiesen werden können. Wird die Angabe weggelassen wird als Defaultwert OFF angenommen.

`persistable = on|off` Definiert ob der veränderte Parameterwert gespeichert werden soll. Wird die Angabe weggelassen wird als Defaultwert ON angenommen.

`inspectable = on|off` Ob der Parameter auf der Property page angezeigt wird oder nicht. Wird die Angabe weggelassen wird als Defaultwert ON angenommen.

```
type = boolean | color | filename | integer | lens  
      | light | material | matrix | model | scalar | shader  
      | string | texture | texturespace | vector | reference  
      | rtrendercontext
```

Gibt den Typ des Parameters an. Diese Arbeit beschränkt sich auf die einfachen Datentypen da die Erläuterung aller möglichen Datentypen und die Beschreibung wie und wann sie eingesetzt werden zu komplex ist.

`type = array,`

`type = struct`

Mit diesem Statement lassen sich Arrays oder Strukturen (structs) von Parametern erstellen. Die dazugehörige Syntax dafür lautet

<pre> type = array {     Parameter "element" input     {         ...     } }; </pre>	<pre> type = struct {     Parameter "member1" input     {         ...     }     ...     Parameter "memberN" input     {         ...     } }; </pre>
--	---

`value = value`      Definiert einen Standardwert Wert für den Parameter. Je nach Parametertyp muss die Richtige angabe gemacht werden.

`range = min_value to max_value`  
ist nur für die Datentypen `integer` und `scalar` sinnvoll. Man kann eine ober- und eine Untergrenze für die Werte die nicht überschritten werden können.

`ui "keyword" = "combo" | "rgb" | "rgba" | "LightList" | "ImageBrowser"`  
spezifiziert die Repräsentation des Parameters. Combo steht für verschiedene Darstellungen die in der "Defaults"-Sektion näher spezifiziert werden. Die Syntax dazu lautet:

```

value = 0; # Make "Item 1" the default.
ui "control" = "combo";
ui "enum" = "Item 1", 0, "Item 2", 1, "Item 3", 2;

```

Eine `LightList` ermöglicht es dem User in Softimage aus den vorhandenen Lichtern auszuwählen und diese in die Liste einzufügen. Die Syntax dazu lautet

```

ui "control" = "LightList";

```

Mit `"rgb"` oder `"rgba"` werden die entsprechenden Farbslider dargestellt:

```

ui "control" = "rgb"; oder ui "control" = "rgba";

```

Ein Imagebrowser lässt einen Dateibrowser erscheinen mit dem man eine Textur auswählen kann.

```
ui "control" = "ImageBrowser";
```

### A.13.3 Der Phenomenon-Teil

Nachdem auf diese Weise alle Parameter deklariert wurden, ist der PropertySet-Teil abgeschlossen.

Danach kommt ein kurzer Teil, in dem man ein Phenomenon zusammenbauen könnte. Für einen normalen Shader reicht allerdings folgende Angabe aus:

```
phenomenon "lumaReflect_declare"  
{  
    Name = "lumaReflect";  
    Version = 1;  
    Use = texture;  
}
```

Der Name bezeichnet den Namen der Funktion, die der Entry-Point in der DLL-Datei ist. Doch das soll weiter nicht kümmern. Diejenigen die eine SPDL ohne Shader Wizard erstellen müssen nur wissen, dass dieser Name gleich lauten muss wie die Funktion welche den Shadercode in der .cpp-Datei beinhaltet.

### A.13.4 Der Defaults-Teil

Der Defaults-Teil spezifiziert die UI-Attribute für die Eigenschaften die in der PropertySet-Sektion getroffen wurden. Wie auch der PropertySet-Teil wird der Defaults-Teil wie folgt deklariert:

```
Defaults  
{  
    parameter_name  
    {  
        Name = "name";  
        Description = "description";  
        UIRange = min to max [by step];  
        UIType = "type";  
        {  
            type_attribute = value;  
        }  
        Items  
        Items  
        {  
            "name1" = value1  
            "name2" = value2  
            ...  
        }  
    }  
}
```

```
        Commands guides
    }
    // ... Defaults für die weiteren Parameter ...
}
```

`parameter_name` muss derselbe Name sein mit dem der Parameter im PropertySet-Teil deklariert wurde.

`Name` ist die Bezeichnung, mit der der Parameter auf der Propertypage aufscheint.

`Description` scheint nicht auf und dient nur der Verständlichkeit. Sie kann auch weggelassen werden.

`UIRange` gibt den Wertebereich eines Integer oder Skalar-Sliders an. Dieser Wert braucht nicht mit dem der Deklaration im PropertySet übereinstimmen. Das bedeutet, das zum Beispiel der Wertebereich auf einen vernünftigen Rahmen begrenzt wird. Wenn User doch einmal einen größeren oder kleineren Wert als man mit dem Slider Einstellen eingeben möchten können dies, soweit es die Definition im PropertySet zulässt, tun.

`UIType` ist noch einmal die genaue Angabe des UI-Typs wie er auf der PPG aufscheint. Dieser kann Bool, BitmapWidget, Check, Combo, Iconlist, ImageBrowser, Lightlist, Number, Radio, ReferenceWidget, RGB, RGBA, static oder String sein.

Die jeweils zulässigen Values sind:

**Bool:** Ein True / False Radiobutton

**Bitmap Widget:** Zeigt ein Bitmap an. Mehr ist dazu der Softimage|XSI Dokumentation zu entnehmen.

**ImageBrowser:** Zeigt einen Imagebrowser an mit dem Man Texturen von der Festplatte definieren kann.

**LightList** erlaubt dem User Lichter auszuwählen die in einer List aufgelistet werden.

**Number** ist eine standardzahlenEingabe für für Integers und Skalare wie man sie aus den PPGs kennt.

**ReferenceWidget** erlaubt mehrere Objekte auszuwählen die dem Shader als Referenz übergeben werden.

**RGB** hat den `UIType = "RGB", 3;`

**RGBA** hat den `UIType = "RGBA", 4;`

**Static** zeigt einen Parameterwert in einem statischen, das heisst unanwählbaren Textfeld an.

**String** ist eine Texteingabe die standardmäßig für Parameter vom Typ string angezeigt wird.

**Check:** Eine Check Box

**Radio** ist eine Serie von einem oder mehreren Radiobuttons. Die Elemente werden mit "Items" definiert.

**Combo** ist eine Dropdownliste mit mehreren Elementen. Die Elemente werden mit "Items" definiert.

Items werden in Item Lists für folgende UITypes verwaltet: Radio buttons, und combo boxes. Die Syntax für eine ComboBox kann zum Beispiel folgendermaßen lauten:

```
UIType = "Combo";
  Items
  {
    "Blinn" = 0,
    "Phong" = 1,
    "None" = 2,
  }
```

## A.14 Wie XSI auf Shader zugreift

*In diesem Kapitel wird erläutert, wie XSI auf Property pages für Shader zugreift und in welchem Dateiformat das Erscheinungsbild dieser Property pages definiert ist.*

In den vorigen Kapiteln wurde gezeigt mit welchem Format Mental Ray arbeitet und dass sich die Repräsentation von 3D-Objekten in Softimage und Mental Ray unterscheidet. Das ist auch verständlich, denn Softimage|XSI ist mehr als nur eine Software die das Modellieren von Geometrien ermöglicht und einen Shader im Rendertree bauen lässt. Softimage ist eine High-End Software mit der viele 3D-Künstler Filme und Effekte für die Film und Fernsehbranche anfertigen. Das Rendern gibt der ganzen Arbeit den finalen Schliff. Ohne Renderer ist es zwar noch möglich Animationen und Modelle anzufertigen und

diese zu komplexesten Szenen zusammensetzen, aber das Bild in seiner tatsächlichen Schönheit auf den Bildschirm zu bringen ist Aufgabe von Mental Ray.

Aufgrund der Tatsache, dass Softimage noch viele andere Informationen in seinen Szenen abspeichern muss, ist auch der Umgang mit Shadern, seien es die selbstprogrammierten oder im Softwarepaket enthaltenen, anders. Softimage bietet im Gegensatz zu Mental Ray eine grafische Benutzeroberfläche. Die Eigenschaften und Informationen über Objekte, Operatoren, Shader, Animationen werden in den Propertypages (PPG) dargestellt. Mit ein wenig Erfahrung lassen sich mit Hilfe der Propertypages rasch Änderungen an Parametern vornehmen oder diese Parameter animieren.

Auch Mental Ray Shader werden in Propertypages dargestellt. Man muss nicht mühevoll wie wir es in der .mi-Datei gesehen haben jeden einzelnen Parameter von Hand eintippen. Komplexe Shader können dank der SPDL-Struktur in mehrere Sektionen unterteilt werden zwischen denen man blättern kann. Der Benutzer hat alles im Überblick und kann bequem mit der Maus alle Werte für einen Shader einstellen. Genauso ist es auch möglich, Shaderparameter wie andere Parameter in Softimage|XSI zu animieren. Soll nun eine Szene gerendert werden, filtert Softimage für Mental Ray nur mehr die Informationen heraus, die der Renderer auch tatsächlich braucht. Dies erledigt das Integration API für Mental Ray, mit dem Mental Ray an Softimage angebunden ist. Mental weiß mit anderen Informationen als denen die in der Sektion „**MI-File Aufbau**“ beschrieben wurden, nichts anzufangen. Also setzt Softimage die MI-Struktur zusammen und übergibt diese an Mental Ray, der darauf prompt zu rendern beginnt.

# B Praktischer Teil

## *Einleitung*

*In Teil eins wurde schon viel zum grundlegenden Verständnis über Mental Ray erzählt. Denjenigen die Teil 1 übersprungen haben, sei an dieser Stelle das Kapitel „Die Mental Ray Architektur“ ans Herz gelegt. Das Kapitel bietet einen groben Überblick über die Arbeitsweise von Mental Ray, wie das Programm das Rendering angeht und vor allem, was es mit dem ominösen „State“ auf sich hat, mit dem in Teil zwei viel gearbeitet wird.*

## **B.1 Die Headerdatei "shader.h"**

*Wenn ein Shader programmiert wird muss immer die Datei „shader.h“ in den Quellcode mit eingebunden werden. Dies geschieht über den Befehl `include "shader.h"`. Diese Befehlszeile weist den Compiler an, nach der Datei mit dem Namen „shader.h“ zu suchen und diese mit einzubeziehen, wenn ein Shader kompiliert wird. Was dabei verborgen bleibt ist, wie die diese Datei aussieht. Der Inhalt der Datei "shader.h" wird auf den folgenden Seiten erklärt.*

In einer Headerdatei werden Funktionen und Routinen zusammengefasst, die in den verschiedensten Programmierprojekten gebraucht werden. Headerdateien sind eine Art „Bibliothek“ für ständig wiederkehrende Aufgaben. Um nicht bei jedem Programm Funktionen aufs Neue schreiben zu müssen, braucht lediglich mehr die Headerdatei mit dem Include-Befehl eingebunden werden.

Dasselbe hat es auch mit der Datei „shader.h“ auf sich. In ihr sind alle Funktionen die Mental Ray bietet angeführt, der State definiert und viele weitere Mental Ray-spezifische Definitionen. Diese Datei hat einen Umfang von 31 Seiten, würde man sie auf Courier New, Schriftgröße 8 in dieser Arbeit abdrucken. Das ist für diese Arbeit zu umfangreich, daher beschränkt sie sich auf die Erklärung der wichtigsten Inhalte.

Für die C-Syntax wird Grundverständnis vorausgesetzt. Das meiste ist ohnehin in anderen Programmiersprachen ähnlich und daher leicht verständlich. Allen, die sich in C/C++ vorher informieren wollen, seien folgende Schlagworte zur Orientierung gesagt:

*Kontrollstrukturen, Variablen, Datentypen, Pointer, <math.h>*

Mehr Basiswissen ist nicht nötig um einen Mental Ray Shader zu schreiben.

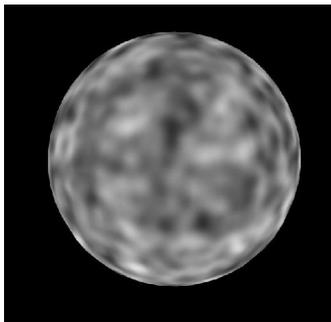
## B.1.1 Die Funktionen Strukturen und Variablen in shader.h

Die Programmierbibliothek von Mental Ray ist voll von nützlichen Funktionen die man beim Shader schreiben zu Hilfe nehmen kann. Das hat den Vorteil das Rad nicht jedes Mal neu erfinden zu müssen, sondern mit vordefinierten Funktionen arbeiten zu können. Diese machen den Code erstens überschaubarer und zweitens verbergen sie zum Teil komplizierte Algorithmen vor dem Shaderwriter. Somit wird die Kryptik der zugrunde liegenden Software vom Programmierer ferngehalten.

Dies hat zwei enorme Vorteile:

Die Programmierer von Mental Ray brauchen nicht zu fürchten, dass User durch ihren eigenen Programmcode die Funktionstüchtigkeit von Mental Ray so stark beeinträchtigen, dass die Software nicht mehr funktioniert. So wird sichergestellt, dass Benutzer nicht auf Teile von Mental Ray zugreifen können und so das stabile Laufen der Software gefährden.

Die Benutzer müssen die Software nicht bis ins kleinste Detail kennen sondern können sich ihren Shader mit Hilfe der zur Verfügung gestellten Funktionen zusammenbauen. Diese Funktionen sind vorprogrammierte Routinen deren interne Funktion dem Benutzer verborgen bleibt. Dadurch wird das Programmieren beschleunigt, weil man anstatt eines unhandlichen Codeteiles nur mehr `mi_trace_reflection()` zu schreiben braucht. Man muss keinen einzigen Gedanken daran verschwenden, wie Mental Ray intern diese Funktion durchführt.



*Abbildung 26: Perlin Noise ist in der Mental Ray Bibliothek gebrauchsfertig implementiert*

Die Vorteile sind klar: Man hat ein Set von vordefinierten Funktionen zur Hand die man nach Bedarf in seinem Programmcode verwenden kann. Dabei stehen hochkomplexe Funktionen zur Verfügung. Perlin Noise zu programmieren wäre enorm aufwändig. Diese Noise-Funktionen selbst zu schreiben würde dem Programmierer viel mathematisches Wissen, Logik, Zeit und Mühe abverlangen. Doch in der von Mental Ray zur Verfügung gestellten Bibliothek ist Perlin Noise gebrauchsfertig als Funktion vorhanden.

In „shader.h“ sind alle Funktionen angeführt, die Mental Ray bietet. Wer nach dem ausprogrammierten Code sucht, der wird ihn in der Headerdatei nicht finden. Der Code selbst wurde in die Datei „shader.lib“ ausgelagert – und wird erst beim Kompilieren aus dieser Datei gelesen und verarbeitet. LIB-Dateien und DLL-Dateien sind einander ähnlich, doch weisen sie einen funktionellen Unterschied auf, der die Koexistenz dieser beiden Systeme begründet: Angenommen man kompiliert ein Programm, das Code aus

einer .lib-Datei benötigt. Das kompilierte Programm enthält den gesamten Funktionscode, wie er in der LIB-Datei steht, in Maschinencodeform. Hingegen Programme die mit DLL-Informationen arbeiten, enthalten im kompilierten Code einen Verweis auf die Routine in der DLL-Datei. Sie benötigen zum Ausführen die DLL-Datei selbst, während dies die mit LIB's erstellten Programme von der LIB-Datei in keiner Weise mehr abhängig sind. Das erweckt jetzt den Eindruck, dass DLL's nur Nachteile bieten, aber nun kommt der Vorteil, durch den sich DLL's auszeichnen: Dieser liegt darin, dass sobald das Programm in den Speicher geladen wurde, der Funktionscode aus der DLL-Datei nur einmal in den Speicher verfrachtet werden muss. Benötigt nun ein anderes Programm genau dieselbe Funktion aus der DLL-Datei, dann kann die sich schon im Speicher befindliche Instanz verwendet werden.

Nun zu den Definition in der Headerdatei:

Wie schon erwähnt, werden in der Headerdatei sämtliche Informationen deklariert, die man benötigt um einen Shader für Mental Ray zu schreiben. Darin enthalten sind zum Beispiel sämtliche Mental Ray-interne Symbole und alle Definitionen von Enumeratoren zur Beschreibung von komplexeren Datentypen sowie der Aufbau der Strukturen (`struct`) die uns im Weiteren noch interessieren werden. Als Betrachtungsprogramm ist eine Entwicklungsumgebung mit Syntax-Highlighting zu empfehlen. Dadurch werden Schlüsselwörter im Code farblich gekennzeichnet was das Lesen des Codes wesentlich erleichtert. Dennoch dauert es eine Weile, bis man die Routine besitzt zwischen interessanten und vorerst uninteressanten Informationen zu unterscheiden. Es bleibt einem also nichts anderes übrig als sich durch den Code zu lesen, oder mit der „Suchen“ Funktion der Entwicklungsumgebung nach einem bestimmten Schlagwort zu suchen.

Die für Shaderwriter anfangs interessanten und relevanten Informationen werden im Folgenden angeführt und teilweise näher erläutert.

### B.1.1.1 Datentypen

In der Headerdatei werden Mental-Ray-spezifische Datentypen per „Typedef“ definiert.

```
typedef enum {miFALSE=0, miTRUE=1}          miBoolean;
typedef unsigned char                      miCBoolean; /* if space is tight */
typedef int                                 miInteger; /* for shader params */
typedef signed char                         miSint1;
typedef unsigned char                       miUint1;
typedef signed short                        miSint2;
typedef unsigned short                      miUint2;
typedef signed int                          miSint4;
typedef unsigned int                       miUint4;
typedef unsigned int                       miUint;
typedef unsigned char                      miUchar;
typedef unsigned short                     miUshort;
```

```

#if defined(WIN_NT) && defined(BIT64)
typedef unsigned __int64      miUlong;
typedef signed   __int64      miSlong;
#else
typedef unsigned long        miUlong;
typedef signed   long         miSlong;
#endif

typedef signed char          miSchar;
typedef float                miScalar;
typedef double                miGeoScalar;
typedef struct {miScalar u, v;} miVector2d;
typedef struct {miScalar x, y, z;} miVector;
typedef struct {miGeoScalar u, v;} miGeoVector2d;
typedef struct {miGeoScalar x, y, z;} miGeoVector;
typedef struct {miScalar x, y, z, w;} miQuaternion;
typedef miScalar             miMatrix[16];
typedef struct {float r, g, b, a;} miColor;
typedef struct {miScalar min, max;} miRange;
typedef struct {miGeoScalar min, max;} miGeoRange;
typedef unsigned short       miIndex;
typedef unsigned int          miGeoIndex;
typedef miBoolean            (*miFunction_ptr)(void);
typedef union {void *p; double d;} miPointer; /* 8-byte aligned ptr*/
typedef miUint                miTag;
typedef miSint4                miThreadID;
typedef miSint4                miHostID;
typedef miSint4                miSessionID;
typedef miSint4                miTransactionID;
typedef miSint4                miWorldID;

```

Wie man sieht, werden Floats in Mental Ray als `miScalar` bezeichnet, Integers als `miInteger`. Für jeden möglichen Datentyp sind Typisierungen durch `typedef` vorgesehen. Dies hat den Grund, dass der Shadercode auf andere Systeme portabel ist ohne dass man fürchten muss, dass Unstimmigkeiten des Codes mit dem Betriebssystem entstehen. In der `shader.h`-Datei wird darauf Rücksicht genommen.

Weiters werden neue Datentypen wie `miVector(2D)` für die Repräsentation von Koordinaten im 2- oder 3-Dimensionalen Raum definiert. Diese Datentypen werden im Verlauf der Headerdatei noch häufig zum Einsatz kommen.

### B.1.1.2 TAGs

Es gibt den Datentyp `miTag` in Mental Ray, einer wohl rätselhaftesten Datentypen. Er ist überall in `shader.h` zu finden, kann aber keinem konkreten Zusammenhang zugeordnet werden. Zur Erklärung muss etwas weiter ausgeholt werden:

Mental Ray läuft auf mehr als nur einem Rechner. Die Software wurde so konzipiert, dass sie auf Multiprozessormaschinen läuft und auch Rendering über das Netzwerk auf mehreren Rechnern unterstützt. Ein Rechner übernimmt die Verwaltung des Renderings. Dieser Rechner teilt den anderen Geräten Arbeitspakete zu und verwaltet alle renderrelevanten Daten, die sogenannte Scenedatabase. Es ist bekannt, dass Pointer die auf einem System gültig sind, auf einem anderen Computer auf einen falschen

Bereich im Speicher zeigen. Da Mental Ray mit Pointern arbeitet und diese Computerübergreifend zum Einsatz kommen, repräsentieren Tags systemunabhängig immer dieselben Daten [MRDOC1, 2004, node139.html]. Das heisst, dass Tags ihre Gültigkeit im gesamten Netzwerk in dem die Mental Ray Rendermaschinen arbeiten, gültig sind. Tags repräsentieren alle möglichen Arten von Daten in der Scenedatabase. Wird ein Pointer zu einem Scenedatabase-Element benötigt, kann man mit "mi\_db\_"-Funktionen dazugehörigen Pointer anfordern. Dieser Pointer wird daraufhin gepinnt, also reserviert. Dies garantiert, dass sich die Adresse des DB-Elements während des Zeitraumes in dem man den Pointer darauf benötigt nicht ändert. Tags die mit „mi\_db\_“-Funktionen gepinnt wurden, müssen mit `mi_db_unpin( const miTag tag)` wieder freigegeben werden – andernfalls stürzt Mental Ray früher oder später ab.

### B.1.1.3 Konstanten:

Natürlich werden in Programmen, die aus 3D-Szenen 2D-Bilder berechnen sollen auch trigonometrische Funktionen eingesetzt. Bestimmte Werte werden daher in der Headerdatei „shader.h“ schon vordefiniert, wie zum Beispiel die Werte von PI, PI/2 und der Wurzel aus ½.

```
#define M_PI           3.14159265358979323846
#define M_PI_2        1.57079632679489661923
#define M_SQRT1_2     0.70710678118654752440
```

Benötigt man den Wert von PI, braucht man nur "M\_PI" zu schreiben und der Compiler ersetzt diesen Wert später mit dem tatsächlichen Zahlenwert.

In der Headerdatei sind auch Hilfsfunktionen zur Vektorrechnung zu finden. Diese sind ebenfalls durch #define – Prozeduren implementiert, was zwar den kompilierten Code länger macht, zur Laufzeit dafür den Funktionsaufruf erspart. Die Funktionen dazu sind im Anhang zu finden.

## B.1.2 Die Struktur miState

Es soll jetzt nicht zu viel Zeit damit verbracht werden die Funktionen und Variablen einzeln durchzugehen. Später wird auf ausgewählte Funktionen welche zum Schreiben des Demonstrationsshaders benötigen werden genauer eingegangen. Was jetzt von Interesse ist, ist die Struktur namens „miState“, das Informationspaket mit dem Shader untereinander während des Renderns kommunizieren, sei es dass sie Kopien ihres eigenen States oder die Adresse ihres eigenen States an andere Shader weitergeben.

Die Struktur wird mit

```
typedef struct miState { ... } miState;
```

deklariert. Zwischen den geschwungen Klammern befinden sich die Variablen des States. Im State findet der Shader die wichtigsten Informationen die er zum Rendern benötigt. Die wichtigsten Variablen werden hier angeführt und nebenbei erklärt. Die vollständige Struktur kann in der Headerdatei eingesehen werden.

```
miTag      camera_inst;          /* camera instance */
```

Die Instanz der Kamera, von deren Perspektive die Szene gerendert wird

```
struct miCamera *camera;        /* camera */
```

In der Struktur „camera“ finden sich alle Einstellungen, die für die Renderkamera getroffen wurden wie Bilddimension, Lensshader, Volumeshader, Clipping Planes, also alle Einstellungen die in der MI-Datei für die Kamera getroffen wurden.

```
struct miOptions *options;     /* options */
```

Der Pointer auf die Struktur miOptions enthält alle eingestellten Optionen die zum Rendern der Szene getroffen wurden.

```
float      raster_x;           /* x in raster space */  
float      raster_y;           /* y in raster space */
```

Die Variablen raster\_x und raster\_y geben die Koordinaten des Pixels, das gerade gerendert wird an.

```
struct miState *parent;
```

Die meisten Shader besitzen einen Vorgänger – den sogenannten „parent“. Über den Parentpointer kann auf die Information des Shaders, der sich in der Reihenfolge vor dem aktuellen Shader befindet, zugegriffen werden. Für den ersten Eye-Ray oder den ersten Lens Shader ist der Wert dieses Pointers „NULL“. Bei hintereinander folgenden Lens Shader zeigt der Pointer auf den State des vorhergehenden Shaders. Bei Light-Rays zeigt er auf den darüberliegenden Ray und bei Shadow-Rays auf den State des Light-Rays der den Shadow-Ray initiiert hat. [

```
miRay_type type;               /* type of the ray */
```

Über diese Variable erfährt man, von welchem Typ ein Ray ist. Es gibt eine Menge von verschiedenen Ray-Typen. Diese sind in dem Enumerator miRay\_type festgelegt. (Siehe Anhang, Seite 123) Mit

dieser Information ist es möglich Shader zu schreiben, die als verschiedene Shader gleichzeitig verwendet werden können, wie zuvor in dieser Arbeit schon erwähnt wurde. Der Vorteil: Man hat in Softimage|XSI nur eine einzige Propertytype die man einzustellen braucht und kann dann Shader gleich an mehrere Eingänge des Materialnodes im Softimage Rendertree anhängen. Viele Material Shader sind so programmiert, dass man die auch als Shadow- und Photon Shader verwenden kann. Man bringt an den entsprechenden Stellen im Shadercode eine Abfrage nach dem Ray-Type an und kann die benötigten Berechnungen zusätzlich durchführen oder eben weglassen. Zum Beispiel darf der Shader, der als Shadow Shader aufgerufen wurde (das heißt, der `miRay_type` ist vom Typ `miRAY_SHADOW`), nicht die Funktion `mi_sample_light()` verwenden.

```
miUchar    qmc_component;    /* next component of current
                             * instance of low discrepancy
                             * vector to be used */
```

Diese Variable ist für das Sampling nach der Quasi-Monte-Carlo Methode [QMC1, 1994, s. 1-12]. Diese Methode ist ein spezieller Sampling-Algorithmus der bessere Samplingwerte liefert in dem die Samplingpunkte in einer speziellen Anordnung über dem Pixel verteilt werden. Mit dieser Variable wird in ganz seltenen Ausnahmefällen von Shaderwritern gearbeitet.

```
miUint1    scanline;        /* intersect ray by scanline?*/
```

Diese Variable gibt an, ob die Intersection mit der Scanline-Methode oder mit Raytracing gefunden wurde.

```
miCBoolean inv_normal;     /* normals are inverted because
                             * ray hit the back side */
char        face;          /* f)ront, b)ack, a)ll, from
                             * state->options for subrays*/
```

Polygone haben eine Vorder- und eine Rückseite. Trifft ein Ray von der Rückseite auf ein Polygon, dann invertiert Mental Ray die Vorder- und die Rückseite entsprechend.

```
int         reflection_level; /* reflection depth of the ray
                             * SHOULD BE SHORT
                             */
int         refraction_level; /* refraction depth of the ray
                             * SHOULD BE SHORT
                             */
```

In den Renderoptionen kann eine maximale Strahltiefe für Reflexions- und Brechungsstrahlen sowie die Gesamtstrahltiefe angegeben werden. Im `reflection_level` und im `refraction_level` ist die

Tiefe des jeweiligen Rays festgehalten.

```
miVector  org;          /* ray origin
                        * for light, shadow rays the
                        * light position */
miVector  dir;          /* ray direction
                        * for light, shadow ray
                        * it points
                        * towards the intersection */
```

Die nächsten Variablen geben Aufschluss über Intersections und Rays generell. Je nachdem um welche Art von Shader es sich handelt, kann der Inhalt dieser Variablen undefiniert sein. außerdem muss man wissen als welcher Shader die Funktion fungiert, weil sich davon abhängig die Richtung und somit die Bedeutung des Vektors ändert, wie man aus dem Kommentar unschwer erkennen kann. `org` ist der Ursprung des Rays und ist als Raumkoordinate zu verstehen. Bei `dir` handelt es sich um die Richtung des Rays, daher versteht sich diese Variable als Raumvektor.

```
double    dist;        /* length of the ray */
```

Die Länge des Rays kann über die Variable `dist` abgefragt werden.

```
miScalar  ior;          /* ior of outgoing ray */
miScalar  ior_in;       /* ior of incoming ray */
```

Die Variablen `ior` und `ior_in` sind freie Variablen die Mental Ray selbst nicht verwendet. Diese Variablen werden eingesetzt, wenn man Shader für Lichtbrechung schreibt: Man kann darin die Brechungsindizes der Materialien, die der Ray durchquert, festhalten. Dies ist notwendig, um die jeweils korrekte Berechnung durchführen zu lassen, je nachdem ob ein Ray in ein lichtbrechendes Material eindringt, oder dieses verlässt.

```
miTag     material;     /* material of the primitive.
                        * Set to miNULL_TAG if no hit.
                        */
miTag     volume;       /* volume shader to be applied*/
miTag     environment;  /* environment shader to be
                        * applied if no hit */
miTag     refraction_volume; /* volume shader to apply
                        * to refraction ray */
```

Die diversen Tags verweisen auf die Shader, die nach dem jetzigen Ray zum Einsatz kommen oder kommen könnten. Environment Shader werden aufgerufen, nachdem ein Strahl die Szene verlassen hat ohne dass er ein weiteres Objekt getroffen hätte. Volume Shader werden am Ende eines Rays aufgerufen und können den berechneten Farbwert nachträglich verändern.

```
miUInt          label;          /* label of hit object */
```

Jedes Objekt kann ein Label besitzen, welches nach dem Rendern in den Tag-Buffer geschrieben wird, falls ein solcher angefordert wurde (Dies kann man in XSI indem man den Renderoptions unter Output die Tags anhakt (nicht zu verwechseln mit den MI-Tags!))

```
miTag          instance;        /* hit object instance */
```

Die `instance` ist sehr praktisch um festzustellen, um welches Objekt es sich konkret handelt. Jedes Objekt hat eine eigenen, einzigartigen `instance`-Wert. Man kann anhand dieses Tags ermitteln ob `child->instance` nach einem `mi_trace_reflection()`-Aufruf den selben Wert hat oder nicht. So lässt sich zum Beispiel einen Shader gestalten der Selbstreflexionen verwirft oder umgekehrt, nur Selbstreflexionen in Betracht zieht.

```
miTag          light_instance;  /* light instance */
```

Die Lichtinstanz verweist auf die Lichtquelle, sollte der Shader ein Light oder ein Shadow Shader sein.

```
void          *pri;             /* box_info of hit box, shaders  
                                * can check if pri== NULL  
                                */  
int           pri_idx;          /* index of hit primitive  
                                * in box */
```

Die Hitbox identifiziert ein einzelnes Dreieck (Mental Ray kann nur Dreiecke behandeln). Diese Dreiecke sind in Boxen enthalten. Manche Shader wie zum Beispiel Light Shader testen, ob die Oberflächennormalen von dem Ray wegzeigen und werden dann gar nicht ausgewertet. Deswegen kann man der Variablen den wert `pri = NULL` zuweisen um dieses Verhalten zu vermeiden. Doch das hat auch Nachteile, da einige `mi_query`-Funktionen nicht mehr funktionieren. Volumeshader weisen in der Regel `pri` immer den Wert `NULL` zu, da sie keine Geometrie treffen, sondern wirklich ein Volumen auswerten. Alle Shader außer Volume Shader (die immer als letzte Shader aufgerufen werden) müssen den Wert von `pri` wiederherstellen, bevor sie ihren Funktionskörper verlassen!

Weitere interessante geometrische Daten im State sind:

```
miVector    point;          /* point of intersection */
```

Der Punkt an dem die Intersection stattfand nennt sich point.

```
miVector    normal;        /* interpolated normal
                             * pointing to the side of the
                             * ray */
```

An dieser Stelle gibt es auch eine interpolierte Oberflächennormale

```
miVector    normal_geom;   /* geometric normal of pri.
                             * pointing to the side of the
                             * ray */
```

Sowie eine uninterpolierte Normale, die genau rechtwinkelig auf die getroffene Fläche steht.

```
miScalar    dot_nd;        /* dot of normal, dir
                             * for light rays,
                             * it's the dot_nd of the
                             * parent (intersection) normal
                             * and the light ray dir */
```

Das Skalarprodukt zwischen der Oberflächennormalen und dem Strahl der auf die Oberfläche trifft, ist in der Variablen dot\_nd festgehalten. Es gibt genügend Shader die diesen Wert benötigen, daher wird er von Mental Ray automatisch zur Verfügung gestellt.

```
double      shadow_tol;    /* the minimum distance to a
                             * plane perpendicular to the
                             * shading normal for which the
                             * triangle is completely on
                             * the negative side. */
```

Die Shadow Tolerance verhindert, dass aufgrund von kleinen Fließkommafehlern fehlerhafter Schattenwurf berechnet wird. Das kann beispielsweise bei Oberflächen die fast parallel zum Lichtstrahl ausgerichtet sind auftreten. Aufgrund von numerischen Ungenauigkeiten des Computers kann es dabei zu Selbstbeschattung kommen. Daher werden Shadow Rays, die kleiner als die Shadow- Tolerance sind nicht beachtet.

```
miVector    *tex_list;     /* list of texture coordinates
                             */
miVector    *bump_x_list;  /* list of bump map basis x
                             * (perturbation) vectors */
miVector    *bump_y_list;  /* list of bump map basis y
                             * (perturbation) vectors */
```

In der Texture List befinden sich die Texturkoordinaten für Shader, die mehrere Texturespaces unterstützen. Die gebräuchlichste Anwendung der Texlist ist, dass man die jeweiligen Koordinaten aus

der Liste herausliest und in die Variable `tex` schreibt, die damit einen Texture-Lookup vollziehen kann.

Das gilt auch für die Bump-Lists.

```
miVector motion;          /* motion vector or (0,0,0) */
```

Bewegte Objekte besitzen einen Motionvektor der ebenfalls für Shadingzwecke zur Verfügung steht.

```
miVector tex;             /* texture coordinates for  
                          * lookup */
```

In `tex` sind die aktuellen Texturkoordinaten (U, V und W) gespeichert, die für Texturlookups eingesehen werden können

```
struct miState *child;    /* child state */
```

Die meisten Shader belassen es nicht bei dem Ray der sie getroffen hat, sondern veranlassen weitere Rays und somit weitere Shader. Ein Shader der Strahlen ausschließlich reflektiert, verändert lediglich die Richtung des Strahls und schickt den Strahl, sollte er die maximale Strahlentiefe noch nicht überschritten haben, weiter in die Szene. Ist dieser Vorgang abgeschlossen, hat der darüberliegende Shader noch Zugriff auf den State des Rays. Dies geschieht mit der Variablen `child`, welche ein Pointer auf die vorige Statestruktur ist. Sie existiert nur für den zuletzt aufgerufenen Shader, also `state->child->child` ist nicht definiert!

```
miUshort count;          /* area light sample counter */
```

Dies ist eine Zählvariable für Arealights (erst ab Mental Ray Version 3.1)

```
void *user;              /* user data */  
int user_size;           /* size of user data */
```

Ganz anspruchsvolle Shaderwriter können Shader mit User-Data erweitern, also Mental Ray-fremder Information. Dies ist sinnvoll, wenn man verschiedener Shader miteinander auf eine Weise kommunizieren lassen will, die über die verfügbaren State-Variablen nicht mehr möglich ist.

## B.2 Übergang von der Theorie zur Praxis – der erste Shader

*Um einen Shader zu schreiben müssen noch ein paar Vorbereitungen getroffen werden. Sich Zeit zu nehmen bevor man mit dem Programmieren anfängt macht sich bezahlt.*

Es gibt in Softimage|XSI eine Fülle von Shader, die dank Shaderassignment im Rendertree zu großen komplexen Gebilden kombiniert werden können. Das Shaderassignment ermöglicht, dass ein Shader ihren Ergebniswert an den nächsten Shader weiterreichen. Somit kann man aus vielen kleinen Nodes ein mächtiges Konstrukt zusammenbauen. Nachteilig dabei ist, dass mit wachsender Anzahl an Shadernodes der Zeitaufwand beim Rendern manchmal exponentiell größer wird. Das hat mehrere Gründe:

- Jeder Shader braucht eine gewisse Zeit bis er sein Ergebnis ermittelt hat. Je mehr Shader verbunden werden, desto mehr Funktionen müssen berechnet werden und desto länger wird auch die Renderzeit.
- Gegenseitige Funktionsaufrufe verursachen natürlich auch einen Overhead: Das Programm muss im Speicher zu einer anderen Funktion springen, gegebenenfalls Variablen übergeben und dort seine Arbeit fortsetzen. Ist der Funktionsaufruf beendet, springt der Renderer wieder an die Stelle zurück, von der aus er die Funktion aufrief. Das kostet natürlich Zeit und zusätzlichen Speicher. Bei kleinen Rendertrees ist dies kaum spürbar. Aber bei großen Rendertrees mit 10 Nodes macht sich das natürlich bemerkbar. Vor allem, wenn diese Arbeit jedes Mal pro Sample erledigt werden muss. Das sind beim Berechnen eines einzigen Bildes hunderttausende Funktionsaufrufe, die hintereinander getätigt werden. Da wird klar, dass sich die Optimierung eines Shaders oder Rendertrees durchaus bezahlt macht um jede auch noch so kleine unnötige Arbeit zu vermeiden.
- Man hat zu wenig Kontrolle über Rays. Gerade diese Renderoptimierung macht vielen 3D-Artisten zu schaffen, da man gelegentlich gezwungen ist einen Kompromiss zwischen Geschwindigkeit und Resultat einzugehen. Angenommen man hat eine Szene mit vielen gläsernen Elementen die gleichzeitig einen Teil des Lichtes reflektieren und einen Teil brechen. Das heisst, an einer Intersection werden 2 neue Rays erzeugt: Stehen die Elemente dicht gedrängt ist es äussert wahrscheinlich, dass die neuen Rays ihrerseits wieder auf dasselbe Material stoßen und wieder einen Reflection und einen Refraction-Ray erzeugen. Im Extremfall summiert sich das bei einer maximalen Strahlentiefe von 8 auf  $2^8 = 256$  Rays für nur ein einziges Sample. Nimmt man an, dass gerade bei

solchen Szenen die Samplingdichte auf mindestens 16 Samples pro Pixel gedreht ist, so ergeben sich nur für die Berechnung eines einzigen Pixels  $256 * 16 = 4096$  Rays, von der Anzahl der Rechenoperationen welche die Shader dafür ausführen müssen ganz zu schweigen. In solchen Fällen zahlt es sich aus die maximale Strahltiefe um 1 zu verringern, was die maximale Anzahl an Rays prompt halbiert. Natürlich kann es dann vorkommen dass Rays den Ergebniswert "schwarz" zurückliefern, was bei lichtbrechenden Objekten sehr störend auffällt.

In solchen Fällen sind schlaue Shader gefragt die für jedes individuelle Problem eine gute Lösung bieten. Nicht immer ist es möglich aufwändige Berechnungen zu vermeiden, aber langen Renderzeiten lässt sich oft vorbeugen.



*Abbildung 27: Caipirinhaglas mit Eiswürfeln. An den Glasrändern ist deutlich zu erkennen, dass der Renderer die maximale Raytracingdepth erreicht hat. Unschöne schwarze Bereiche sind die Folge.*

Noch ein Grund, der manchmal gegen die Verwendung eines Rendertrees spricht ist, dass man oft zu wenig Kontrolle über den State hat. Der State ist die Informationseinheit welche Kommunikation zwischen verschiedenen Shadern ermöglicht. Mental Ray erledigt eine Menge dieser Kommunikation automatisch. Man muss sich nicht um Intersectionberechnung kümmern wenn man einen Shader im Rendertree zusammenbaut. Das ist zugleich die Stärke und Schwäche des Rendertrees: Er versucht den komplizierten Inhalt von Mental Rays Shadingfunktionalität auf einfache Weise dem User in Softimage|XSI zugänglich zu machen.

Dies geschieht, indem Softimage mit einer großen Bibliothek an Basis Shadern geliefert wird. Diese Shadernodes lassen sich im Rendertree zu mächtigen Shadertrees verarbeiten. Die einzelnen Nodes

erfüllen nur Basiszwecke die sich zu beliebiger Komplexität verbinden lassen. Das ist für weniger belaufene Anwender die beste und sicherste Methode, sich nicht durch unüberschaubare Shader wählen zu müssen. Versierte Benutzer vermissen hingegen die Kontrolle über gewisse Mechanismen. Dadurch passiert es oft, dass Shader zu rechenaufwändig werden, weil Materialien scheinbar außer Kontrolle geraten und unnötige Berechnungen durchführen.

Gerade in der Unterhaltungsbranche geht es weniger darum physikalische akkurate Ergebnisse zu haben, als Effekte zu liefern die den Zuseher überzeugen. Die nach wie vor komplexesten Phänomene sind Lichtbrechungen gepaart mit Reflexionen. Werden diese gleichzeitig eingesetzt, ergibt sich mit wachsender Raytracingtiefe ein exponentielles Wachstum der Renderzeit. Es ist schwer einem Shader im Rendertree klar zu machen, ob und wann er mit der Berechnung eines Rays aufhören kann wenn die Fortsetzung der Berechnung nur mehr eine marginale Änderung des Pixels bringen würde. Für solche Zwecke zahlt es sich aus, angepasste Shader selbst zu programmieren und diese so zu gestalten, dass sie gewisse Optimierungen vornehmen. Zum Beispiel kann man einen Shadow Shader schreiben der die Weiterberechnung stoppt, sobald der Schatten praktisch voll opak ist, also *fast* keine Transparenz mehr durch lichtdurchlässige Objekte hat. Ein korrekt programmierter Shadow Shader würde seine Aufgabe stur fortsetzen bis er auf ein undurchsichtiges Objekt stößt oder den Intersectionpoint erreicht.

## B.2.1 Vorüberlegungen

Bevor man sich ans Werk macht um selbst einen Shader zu schreiben, sollte man den genauen Zweck des Shaders noch einmal überdenken. Es gibt dazu mehrere Beweggründe.

### B.2.1.1 Komfort

Man verfasst keinen neuen revolutionären Shader sondern arrangiert sich eine Palette an Funktionalitäten in einem Stück Code so zusammen, dass man einen Shader für einen bestimmten häufigen Gebrauch optimiert.

### B.2.1.2 Optimierung

Bestimmte Rendertrees benötigen beim Rendern zu viel Zeit und sind unüberschaubare Konstrukte. Hier kann der findige Shaderwriter aus vielen kleinen Shadernodes einen Code bauen, der zur Renderzeit zusätzlich Optimierungen vornimmt indem er einen Shader schreibt, welcher die Lichtbrechungs- und Reflexionsstrahlen kontrolliert. So ist es möglich dass ein Shader dem nächsten Aufruf zu erkennen gibt,

dass sein Beitrag gemessen am Gesamtergebnis des Samplingwertes nur mehr gering ist und aufgibt obwohl die maximale Strahltiefe noch nicht erreicht ist. Es gibt genügend Situationen, in denen solche Fälle auftreten. Was Produzenten mit einem Rechencluster egal ist, macht für kleine Systeme durchaus Sinn wenn man mit Tricks die Rechenzeit verkürzen kann.

#### B.2.1.3 Was begründet das Schreiben des Shaders genau?

Hier soll sich der User fragen, ob der Effekt nicht doch im Rendertree nachgebildet werden kann. Oft sind es Kleinigkeiten die ein Shader vermissen lässt und die man sich selbst implementieren kann. Zum Beispiel ist es in manchen Fällen bequem in den Shader sofort eine Farbkorrektur einzubauen die sich über Parameter steuern lässt. Zum Beispiel kann man einen Texture-Lookupshader gleich mit einer Gammakorrektur zur Renderzeit verbinden und dieses Ergebnis per Shader-Assignment an einen anderen Shader weitergeben.

Oder man programmiert sich mit Hilfe der verfügbaren Perlin-Funktionen neue prozedurale Textur-Shader. In diesem Fall ist natürlich schon fundierteres mathematisches Wissen notwendig um aus einfachen Perlinmustern komplexe Strukturen zu erzeugen, aber es ist grundsätzlich möglich.

#### B.2.1.4 Wie präzise soll der Shader werden?

Man kann seinen Code natürlich nach Belieben justieren um jede noch so unnötige Renderzeit einzusparen. Natürlich stellt sich die Frage welchem Zweck der Shader dienen soll. Für architektonische Renderings sind Shader wichtig, welche die Interaktion von Material und Licht physikalisch korrekt berechnen damit man ein aussagekräftiges Bild erhält, das die tatsächliche Beleuchtungssituation der Realität widerspiegelt. Wenn es jedoch darum geht, einen Effekt für eine Fernsehproduktion umzusetzen wird dem Produzenten mehr daran liegen, dass das Videomaterial rechtzeitig fertig ist und beim Zuseher gut ankommt als dass es physikalisch korrekt berechnet wurde.

Man sieht, es ist vom „Tool“-Shader der statt neuer Funktionen nur eine Erleichterung für spezifische Aufgaben des 3D-Alltages bietet, bis zum höchstkomplexen Volumeshader alles möglich. Beim Shaderwriting können sowohl mathematisch versierte Benutzer als auch der Durchschnittsuser ans Werk gehen und los schreiben.

## B.3 C-Coden, Teil 1 – Vorüberlegungen

*Nachdem die Entscheidung gefallen ist, einen Shader nicht im Rendertree von Softimage zu bauen, sondern sich die Mühe zu machen ihn selber zu programmieren, sind viele Vorarbeiten nötig die unbedingt zu erledigen sind. Es ist Vorbereitungszeit nötig bevor man zur Tat schreitet, egal ob es sich nun um einen kleinen Shader handelt oder um einen komplexen. Gute Kenntnisse im Rendertree von Softimage und eine Vorstellung von dem gewünschten Resultat sind dafür die Voraussetzung.*

Bevor man sich an den Computer setzt, sollte man auf einem Blatt Papier festhalten, was man sich von dem Shader erwartet. Das heisst, man muss sich für den Typ des Shaders entscheiden, ob es ein Volume, ein Shadow Shader oder ein anderer der breiten Shaderpalette werden soll. Dies ist im Folgenden sehr wichtig, da man dies im Shaderwizard, der im kommenden Kapitel durchgenommen wird, einfach einstellen kann. Nachhaltige Änderungen sind nur umständlich zu erledigen und wesentlich zeitaufwändiger.

Je nach Art des Shaders besitzt er mehrere (eventuell per Shader Assignment zuweisbare) Parameter, über die sich das Verhalten des Shaders steuern lässt. Der Shader reagiert je nach Einsatz auf bestimmte „Ray-Types“, interagiert mit Lichtern in der Szene und liefert Resultate vom Typ miColor, miScalar, miVector, miBoolean, miMatrix oder miInteger. Shader wie Geometryshader liefern gar keinen Ergebnistyp, da sie mit den anderen Shadern kaum mehr Ähnlichkeiten haben. Aber hier soll das Hauptaugenmerk auf Shader gerichtet sein, die vor allem mit Farbwerten arbeiten, da diese die interessantesten Ergebnisse liefern und für das Grundverständnis am besten geeignet sind. Mit dem beiliegenden Codematerial sollte es dann einfach sein, neue und andere Shader zu entwickeln oder das hier dargestellte Modell zu verbessern und zu erweitern.

Als nächstes stellt sich noch die Frage, welche wissenschaftlichen Besonderheiten der Shader aufweist. Wenn komplizierte Berechnungen im 3D-Raum für fraktale Shader vorgenommen werden, zahlt sich die vorhergehende Recherche zu solchen Themenbereichen auf jeden Fall aus. Es ist besser sich vor dem Programmieren in vertiefende Materie einzulesen, als während dem Programmieren feststellen zu müssen, dass man an einem bestimmten Problem scheitert. Für prozedurale Muster gibt es im Internet hunderte Websites und Bücher.

## B.4 Beispielshader „LumaReflect“

*In diesem Beispiel wird ein Shader programmiert, der Reflexionen abhängig von ihrer Luminanz zu der Materialfarbe beimischt. Das Reflexionsmodell, das in den Standardshadern von Softimage implementiert ist, dämpft die Originalfarbe des Objektes ab, je stärker man die Reflexion einstellt.*

Das Reflexionsmodell das allen Basis-Shadern aus Softimage zueigen ist berechnet perfekte Reflexionen. Das ist ein im Regelfall unerwünschtes Verhalten. Ein spiegelndes Objekt wie das Auto in Abbildung 28 kann nur Lichtstrahlen spiegeln, die tatsächlich auf den Lack treffen. Wo kein Licht auf das Auto auftrifft, kann dieses auch nichts spiegeln. Trifft aus der Reflexionsrichtung kein Licht auf die Oberfläche, kann dieses auch nicht gespiegelt werden. Dann dringt die diffuse Farbe des Lacks zum Vorschein, der sich unter dem Klarlack befindet. Dieses Phänomen tritt bei allen Oberflächen auf, die mit durchsichtigen Materialien beschichtet sind. Auch fettige Haut glänzt aufgrund dieser Tatsache.



**Abbildung 28:** Reflexionen auf einem Auto

Die dahinter stehenden physikalischen Gesetze sollen hier nicht untersucht werden. Von Interesse ist die einfache Umsetzung dieses Effektes.

Weiters ist bei matten Oberflächen oft zu bemerken, dass sie nur Objekte in nächster Nähe spiegeln. Je weiter sich das Objekt von der Oberfläche entfernt, desto diffuser wird es, bis schließlich nur mehr die diffuse Oberflächenfarbe zu sehen ist. Dieser Effekt soll ebenfalls rechengünstig in den Shader mit aufgenommen werden. außerdem sollen Reflexionen gleich auf

der Materialoberfläche farbkorrigiert werden können. Das heisst, man kann mit einem Slider einen Threshold für die Minimalhelligkeit angeben, ab der eine Reflexion in Betracht gezogen werden soll.

Des Weiteren soll der Shader die üblichen Einstellungen für Highlights haben. außerdem soll das Specularmodell der Reflexion selbst eingestellt werden können, also ob die Glanzlichter nach dem Phong oder dem Blinn-Modell berechnet werden. Beim Blinn-Modell hat man mehr Einfluss auf das Aussehen des Highlights während beim Phongmodell nur ein einziger Parameter die Breite des Glanzlichtes bestimmt. Sämtliche Parameter bis auf die Auswahl ob Blinn oder Phong-Specular sollen per Shaderassignment gesteuert werden können.

Zusammenfassend muss der Shader also folgende Spezifikationen erfüllen:

- Normales Shading mit wählbarem Phong / Blinn-Specular Highlight
- Reflexionen sollen wie gewohnt mit einem RGBA-Slider eingestellt werden können
- Spiegelungen sind abhängig von der Helligkeit des gespiegelten Objektes.

Des Weiteren soll man zwischen der herkömmlicher Reflexion und der Lumareflexion fließend hin- und herblenden können.

Reflexionen können mit zunehmender Distanz ausgefadet werden. Um die Abnahme gleichmäßiger werden zu lassen nimmt sie mit der Wurzel der Entfernung ab, da die Objektgröße mit dem Quadrat der Entfernung abnimmt.

Reflexionen können schon während des Shadings bearbeitet werden können. Dazu kann ein Threshold unter dessen Helligkeitswert es keine Reflexionen mehr gibt, eingestellt werden. Per Checkbox kann man eine S-curve aktivieren, die einen nichtlinearen Übergang von Nicht-Reflexion zu Reflexion macht. Siehe Abbildung 32 auf Seite 106.

Nun ist der nächste Schritt nötig: Man muss sich überlegen welche Parameter der Shader hat und von welchem Datentyp die Parameter sind, die der User selber einstellen kann. Für den Code werden englische Bezeichnungen gewählt, da die englische Sprache in der Programmierung Standard sein sollte. Besonders wenn der Code von anderssprachigen Usern gelesen wird verstehen die meisten davon Englisch und können mit den Parameternamen etwas anfangen. Deswegen werden auch aussagekräftige, kurze Namen vergeben. Lange Variablenamen verursachen Codezeilen die über den Bildschirmrand hinausgehen und unbequem zu lesen sind. Die Namensgebung wie sie hier getroffen wurde ist nicht die kürzeste, aber aus Gründen der Verständlichkeit für den Leser wurde in Zweifelsfällen ein längerer Variablenname gewählt. Für den Shader werden folgende Interface Parameter benötigt:

baseColor	RGBA
specularColor	RGBA
specularMode	Integer
specularDecay (bei Phong)	Scalar
roughness(Blinn)	Scalar
specularRefraction (Blinn)	Scalar
reflection	RGBA

lumaAmount	Scalar
lumaThreshold	Scalar
s_curve	Boolean
fadeOut	Boolean
inverseFade	Boolean
squareFade	Boolean
nearDist	Scalar
farDist	Scalar

Der Shader liefert als Ergebnis wieder einen Farbwert, der natürlich auch wieder anderen Shadern per Shader-Assignment übergeben werden kann. Das ermöglicht es, den Shader im Rendertree von Softimage mit anderen Shadern zu verknüpfen.

## B.5 Der Shader-Wizard

*Der Shader-Wizard ist ein nützliches Werkzeug zum Erzeugen der Ausgangsdateien, die zum Programmieren eines Shaders nötig sind. Welche Funktionen der Shader-Wizard bietet, erfährt man auf den folgenden Seiten.*

Hat man die Vorarbeiten erfolgreich abgeschlossen, kann man zum nächsten Schritt übergehen. Im Shader Wizard werden die Basis-C-Dateien und die dazugehörige SPDL-Datei erzeugt. Mit Softimage|XSI wird nämlich auch ein SDK mitgeliefert, das den sogenannten Shader-Wizard beinhaltet. Der Shader Wizard ist ein Java-Applet das im Browserfenster läuft. Dieses Programm nimmt einem viel Arbeit ab, indem es auf einer grafischen Benutzeroberfläche dem User den Rohshader zusammenbaut. Dieser Rohshader besteht dann aus folgenden Dateien, die das Programm schon entsprechend ausgefüllt in ein ausgewähltes Verzeichnis schreibt:

**<shader>.spdl file: Definiert das Userinterface in Softimage|XSI**

**<shader>.h: Enthält alle Parameter für den Shader**

**<shader>.cpp: Die Implementierung des Shadercodes (in C/C++)**

**<shader>\_dll.cpp: Die callbacks des Shaders für die DLL-Datei**

**<shader>.dsp: Ein Project File für den Visual C++ Compiler.**

**<shader>.gnu: Das makefile für den gnu Compiler.**

**<shader>\_install.bat: Das Installationskommando für den Shader unter Windows**

## **<shader>\_install.csh Das Installationskommando für den Shader unter Unix/Linux**

Ohne dem Shaderwizard wäre noch viel mehr Tipparbeit zu erledigen. Die SPDL selbst zu schreiben ist ein nicht unaufwändiger Prozess. Ganz vermeiden lässt es sich ohnehin nicht, will man in Softimage|XSI Propertypages machen, die auf Einstellungen des Users reagieren und je nachdem ob eine Checkbox angehakt ist oder nicht Elemente ein- oder ausgeblendet werden.

### **B.5.1 Die Sicherheitseinstellungen von Java Applets**

Bevor man mit dem Shader-Wizard zu arbeiten beginnt, ist es unbedingt notwendig im Browser Einstellungen zu treffen, da sonst das Applet nicht einwandfrei funktioniert. Das liegt daran, dass Java Applets seit einiger Zeit Sicherheitsbestimmungen unterliegen damit etwaig bössartige Java Applets auf dem System keinen Schaden anrichten können. Seit einiger Zeit laufen Java Applets im sogenannten Sandbox-Modus [JAVA01, 2005, [http://www.unix.org.ua/oreilly/java-ent/security/ch01\\_02.htm](http://www.unix.org.ua/oreilly/java-ent/security/ch01_02.htm)]. Diese Sandbox hat nur Zugriff auf bestimmte Ressourcen im Computer, nämlich auf Bildschirm, Keyboard, Mouse und Speicher. Diese Einschränkung muss gelockert werden, damit das Applet auch die Berechtigung besitzt die Dateien auf die Festplatte zu schreiben. Dieses Verhalten ist von Browser zu Browser verschieden und wurde mit folgenden Browsern getestet:

- Microsoft Internet Explorer
- Mozilla Firefox
- Opera

Der Pfad zum Shader Wizard lautet:

```
\<XSI_Installverzeichnis>\XSISDK\wizards\shader\index.htm
```

#### **B.5.1.1 Opera:**

In der Version 6.05 konnte das Applet trotz installiertem Java Runtime Environment 1.4 nicht starten und gab stattdessen eine Fehlermeldung aus, dass Java nicht installiert sei. Hoffentlich ist dieser Bug bei der aktuellen Operaversion nicht mehr vorhanden.

#### **B.5.1.2 Internet Explorer (Windows 2000)**

Hier muss man die richtigen Einstellungen treffen, um die Sandboxrestriktionen zu lockern. Im Menüpunkt "Extras > Internetoptionen" muss man die Registerkarte "Sicherheitseinstellungen" anwählen. Dort kann man unter "Stufe anpassen" diverse Browserspezifischen Sicherheitsstufen wählen. Man scrollt

bis zum Punkt "Microsoft VM" hinunter. Für Java muss man den Sicherheitsstandard auf "Benutzerdefiniert" einstellen. Darauf erscheint im selben Fenster unten ein Button mit der Aufschrift „Java Einstellungen“. Unter der Registerkarte „Zugriffsrechte bearbeiten“ kommt man dann zu dem gewünschten Fenster, in dem sich die Sicherheitsstufe individuell anpassen lässt. Man kann die Sicherheitsstufen getrennt, also sowohl für zertifizierte als auch nicht zertifizierte Applets anpassen. Der Shader-Wizard ist ein von Verisign zertifiziertes Applet, daher braucht man nur die Einstellungen für zertifizierte Applets einstellen. Auf jeden Fall muss dem Applet der Zugriff auf das Dateisystem ermöglicht werden. Nachdem diese Einstellungen übernommen wurden, lässt sich das Applet starten<sup>9</sup>.

### B.5.1.3 Mozilla Firefox:

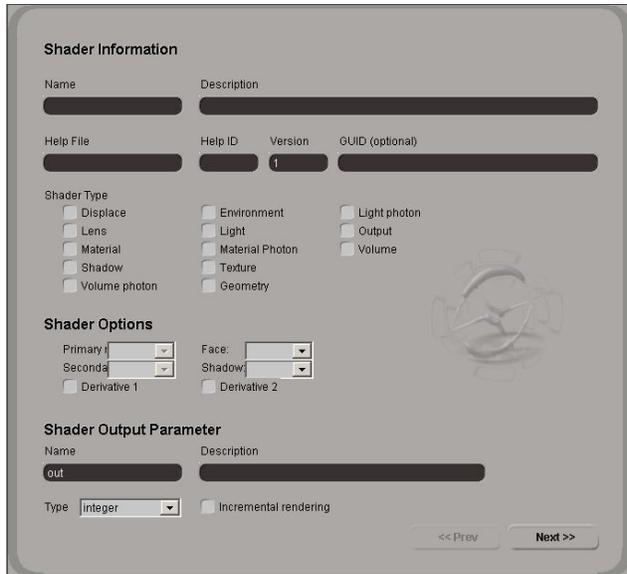
Beim Firefox gilt ähnliches wie für den Internet Explorer. Die Installation des neusten Java-Plugins lässt das Applet zwar starten, doch das Schreiben der Dateien auf die Festplatte funktioniert nicht. Auch hier hilft etwas experimentieren.

---

<sup>9</sup>Anm. d. Autors: Bei mir trat unter Windows 2000 der Fehler auf, dass die Javakonsole die Meldung ausgibt "Netscape Sicherheitsmodell wird nicht mehr unterstützt. Bitte stellen sie auf das Java 2 Sicherheitsmodell um". Das Applet sicherte daraufhin keine Dateien. Ich vermute dahinter eine Inkompatibilität des Programmes mit der installierten Java-Umgebung. Unter dem J2RE Runtime Environment 5.0 von Java läuft das Applet nicht. Abhilfe soll die Installation einer neueren Microsoft VM-Ware für Java schaffen, was ich selbst jedoch nicht gemacht habe. Ich habe die neuere Java Installation entfernt und mit der Windows 2000 Standardinstallation das Applet gestartet. Unter Windows XP lief das Applet ohne Probleme.

## B.5.2 Den Shader im Wizard zusammensetzen

### B.5.2.1 Die Einstellungen im Shader Wizard

The screenshot shows the 'Shader Information' and 'Shader Options' sections of a wizard. The 'Shader Information' section includes fields for Name, Description, Help File, Help ID, Version (set to 1), and GUID (optional). The 'Shader Type' section has a grid of checkboxes for Displace, Lens, Material, Shadow, Volume photon, Environment, Light, Material Photon, Texture, Geometry, Light photon, Output, and Volume. The 'Shader Options' section includes dropdowns for Primary and Secondary, Face, and Shadow, and checkboxes for Derivative 1 and Derivative 2. The 'Shader Output Parameter' section has fields for Name (set to 'out') and Description, and a dropdown for Type (set to 'integer') and an 'Incremental rendering' checkbox. Navigation buttons '<< Prev' and 'Next >>' are at the bottom.

**Abbildung 29:** Die Startseite des Shader-s. Hier muss man die Entscheidung treffen, für welche Shadingzwecke der Shader eingesetzt wird. Spätere Änderungen sind nur mehr durch Editieren der SPDL-Datei möglich.

Die jeweiligen Inputs im Materialnode anhängen. Dies ist ein Sicherheitsmechanismus den Softimage vor versehentlichem Missbrauch bietet. Bitte aber nicht den Shader einschränken. Wenn man weiss, dass ein Shader zwar als Material Shader programmiert wurde, aber konform für die Anforderung anderer Shader ist, dann soll man das auch diesem Shader erlauben für andere Zwecke als dem ursprünglich vorgesehenen zu fungieren. Es ist kein Fehler dem Benutzer die Verwendung des Shaders offen zu lassen, solange er keine unerlaubten Operationen ausführt. Denn wie mittlerweile schon bekannt ist, kommt es ganz auf den Shader darauf an, welche Funktionen er verwenden darf, welche Variablen des States für ihn gelten und welche undefiniert sind. Man wählt die entsprechenden Shader Types aus, von denen man weiß, dass sie der Shader behandeln wird können. Auf Seite 123 wird gezeigt, wie man im C-Code Aufrufe des jeweiligen Shadertypes erkennen kann. Darauf kann der Shader die entsprechende Operationen ausführen.

Der Shader-Wizard wird in einem Webbrowser gestartet. Mit den Vorüberlegungen, welche schon angestellt wurden, kann man den Shader rasch im Wizard aufbauen.

Der Name des Shaders bestimmt seine Bezeichnung, mit der er später auch im Rendertree von XSI zu sehen sein wird. Die "Description" scheint nicht direkt auf, sondern ist im C-Code als Kommentar zu finden.

Danach folgt die Entscheidung für welchen Aufgabentyp der Shader eingesetzt werden soll. Je nachdem welche Shader Types angehakt werden, kann man im Softimage Rendertree den Shader an

Es ist kein Fehler einen Shader auch andere Shading Types berechnen zu lassen, solange er keine Funktionen verwendet, die er in der Funktion eines anderen Shaders nicht verwenden dürfte.

In den Shader Options ist angeben, welche Options gesetzt sein müssen, damit der Shader auch einwandfrei funktioniert.

### **Primary / Secondary: ON / OFF**

Für Material und Lens Shader kann angegeben werden, dass diese zwingend verlangen, dass diese nur einwandfrei funktionieren wenn Primary Rays beziehungsweise Secondary Rays aktiviert oder deaktiviert sind.

Für **Face** und **Shadow** gilt dasselbe, nur dass diese Optionen unabhängig bei allen Shadertypes (nicht nur für Material und Lens) eingestellt werden können. Die möglichen Werte für Face sind: Both, Front oder Back. Shadows können aktiviert oder deaktiviert sein.

**Incremental Rendering** sollte für Shader deaktiviert werden die von den `_init` und `_exit` Routinen Gebrauch machen. Incremental Rendering macht nichts anderes, als bei einer Parameteränderung nicht wieder von vorne zu rendern zu beginnen. Stattdessen wird einfach mit dem geänderten Parameterwert fortgesetzt. Das ist in den seltensten Fällen erwünscht und kein Basissshader von Softimage macht von dieser Technik gebrauch [NAN2, 2003, <http://www.nanomation.co.uk/Programming-102.html>].

Auf der nächsten Seite müssen die Shaderparameter eingestellt werden. Diese Einstellungen betreffen sowohl die C-Dateien als auch die Spdl-Dateien die vom Shader-Wizard erstellt werden. Als erstes muss man dem Parameter eine Bezeichnung geben unter dem er für die Software zugänglich gemacht wird.

`baseColor:`

Im Falle unseres ersten Parameters lautet seine Bezeichnung `basecolor`. Die Description kann auch entfallen. Diese ist lediglich ein Kommentar die in der Headerdatei des Shaders neben der Variablendeklaration eingefügt wird. Der Typ der Variablen ist wieder von Bedeutung. In diesem Falle ist das „color“, da ja einen RGB-Wert in Softimage|XSI eingestellt werden soll. Der Parameter muss auch noch mit Standardwerten versehen werden die er aufweist, sobald man den Shader initialisiert. Hier wurde einen Rot-Ton gewählt mit den Werten  $R = 1.0$ ,  $G = 0.2$ ,  $B = 0.2$ . Der Alphawert lautet 1 – würde der Wert anders lauten, so würde das Pixel trotzdem mit einem Alphawert von 1.0 in RGBA-Bilder abgespeichert werden, was wenig Sinn macht.

Die Checkboxes sind für spezifische Einstellungen gedacht und sollten wenn nicht anders erwünscht so

gelassen werden wie sie sind. In der zweiten Zeile der Checkboxes wird die letzte Checkbox durch den Text von "PersiLayoutTable" verdeckt. Sie lautet "Inspectable" und dieser Wert muss mit dem Editor direkt in der SPDL-Datei wenn gewünscht geändert werden. Die Funktionen der Checkboxes sind folgende [SPDL, 2005, SPDL\_shader6.html.]:

### *Readable*

Definiert ob der Parameter ausgelesen werden kann. Der Defaultwert ist ON.

### *Animatable*

Definiert ob der Wert animiert werden darf. Der Defaultwert ist ON.

### *Texturable*

Definiert ob dem Parameter andere Shader zugewiesen werden können. Der defaultwert ist OFF.

### *Writable*

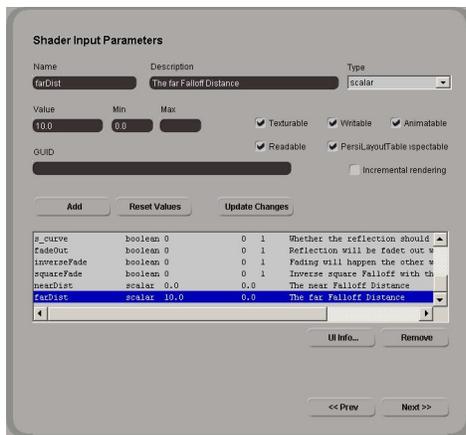
Definiert ob der Parameter von außerhalb verändert werden kann, beispielsweise durch ein Script. Der Defaultwert ist ON

### *PersiLayoutTable*

Definiert ob der veränderte Parameterwert gespeichert werden soll. Default ist ON

### *Inspectable (verdeckt)*

Ob der Parameter auf der Property page angezeigt wird oder nicht. Default ist ON.



**Abbildung 30:** Erstellen der einzelnen Parameter im Shader-Wizard

erste Parameter korrekt adjustiert. Nun fährt man mit den weiteren Parametern fort. Wo nicht anders angegeben braucht man sich um das UI-Info nicht zu kümmern.

Die GUID kann man leer lassen, denn diese wird automatisch vom Wizard erzeugt und in die SPDL eingefügt wenn hier nichts eingetragen wird. Mit diesen Einstellungen kann man den Parameter vorerst mit dem Button „Add“ hinzufügen. Eine Sache muss noch eingestellt werden: Damit niemand mit dem Alphawert herumspielen kann, muss man noch den bascolor-Parameter in der Liste anwählen (was er als erster Parameter automatisch schon ist) und „UI Info“ auswählen. In diesem Bereich kann man noch den genauen User Interface Typ einstellen. Man wählt aus der Liste „RGB“ aus. Somit wurde der

### SpecularColor

RGBA wird auf 1.0 eingestellt und der UI-Type lautet wie die baseColor ebenfalls RGB.

SpecularMode

Der Specular mode ist eine Liste, deren Einträge den Specularmodus spezifizieren. Der Datentyp ist Integer. Der Standardwert lautet 1, der zugelassene Minimalwert ist 1, der maximalwert 2 – schließlich haben wir nur zwei Einträge. Der UI-Type ist „Combo“. Wir fügen folgende Parameter als Combоеinträge hinzu: „Blinn“ und „Phong“

specularDecay

Ein Skalar (Scalar) mit Standardwert 50.0, Minimalwert 0.0 und Maximalwert 200.0.

roughness

Ein Skalar mit Standardwert 0.3, Minimalwert 0.0 und Maximalwert 2.0.

specularRefraction

Ein Skalar mit Standardwert 10.0, Minimalwert 1.0 und Maximalwert 100.0.

reflection

Die Reflexion ist ein color-Wert mit Standardwert R= 0.0, G = 0.0, B = 0.0, A = 0.0. Der UI-Typ ist RGBA.

LumaAmount

Ein Skalar mit Standardwert 1.0, Minimalwert 0.0 und Maximalwert 1.0.

lumaThreshold

Ein Skalar mit Standardwert 0.0, Minimalwert 0.0 und Maximal undefiniert, also keinem Eintrag.

s\_curve

Ein Boolean-Datentyp. Standardwert 0, Minimalwert 0, Maximalwert 1. UI-Typ ist „Checkbox“.

FadeOut

Wie s\_curve

inverseFade

Wie s\_curve

squareFade

Wie s\_curve

nearDist

Ein Skalar, Standardwert 0.0, Minimalwert 0,0, Maximalwert undefiniert

farDist

Ein Skalar, Standardwert 10.0, Minimalwert 0.0, Maximalwert undefiniert.

Damit ist die Definition der Parameter abgeschlossen. Die nächste Seite lässt das Layout der Property Page für Softimage definieren. Dieser Schritt ist wesentlich weniger Arbeit. Der Shader soll in zwei Bereiche geteilt sein welche durch Reiter (Tabs) getrennt sind.

Vorerst werden all Einträge von der linken Seite entfernt. Man muss keine Angst haben dabei versehentlich Parameter zu löschen. Wenn die Liste leer ist, fügen wir einen Tab mit der Bezeichnung „Shading“ hinzu. Der Shader-Wizard markiert diesen Parameter automatisch und die Groups „Base Color“ und „Specular Model“ werden dem Tab hinzugefügt.

Dann kommt das Tab „Reflections“ mit den Groups „Reflection Color“, „Luminance“ und „Falloff“ hinzu.

Nun müssen dem Layout die richtigen Variablen zugewiesen werden. Dazu wird immer das jeweilige Tab beziehungsweise die Group markiert. Danach wählt man von der rechten Seite einen Parameter aus und fügt diesen über „Add Parameter“ hinzu. Hier die fertige Layoutstruktur:

#### Shading

- Base Color

  - baseColor

- Specular Model

  - specularMode

  - specularDecay

  - roughness

  - specularRefraction

#### Reflections

- Reflection Color

  - reflection

- Luminance

  - lumaAmount

  - s\_curve

  - lumaThreshold

- Falloff

  - fadeOut

  - inverseFade (c)

  - squareFade

  - nearDist

  - farDist

Die Parameter mit dem eingeklammerten „c“ werden noch einmal markiert und mit dem Häkchen „continue“ versehen. Dies bewirkt, dass dieser und der nächste Parameter in der Propertypage nebeneinander stehen.

Setzt man auf die nächste Seite fort, sieht man im Fenster schon die SPDL-Datei die der Wizard erzeugen wird. Geübte Programmierer können hier rasch noch den Code der SPDL-Datei überprüfen. Falls man

einen Parameter falsch definiert hat, kann man jetzt noch immer zurück gehen und ihn entsprechend korrigieren. Ist man zufrieden, braucht man nur mehr den Zielpfad angeben, in welchen die Dateien geschrieben werden sollen.

#### B.5.2.2 Fehler des Shader-Wizards ausbessern

Bei der Definition der UI-Types "Check", "Radio" oder "Combo" verabsäumt der Shader Wizard eine Zeile mit einem Strichpunkt abzuschließen, der an dieser Stelle erforderlich gewesen wäre. außerdem wird ungeachtet des gewählten welchen UI-Typs immer "Combo" eingestellt. In Kapitel B.10 wird ein Tool gezeigt welches die Syntax von SPDL-Dateien überprüft und die Zeilennummer der fehlerhaften Zeilen ausgibt.

## B.6 C-Coden, Teil 2 – Learning by doing

*In dieser Diplomarbeit wird der Beispiel-Shader in einem Durchgang geschrieben. Normalerweise wird dieser Prozess in kleinere Schritte aufgeteilt: Man programmiert schrittweise die Funktionalitäten hinzu, für die man in der Shader-Propertypage Eingabeparameter vorgesehen hat. Nach jedem neuen programmierten "Feature" testet man dieses in Softimage. Bei diesem Vorgehen kann man Fehler im Code frühzeitig erkennen und ausbessern. Der Leser halte sich diesen Umstand vor Augen und wird um Verständnis gebeten, da aus Gründen der Wissenschaftlichkeit dieser Arbeit diese Vorgehensweise im Text nicht berücksichtigt wird.*

Nachdem alle Dateien im Shader Wizard erstellt wurden kommt noch eine kurze Erläuterung. Interessant sind folgende Dateien:

#### **lumaReflect.spdl:**

In SPDL-Dateien ist der Aufbau für Propertypages (PPG's) unter SoftimageXSI festgelegt. Darin steht welche Parameter vorkommen und sie verweist auf die DLL-Datei in der der Funktionscode für die Propertypage steht. Im Falle von Shader unterscheiden sich die Propertypages von den herkömmlichen PPG's weshalb sie auch in der Dokumentation von SoftimageXSI ein eigenes Kapitel einnehmen [SPDL, 2005, \_start.htm].

### **lumaReflect.h:**

In dieser Datei wurde eine Datenstruktur mit allen Parametern angelegt, die zuvor im Shader-Wizard erstellt wurde. Eine Spezialität ist hinzugekommen: Die Variablennamen beginnen nun alle mit "m\_". Das liegt daran, dass die Werte der PPG aus Softimage in den Funktionscode noch einmal mit der Funktion `mi_eval()` importiert werden müssen. Die Variablen werden mit dem dem gewohnten Namen deklariert und der Wert der "m\_" dekorierten Variablen zugewiesen. Die genaue Funktionsweise von `mi_eval()` ist auf Seite 99 dargestellt.

### **lumaReflect\_dll.cpp:**

Der Shader wird in eine DLL kompiliert. Insgesamt besteht ein Shader aus 4 Funktionen die zum Einsatz kommen können. Diese Funktionen kümmern sich um den Einstieg in die DLL-Datei wenn der Shader aufgerufen wird [DLLEX, 2000, [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/msmod\\_20.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/msmod_20.asp)]. Für einen Shaderwriter hat dieser Inhalt keine weitere Bedeutung und muss nie verändert werden.

### **LumaReflect.cpp:**

Dort passiert alles, was mit dem eigentlichen Shader zu tun hat. Hier finden sich die vier Funktionen die im Laufe des Renderns aufgerufen werden. Diese werden mit `DLL_EXPORT` angeführt, was dem Compiler zu erkennen gibt, dass er die Funktion in eine DLL-Datei exportieren soll. Der Ganze DLL-Mechanismus ist zu vertrakt um ihn in dieser Arbeit offenzulegen, daher begnügen wir uns einfach mit der Tatsache, dass die vom Shader-Wizard erstellten Dateien sich mit dem Visual C-Compiler ohne Fehlermeldungen kompilieren lassen. Die vier Funktionen lauten folgendermaßen:

```
DLL_EXPORT int lumaReflect_version(void)
```

Diese Funktion liefert die Version des Shaders zurück. Ein Mechanismus der User warnt, falls ein registrierter Shader auf eine DLL-Datei des Shaders mit einer unterschiedlichen Version zugreifen will. Das ist der Fall wenn man den Shader ein weiteres Mal mit veränderter Versionsnummer kompiliert, aber noch die alte Version des Shaders in XSI registriert ist. Mental Ray versucht dann trotzdem mit dem Shader zu rendern, aber es ist keine Garantie auf die Funktionstüchtigkeit mehr gegeben.

Allerdings sollen Änderungen des Shaders so gemacht werden, dass sich am Interface nichts ändert. Das heisst, es sollen nur mehr Codeoptimierungen vorgenommen werden. Dadurch verändert sich das Verhalten des Shaders nicht sichtbar.

```
DLLEXPORT void lumaReflect_init(  
    miState * state,  
    lumaReflect_params * in_pParams,  
    miBoolean *inst_req)
```

```
DLLEXPORT void lumaReflect_exit(  
    miState * state,  
    lumaReflect_params * in_pParams)
```

Die Shader-init und -exit Routinen sind selten in Gebrauch. Diese Funktionen dienen der Initialisierung des Shaders. Zum Beispiel kann man Speicher reservieren und mit Pseudozufallszahlen füllen um daraus Prozedurale Texturen zu erstellen. Eine weitere Möglichkeit ist auch dort static-Variablen anzulegen um Information von Sample zu Sample weiterzugeben. Es gibt sonst nämlich keinen anderen Mechanismus im Funktionscode.

Das Hauptinteresse gilt der Funktion mit dem Namen des Shaders, nämlich

```
DLLEXPORT miBoolean lumaReflect  
(  
    miColor * out_pResult,  
    miState * state,  
    lumaReflect_params * in_pParams  
)
```

In dieser Funktion findet sich der gesamte Code für den Shader wieder. Für fast alle Shader reicht es, diese Funktion mit Anweisungen zu füllen. Nur in seltenen Fällen ist es notwendig auch Anweisungen in die Shader-Init und -exit Routinen zu schreiben. Im Kommenden wird in die Funktion `lumaReflect()` der Funktionscode geschrieben.

```
miColor * out_pResult
```

Die Funktion erhält als Argumente einen Pointer auf das Ergebnis, in unserem Fall ist

der Ergebnistyp ein RGBA Wert, weshalb auch `oup_pResult` ein Pointer vom Typ `miColor` ist.

```
miState * state
```

Das zweite Argument ist ein Pointer auf den aktuellen State und das dritte Argument ein Pointer auf die Datenstruktur mit den eingestellten Parameterwerten des Shaders. Der Grund warum Pointer übergeben werden liegt in der schnelleren Funktionsweise von Pointern. Dadurch wird unnötiges kopieren von Variablen erspart. Die `state`-Variable soll immer unbedingt auch "`state`" lauten. Einige Funktion der Mental Ray Bibliothek verlassen sich nämlich darauf, dass diese Variable exakt so heisst. Hat sie einen anderen Namen, ist die Funktionstüchtigkeit dieser Funktionen nicht mehr gegeben.

```
lumaReflect_params * in_pParams
```

Die letzte Variable ist ein Pointer auf die Struktur welche alle Werte der Propertytype des Shaders enthält. Man darf allerdings nicht einfach per Zeigerzugriff auf die Parameter zugreifen sondern muss sich der Funktion "`mi_eval_*`" bedienen. Das wird im nächsten Kapitel noch gezeigt werden.

## B.6.1 Der Shadercode für `lumaReflect`

Es ist nun soweit dass alle Vorarbeiten abgeschlossen sind. Nun muss der Funktionscode implementiert werden. Nachdem die Anforderungen an den Shader ja schon ein paar Seiten früher aufgelistet wurden, kann man diesen Anforderungen getreu programmieren. Der Shader wird in der Microsoft Visual C/C++ Entwicklungsumgebung geschrieben. Diese Entwicklungsumgebung ist altbewährt und einfach zu bedienen.

### B.6.1.1 Erster Teil – Variablen, Shading und Glanzlicher

Der erste Schritt ist, dass die benötigten Variablen erst einmal deklariert werden. Der Code hier ist nicht so kommentiert wie der in der beiliegenden CD-Rom da er ohnehin mit diesem Text schon auskommentiert wird. Die Stelle an der man schreiben ansetzt ist nach den öffnenden geschwungen Klammern der `lumaReflect`-Funktion (Deutlich zu erkennen durch die Anweisung: "`put your coede here`"). Hier ist die erste Hälfte des Shadercodes abgedruckt und wird im kommenden erläutert. Der zweite Teil wird weiter unten erörtert.

```

DLLEXPORT miBoolean lumaReflect
(
    miColor * out_pResult,
    miState * state,
    lumaReflect_params * in_pParams
)
{
    /* put your code here */

    miColor    *baseColor, *specularColor, *reflection;

    /* Spcular eigenschaften */
    miInteger    specularMode;
    miScalar    specularDecay;
    miScalar    roughness;
    miScalar    specularRefraction;

    miColor    reflectResult;
    miScalar    lumaAmount;
    miScalar    lumaThreshold;
    miBoolean    s_curve;
    miBoolean    fadeOut;
    miBoolean    inverseFade;
    miBoolean    squareFade;
    miScalar    nearDist;
    miScalar    farDist;

    miScalar    specFactor;
    int        samples;
    int        light_n;    /* Anzahl der Lichter */
    miTag    *light;    /* Pointer zum 1. Licht im glob. Lichtarray (mi_query()) */
    miColor    lightResult; /* Resultat der Lichtberechnung */
    miColor    lightSum;    /* Variable zur Summation der Lichter bei
                           mehreren Lichtsamples von "mi_sample_light()" */

    miVector    dir;    /* Ray-Richtung des Lichtes */
    miVector    reflectionDir;

    miScalar    dot_nl;    /* Dot-Produkt von SurfaceNormal und dir */

    baseColor = mi_eval_color(&in_pParams->m_baseColor);
    specularColor = mi_eval_color(&in_pParams->m_specularColor);
    reflection = mi_eval_color(&in_pParams->m_reflection);

    //wir besorgen uns die globale Lichtliste
    mi_query(miQ_GLOBAL_LIGHTS, state, miNULLTAG, &light_n);
    if(light_n >= 1)
        mi_query(miQ_GLOBAL_LIGHTS, state, miNULLTAG, &light);
    else
        /* Keine Lichtquelle, daher alles schwarz */
        /* Das muss noch geändert werden, weil sonst auch nix Reflectiert wird! */
        {
            out_pResult->r = out_pResult->g = out_pResult->b = 0.0;
            out_pResult->a = 1.0;
            return miTRUE;
        }

    /* nun über die das Objekt beleuchtenden Lichtquellen loopen */
    for(int i = 0; i < light_n; i++)
    {
        lightSum.r = lightSum.g = lightSum.b = 0.0;
        samples = 0;
        while(mi_sample_light(&lightResult, &dir, &dot_nl, state, light[i], &samples))
        {
            lightSum.r += dot_nl * baseColor->r * lightResult.r;
            lightSum.g += dot_nl * baseColor->g * lightResult.g;
            lightSum.b += dot_nl * baseColor->b * lightResult.b;

            /* Berechnung des Highlights mittels */
            /* der integrierten mi-Funktionen */
            switch(specularMode = *mi_eval_integer(&in_pParams->m_specularMode))

```

```

    {
        //Case 0: Das Blinn-Modell
        case 0:
            specularRefraction = *mi_eval_scalar(&in_pParams->m_specularRefraction);
            roughness = *mi_eval_scalar(&in_pParams->m_roughness);
            specFactor = mi_blinn_specular(&state->dir, &dir, &state->normal ,
roughness,
                                     specularRefraction);

            break;

        //Case 1: Das Phong-Modell
        case 1:
            specularDecay = *mi_eval_scalar(&in_pParams->m_specularDecay);
            specFactor = mi_phong_specular(specularDecay, state, &dir);
            break;

        //case 2: Kein Specular
        case 2:
            specFactor = 0.0;
        default:
            break;
    }

    if(specFactor > 0)
    {
        lightSum.r += specFactor* lightResult.r * specularColor->r;
        lightSum.g += specFactor* lightResult.g * specularColor->g;
        lightSum.b += specFactor* lightResult.b * specularColor->b;
    }
}
/* Dividieren des Ergebnisses durch die Anzahl der gewonnenen Samples */
if(samples)
{
    out_pResult->r += lightSum.r / samples;
    out_pResult->g += lightSum.g / samples;
    out_pResult->b += lightSum.b / samples;
}
}
// ...Fortsetzung weiter unten

```

Ganz zu Beginn werden die Variablen, welche die Werte für die Farbgreier des Shaders beinhalten sollen, definiert. Da die Werte zum Zeitpunkt des Renderns schon irgendwo im Speicher liegen, braucht man nur mit einem Pointer auf diese zu zeigen um sie zu bearbeiten.

Die Skalar- und Integervariablen werden nicht als Pointer sondern als richtige Wertvariablen angelegt.

Die Variablen sollten von einem der per typedef definierten Datentypen sein (Siehe Kapitel B.1.1.1 auf Seite 69). Eine Integervariable wird demnach als `miInteger` bezeichnet, eine Fließkommavariable als `miScalar` und so weiter. Dadurch wird sichergestellt, dass der Code portabel bleibt, wollte man ihn auf einem Unix/Linux-system erneut kompilieren.

```

miColor      *baseColor, *specularColor, *reflection;

/* Spcular eigenschaften */
miInteger    specularMode;
miScalar     specularDecay;
miScalar     roughness;
miScalar     specularRefraction;

miColor      reflectResult;
miScalar     lumaAmount;
miScalar     lumaThreshold;

```

```

miBoolean    s_curve;
miBoolean    fadeOut;
miBoolean    inverseFade;
miBoolean    squareFade;
miScalar     nearDist;
miScalar     farDist;

miScalar     specFactor;
int          samples;
int          light_n;      /* Anzahl der Lichter */
miTag        *light;      /* Pointer zum ersten Licht im globalen
                           Lichtarray (mi_query()) */
miColor      lightResult; /* Resultat der Lichtberechnung */
miColor      lightSum;    /* Variable zur Summation der Lichter bei
                           /* mehreren Lichtsamples von "mi_sample_light()" */
miVector     dir;         /* Ray-Richtung des Lichtes */
miVector     reflectionDir;

miScalar     dot_nl;      /* Dot-Produkt von SurfaceNormal und dir */

```

Soweit die Deklaration der Funktionsvariablen. Man sieht, dass hier mehrere Variablen deklariert wurden als der Shader Parameter hat. Diese zusätzlichen Variablen werden für zusätzliche Berechnungen des Shaders benötigt. Auf den nächsten Seiten wird auf diesen Umstand näher eingegangen.

Erstmals werden die Werte für Farbe, Specular und Reflexion geladen. Da diese ohnehin gebraucht werden, zahlt es sich aus, sie gleich zu Beginn zuzuweisen. Der Grund warum Mental Ray nicht automatisch alle Parameter des Shaders ausgewertet ist, dass Parameter oft gar nicht für Berechnungen benötigt werden. Beispielsweise gibt es Toolshader, die, je nachdem ob ein Ray auf die Vorder- oder die Rückseite eines Polygons auftrifft, verschiedene Shader aufrufen. Es wäre unsinnig beide Eingänge auszuwerten und möglicherweise einen aufwändigen Shader aufzurufen, dessen Ergebnis letztendlich verworfen wird. Stattdessen wird geprüft ob nun der Strahl die Vorder oder die Rückseite getroffen hat und erst *dann* wird der entsprechende Parameter ausgewertet. Diese Technik nennt sich Lazy Evaluation [MEY]. Den Mechanismus der Auswertung betreibt die Funktion `mi_eval_*()`. Diese Funktion kommt in mehreren Ausführungen:

```

miBoolean *mi_eval_boolean (miBoolean *param)
miInteger *mi_eval_integer (miInteger *param)
miScalar *mi_eval_scalar (miScalar *param)
miVector *mi_eval_vector (miVector *param)
miScalar *mi_eval_transform (miScalar *param)
miColor *mi_eval_color (miColor *param)
miTag *mi_eval_tag (miTag *param)

```

Je nachdem von welchem Typ die Variable ist, muss die jeweilige Funktion verwendet werden. Vorsicht sei beim Zuweisen des Ergebnisses geboten: Die Funktion liefert einen Pointer als Rückgabewert. Will man nicht den Pointer, sondern den Inhalt des Ergebnisses haben, so muss der Rückgabewert mit dem Zeigerzugriffoperator ("`*`") zugewiesen werden. Hier ein Beispiel:

```
miColor myColor, *myColorPtr;
myColorPtr = mi_eval_color(&in_pParams->baseColor);
myColor = *mi_eval_color(&in_pParams->basColor);
```

Die Aufrufe um den Inhalt der Shaderparameter zu bekommen, lauten demnach

```
baseColor = mi_eval_color(&in_pParams->m_baseColor);
specularColor = mi_eval_color(&in_pParams->m_specularColor);
reflection = mi_eval_color(&in_pParams->m_reflection);
```

Die nächste Aufgabe des Shaders ist, sich eine Liste mit allen Lichtern der Szene zu besorgen. Dazu wird die Funktion `mi_query()` benötigt. Über diese Funktion kann man auf die vielen Elemente der Szene zugreifen und beinahe jede Information besorgen, die man im State nicht findet. Man übergibt dieser Funktion den Query-Code, den State-Pointer, ein Tag (also einen Netzwerkunabhängigen "Mental Ray-Pointer") und einen Pointer für das Ergebnis. Manche Abfragen benötigen kein Tag. In diesem Fall übergibt man der Funktion `mi_NULLTAG`. Im vorliegenden Anwendungsfall müssen zwei Aufrufe mit `mi_query()` getätigt werden. Der erste Aufruf gibt die Anzahl der Lichter, der zweite Aufruf einen Pointer auf das erste Licht. Die Lichter stehen im Speicher hintereinander. Der `mi_query()`-Aufruf liefert lediglich die Adresse des ersten Lichtes. Um auf die weiteren Lichter zuzugreifen braucht man lediglich den Pointer zu inkrementieren (auf keinen Fall den Inhalt!) oder man greift mit dem Array-Zugriffoperator ("`[]`") auf das gewünschte Licht zu. Vorsicht ist dabei geboten, nicht über das Ziel hinaus zu schießen und auf Arrayelemente zuzugreifen, die gar nicht mehr auf ein Licht verweisen. In C/C++ ist dies bekannterweise möglich und führt im schlimmsten Fall zum Absturz, wenn auf fremde Speicheradressen zugegriffen wird. Deswegen wird auch überprüft, ob mindestens eine Lichtquelle vorhanden ist. Erst danach wird nach der Adresse des ersten Lichtes gefragt.

Als nächstes wird durch den Funktionsaufruf von `mi_sample_light()` die Beleuchtung aller Lichtquellen am gegebenen Intersection-Punkt berechnet.

```
miBoolean mi_sample_light(
    miColor      *result, /* Der berechnete Lichtwert*/
    miVector     *dir,    /* erhält die Lichtrichtung */
    miScalar     *dot_n1, /* erhält das Ergebnis des Skalarprodukts */
    miState      *state,  /* Der State */
    miTag        light_inst, /* Ein gültiges Licht-Tag */
    miInteger    *samples) /* Anzahl der Samples */
```

Diese Funktion wird sowohl für Pointlights als auch für Arealights verwendet. Die Variablen `*result`, `*dir` und `*dot_n1` erhalten das Ergebnis des Funktionsaufrufes. Die Variable `*dir` enthält den Richtungsvektor von der Lichtquelle zum Intersectionpoint und in `*dot_n1` wird das Skalarprodukt

geschrieben. Die `light_inst` ist das Licht-Tag und `samples` eine Zählervariable für Arealights, wenn `mi_sample_light()` mehrmals hintereinander aufgerufen wird. Deswegen wird `counter` vor dem ersten Aufruf von `mi_sample_light()` mit 0 initialisiert.

Die Funktion wird als "while"-Schleife ausgeführt. Sie gibt TRUE zurück, solange nicht alle Lichtamples berechnet wurden. Bei Pointlights ist das ein einziger Aufruf. Innerhalb der Schleife wird die Beleuchtung an dem gegebenen Punkt berechnet. Das Standardverfahren verwendet das Gesetz von Lambert. Kurz: Die Helligkeit am gegebenen Punkt ist das Ergebnis der Multiplikation des Skalarproduktes von Oberflächennormalen und Lichtrichtung mit der Materialfarbe und der Lichthelligkeit an diesem Punkt. Das Ergebnis wird in der Variablen `lightSum` summiert falls dass die Funktion mehrfach aufgerufen wird.

Als nächstes ist die Berechnung des Glanzlichtes an der Reihe. Im Beispiel-Shader kann man zwischen den 3 Modi "Blinn", "Phong" oder "kein Glanzlicht" auswählen. Je nachdem welche Einstellung in der PPG des Shaders getroffen wurde, wird die korrekte Berechnung durchgeführt. Glücklicherweise gibt es auch für die Glanzlichtberechnung bereits Mental Ray-Funktionen, denen man nur mehr die benötigten Variablen zu übergeben braucht. Die Funktion `mi_blinn_specular()` hat als Funktionsargumente die Richtung des aktuellen Strahles und die Richtung des Lichtstrahles, welche von `mi_sample_light()` mitgegeben wurde. Als letzten Vektor benötigt man den Vektor der Oberflächennormalen, der in `state->normal` steht. Hier muss man wieder beachten, die Adressen der verlangten Variablen an die Funktion zu übergeben. Die Parameter `roughness` und `specularRefraction` definieren das Aussehen des Glanzlichtes. Das Ergebnis ist eine Zahl, welche die Intensität des Glanzlichtes an der gegebenen Stelle beschreibt.

Beim Phong-Highlight verhält es sich ähnlich. Auch hier wird ein Faktor der Intensität des Glanzlichtes zurückgegeben. Allerdings wird das Glanzlicht von Phong nur durch eine Eigenschaft, dem `specularDecay` beschrieben. So benötigt der Aufruf von `mi_phong_specular()` nur als Argument den `specularDecay`, den State und die Lichtrichtung. Hier noch einmal die Funktionsdefinitionen:

```
miScalar mi_phong_specular(
    miScalar    spec_exp,
    miState     *state,
    miVector    *dir);

miScalar mi_blinn_specular(
    miVector    *dir_in,
    miVector    *dir_out,
```

```

miVector      *normal,
miScalar      roughness,
miScalar      ior);

```

Der Specularwert berechnet sich ähnlich wie die Beleuchtung an der Oberfläche, der Specular aus Specularintensität mal Lichtfarbe mal Specularfarbe. Die Rechnung ist der Berechnung der Beleuchtungsintensität auf der Oberfläche sehr ähnlich. Das Ergebnis der Specularberechnung wird ebenfalls zur Lichtsumme addiert.

Nachdem alle Lichtsamples ermittelt wurden, wird die Lichtsumme durch die Anzahl der Samples dividiert und dem Pointer mit dem Ergebnis der Lichtberechnung, hinzuaddiert. Danach wird in der umschließenden for-Schleife entweder die nächste Lichtquelle berechnet, oder aus der Schleife ausgestiegen, falls keine weiteren Lichter mehr vorhanden sind. Der erste Teil der Berechnung des Shaders ist somit abgeschlossen. Der zweite Teil wird etwas komplexer. Hier werden die Reflexionen berechnet. Verkomplizierend ist, dass durch die zusätzlichen Parameter der Code unübersichtlich wird.

### B.6.1.2 Zweiter Teil – Reflexionen, Falloffs und Gammakorrekturen

```

// ...lumaReflect forts.

if(reflection->r != 0.0 && reflection->g != 0.0 && reflection->b != 0.0
    && reflection->a != 0.0)
{
    /* Wenn wir tatsächlich Reflexionen haben,          */
    /* dann müssen die restlichen notwendigen          */
    /* Parameter ausgewertet werden!                  */
    /* Eine weitere Überprüfung ist, ob die maximale Raydepth schon */
    /* erreicht wurde. Es ist zwar erlaubt, dass Shader die Einstellung */
    /* der Options umgehen, soll aber vermieden werden */
    if (
        state->reflection_level < state->options->reflection_depth &&
        state->reflection_level + state->refraction_level <
            state->options->trace_depth)
    {
        lumaAmount = *mi_eval_scalar(&in_pParams->m_lumaAmount);
        fadeOut = *mi_eval_boolean(&in_pParams->m_fadeOut); /* Wenn die Reflexion
ausfaden soll */
        inverseFade = *mi_eval_boolean(&in_pParams->m_inverseFade);
        mi_reflection_dir(&dir, state);

        //Wenn die Reflexion auch ein Objekt
        if(ok = mi_trace_reflection(&reflectResult, state, &dir))
        {
            if(fadeOut)
            {
                nearDist = *mi_eval_scalar(&in_pParams->m_nearDist);
                farDist = *mi_eval_scalar(&in_pParams->m_farDist);
                reflectionRayLength = state->child->dist;

                falloffRange = farDist - nearDist;
                if(reflectionRayLength <= nearDist)
                {
                    inverseFade ? reflectionDamp = 0.0 : reflectionDamp = 1.0;
                }
            }
        }
    }
}

```

```

        }
        else if (reflectionRayLength >= farDist)
        {
            inverseFade ? reflectionDamp = 1.0 : reflectionDamp = 0.0;
        }
        else
        {
            // linearer Falloff der Reflexion
            if(inverseFade)
            {
                reflectionDamp = (float)(reflectionRayLength-nearDist)/
falloffRange;
            }
            else
            {
                reflectionDamp = 1-(float)(reflectionRayLength-nearDist)/
falloffRange;
            }
        }
        if(*mi_eval_boolean(&in_pParams->m_squareFade))
            reflectionDamp = sqrt(reflectionDamp);
        }
        else
        {
            reflectionDamp = 1.0;
        }
    }
    if(!(ok && fadeOut && inverseFade) || (!ok && !fadeOut))
    {
        mi_trace_environment(&reflectResult, state, &dir);
    }

    rlf = reflectResult.r*0.299f + reflectResult.g*0.587f + reflectResult.b*0.114f;
    lumaThreshold = *mi_eval_scalar(&in_pParams->m_lumaThreshold);
    if(lumaThreshold > 0.0)
    {
        rlf = (rlf-lumaThreshold)/(1.0f-lumaThreshold);
        if(rlf<0)
            rlf = 0.0;
    }

    s_curve = *mi_eval_boolean(&in_pParams->m_s_curve);
    if(s_curve)
    {
        if(rlf > 1 )
            rlf = 1;
        rlf = 3*rlf*rlf - 2*rlf*rlf*rlf;
    }

    out_pResult->r = (out_pResult->r*(1-reflection->r)+
reflectResult.r*reflection->r)*(1-lumaAmount) + lumaAmount*(out_pResult->r +
reflectResult.r*reflection->r*rlf*reflectionDamp);
    out_pResult->g = (out_pResult->g*(1-reflection->g)+
reflectResult.g*reflection->g)*(1-lumaAmount) + lumaAmount*(out_pResult->g +
reflectResult.g*reflection->g*rlf*reflectionDamp);
    out_pResult->b = (out_pResult->b*(1-reflection->b)+
reflectResult.b*reflection->b)*(1-lumaAmount) + lumaAmount*(out_pResult->b +
reflectResult.b*reflection->b*rlf*reflectionDamp);
    out_pResult->a = (out_pResult->a*(1-reflection->a)+
reflectResult.a*reflection->a)*(1-lumaAmount) + lumaAmount*(out_pResult->a +
reflectResult.a*reflection->a*rlf*reflectionDamp);

    }
}
return miTRUE;
}

//Ende der Fkt. LumaReflect()

```

Der zweite Teil beschäftigt sich ausschließlich mit den Reflexionen des Shaders. Bevor Reflexionsstrahlen in die Szene geschossen werden, wird noch überprüft ob eventuell eine Chance bestehen die Reflexionsberechnung zu übergehen. Dazu wird nachgesehen, ob die Reflexionsintensität für RGBA jeweils 0.0 lautet. In diesem Fall ist keine Reflexion zu erwarten. Wenn doch, wird festgestellt, ob die maximale Raydepth oder die Raydepth für Reflections Rays erreicht wurde. Dann wird auch die Reflexion eingespart.

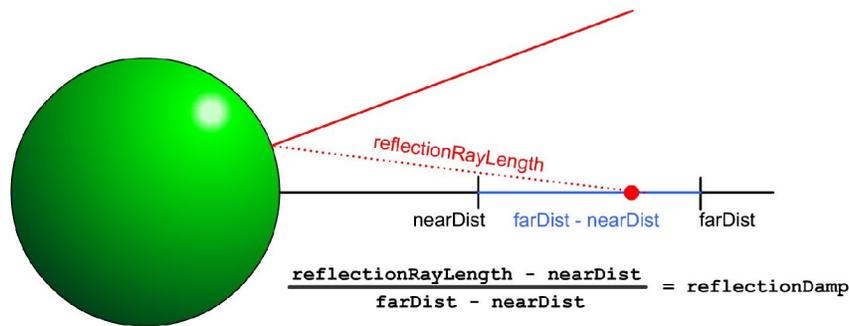
Ist auch dieser Test gelungen muss die Reflexion durchgeführt werden. Dazu wird auf jeden Fall der Parameter `lumaAmount` ausgewertet.

Die Funktion `mi_reflecion_dir()` berechnet den reflektierten Ray an der Intersection. Die neue Richtung wird in `dir` geschrieben und danach die Reflexion durchgeführt.

Die Funktion, welche die Reflexion berechnet, lautet `mi_trace_reflection()`. Es gibt eine ganze Reihe von trace-Funktionen für verschiedenste Zwecke. Die Funktion `mi_trace_reflection()` wird mit Pointern auf das Reflexionsergebnis, den State und die neue Reflexionsrichtung versorgt. Wenn die Funktion den Tracevorgang erfolgreich beenden konnte, gibt sie den den Booleanwert `TRUE` zurück. Wenn nicht, verließ der Strahl die Szene ohne etwas zu treffen. Der Ergebnistyp ist dann `FALSE`. Das Ergebnis wird der Variablen `ok` zugewiesen, damit man später auf einen Misserfolg der Reflexion reagieren kann und den Environment Shader als letzte Alternative aufrufen kann.

War der Aufruf von `mi_trace_reflection()` erfolgreich, wird der Status von `fadeOut` abgefragt. Der FadeOut-Mechanismus dient dazu, Reflexionen über die Distanz ausfaden zu lassen. Einerseits kann man so lokale Reflexionen erzeugen, andererseits ist es möglich, mit `inverseFade` genau das Gegenteil zu erreichen und Reflexionen erst ab einer gewissen Distanz zum Objekt anzeigen zu lassen. Sollte die Option aktiviert sein, dann wird linear von den Distanzen `nearDist` bis `farDist` übergeblendet. Die Variable für die Dämpfung heisst `reflectionDamp`.

Wie man in der Abbildung gut erkennen kann, bewegt sich der Wertebereich von `ReflectionDamp` mit dieser Formel zwischen 0 und 1, wenn die `ReflectionRayLength` zwischen `nearDist` und `farDist` liegt. Je nachdem welches fade-Modell man selektiert hat, wird `reflectionDamp` von 1 nach 0 oder von 0 nach 1 gefadet.



**Abbildung 31:** Berechnung des Falloff unter Berücksichtigung der vorderen und der hinteren Falloffgrenze

Wurde `fadeOut` nicht angehakt, so findet keine Abschwächung der Reflexion über die Distanz statt und der Wert lautet immer 1.

```
if((!ok && fadeOut && inverseFade) || (!ok && !fadeOut))
{
    mi_trace_environment(&reflectResult, state, &dir);
}
```

Hier wird im Falle eines Scheiterns von `mi_trace_reflection()` versucht, einen eventuell vorhandenen Environment Shader aufzurufen. Dieser Aufruf soll allerdings nur stattfinden, wenn folgende Situationen herrschen:

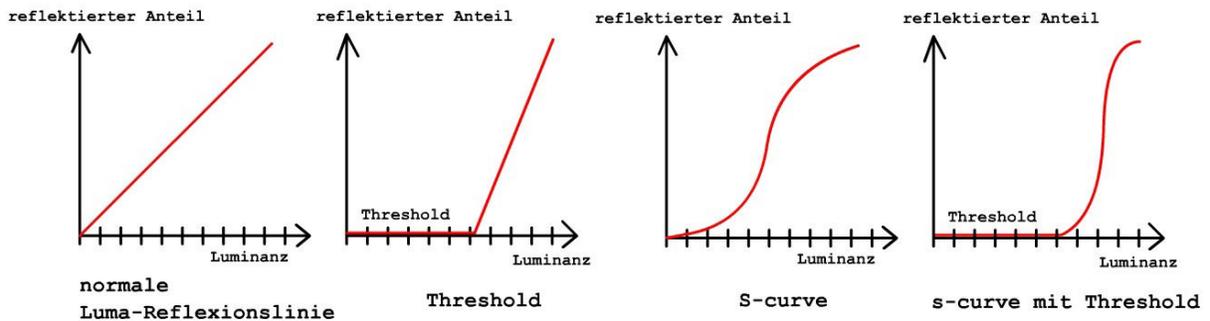
- `mi_trace_reflection()` brachte kein Ergebnis und es wird invers ausgefadet, das heisst die Umgebung ist sichtbar
- `mi_trace_reflection()` brachte kein Ergebnis und es wird nicht ausgefadet

```
r1f = reflectResult.r*0.299f + reflectResult.g*0.587f + reflectResult.b*0.114f;
lumaThreshold = *mi_eval_scalar(&in_pParams->m_lumaThreshold);
if(lumaThreshold > 0.0)
{
    r1f = (r1f-lumaThreshold)/(1.0f-lumaThreshold);
    if(r1f<0)
        r1f = 0.0;
}
```

Was diesen Shader von anderen Reflexionsshader unterscheidet ist, dass die Luminanz der Reflexion berechnet wird. Dieser Wert bestimmt mit welchem Anteil Reflexion der Materialfarbe zugemischt wird. Die Berechnung der Luminanz erfolgt nach der Gewichtung der Luminanz für Fernsehsignale [vgl. PROVID s. 49]. Diese Gewichtung besagt, dass ein gesättigtes Blau rund 11%, ein gesättigtes Rot 30% und ein gesättigtes Grün 59% der Luminanz ausmacht. Nach diesem Modell wird auch der Luminanzfaktor des `lumaReflect`-Shaders gewichtet.

Der Threshold ist eine weitere interessante Funktionalität. In erhellten Räumen fallen nur extrem helle Reflexionen auf Oberflächen auf, da die dunkleren von der Umgebung mehr oder weniger verschluckt werden. Diesem Phänomen wird mit dem Lumathreshold Rechnung getragen. Der Lumathreshold schwächt Reflexionen um den eingestellten Luminanzwert ab und skaliert die Kurve.

Die S-Curve verändert die Luminanz noch einmal. Dunkle Anteile werden weiter abgeschwächt, hellere Anteile werden verstärkt. Der Bereich der S-Curve gilt für Werte zwischen 0.0 und 1.0. Die Funktion dafür lautet  $3x^2-2x^3$ .



*Abbildung 32: Graphen verschiedener Einstellungen für Threshold kombiniert mit s-curve*

Den letzte Teil bildet das Zusammenfügen der Materialfarbe mit der Reflexion. Der Grund für die Rechnung liegt darin, dass auch das herkömmliche Reflexionsmodell integriert ist – mit dem `lumaAmount` kann zwischen dem alten und dem neuen Modell fließend übergeblendet werden. Somit hat man auch das alte Reflexionsmodell zur Hand, wenn zum Beispiel ein Spiegel gerendert werden soll.

Wie man unschwer erkennen kann, besteht die Rechnung aus der Addition zweier geklammerter Multiplikationen. Innerhalb der Multiplikationen werden Oberflächenfarbe und Reflexion zueinander gewichtet. Die Erläuterung der einzelnen Summanden und Faktoren der Rechnung ist der Abbildung zu entnehmen. Die Rechnung lautet für alle Farbkomponenten gleich, daher ist hier nur die R-Komponente angeführt.

Die Materialfarbe wird um 1 minus der Reflexionsstärke für die jeweilige Komponente abgeschwächt

Dies wird ausgeglichen, indem das Reflexionsergebnis mal der Reflexionsstärke hinzuaddiert wird

$$(out\_pResult \rightarrow r * (1 - reflection \rightarrow r) + reflectResult.r * reflection \rightarrow r) * (1 - lumaAmount) + lumaAmount * (out\_pResult \rightarrow r + reflectResult.r * reflection \rightarrow r * rlf * reflectionDamp)$$

① Die Basisfarbe wird nicht durch die Reflexionsstärke abgeschwächt

② Das Reflexionsergebnis wird mit der Reflexionsstärke, dem rfl und dem Reflexionsfalloff multipliziert und zur Materialfarbe hinzuaddiert

① ② Die beiden großen Multiplikationen werden über lumaAmount zueinander gewichtet. Die Werte von lumaAmount bewegen sich zwischen 0 und 1. Je größer der eine Faktor wird, desto geringer wird der andere.

**Abbildung 33:** Diese Rechnung gewichtet das herkömmliche und das im lumaReflect-Shader eingeführte Reflexionsmodell zueinander.

Somit ist die Berechnung des Shaderaufufes abgeschlossen. Der Shader gibt mit der Rückgabe von miTrue den Abschluss seiner Arbeit zu erkennen.

## B.7 Einzubindende Libraries und Pfade

Bevor man den Shader kompilieren kann, müssen ein paar Einstellungen der Entwicklungsumgebung von MS Visual C++ getroffen werden, da sonst der Compiler seine Arbeit nicht erledigen kann. Es muss der Phad zu "shader.lib" und "shader.h" angegeben werden. Unter Extras > Optionen > Verzeichnisse wählt man aus dem Dropdown-Menü "Include-Verzeichnisse", doppelklickt auf die nächste freie Zeile und fügt folgenden Pfad inklusive Laufwerksangabe hinzu: <XSIInstallation>\XSIDDK\include. Einfacher geht es, wenn man über den Dateibrowser das Verzeichnis angibt, indem man im Fenster mit der Pfadangabe auf den Button mit den drei Punkten klickt.

Das selbe Vorgehen ist für die Bibliothekdateien notwendig. Dazu wählt man aus dem Dropdownmenü von vorhin "Bibliothekdateien" aus und gibt im Fenster folgenden Pfad an:

```
"<XSIInstallation>\XSI_4.2\XSIDDK\LIB\NT-X86".
```

Zuletzt wird in den Projekteinstellungen (Projekt > Einstellungen > Reiter: "Linker") in der Zeile Objekt-/ Bibliothekmodule "shader.lib" eingegeben. In dieser Datei befindet sich der Code für die Funktionen welche in "shader.h" nur deklariert wurden. Zu beachten ist die Einstellung links oben im Dropdown "Win Release" oder "alle Konfigurationen". Es reicht die Einstellungen für die Release-Konfiguration zu treffen.

## B.8 Kompilieren

Vor dem Kompilieren des Projektes muss man noch einstellen, dass es sich um die Release-Version handelt. Würde man den Code als Debugversion kompilieren ergäbe das eine große .DLL-Datei voller Debuginformationen für Entwickler. Man stellt die Release-Konfiguration unter "Erstellen >aktive Konfiguration festlegen..." ein.

Das Projekt sollte sich jetzt einwandfrei kompilieren lassen. Der Compiler hält bei Tippfehlern an und gibt den Ort des Fehlers bekannt. Der hier abgedruckte Code (der auch auf CD-Rom beiliegt) wurde mit Visual C++ Version 6.0 getestet und kompiliert.

## B.9 SPDL-Logik

Der Shader ist soweit betriebsfertig. Allerdings soll noch etwas Klarheit in die Propertypage gebracht werden. Wird zum Beispiel der Fadeout-Mechanismus deaktiviert, so sollen auch die Eingebemöglichkeiten in der Propertypage nicht mehr vorhanden sein, in dem sie deaktiviert werden. Dazu gibt es am unteren Ende der SPDL-Datei eine Sektion mit der Bezeichnung "Logic". Der Körper besteht nur aus einer öffnenden und einer schließenden Klammer. Einzelne Routinen werden mit `sub` eingeleitet und mit `end sub` geschlossen.

Im Logic-Teil von SPDL-Dateien lässt sich, wie der Name schon sagt, die interne Logik der Propertypage festlegen.

Hier kann programmiert werden was geschieht, wenn gewisse Parameter an- oder abgewählt werden, welche Auswirkungen Änderungen eines Parameters auf andere Parameter haben und so weiter. Es gibt für die Parameter von Propertypages verschiedene Event Handler. Diese sind

<code>OnInit()</code>	Ein <code>OnInit()</code> -Event wird ausgelöst, sobald man die Propertypage öffnet. Im <code>OnInit()</code> -Eventhandler müssen alle <code>OnChanged()</code> -Handler registriert werden, damit gleich beim Öffnen der PPG der Status aller Parameter abgefragt und die PPG richtig angezeigt wird.
<code>parameter_OnChanged()</code>	Ein <code>parameter_OnChanged()</code> -Event wird ausgelöst, wenn sich der Wert eines Parameters ändert.
<code>scriptname_OnClicked()</code>	Ein <code>&lt;button_name&gt;_OnClicked()</code> -Eventhandler wird ausgelöst, wenn der damit verbundene Button geklickt wird

`tabname_OnTab()` Ein `tabname_OnTab()`-Event wird ausgelöst, wenn der User die entsprechende Registerkarte auswählt.

Die Logik-Sektion verwendet dasselbe Scriptingmodell wie der Softimage|XSI Skripteditor. Somit hat man über das Scriptingfenster Zugriff auf alle XSI-Kommandos sowie das Object-Model. Innerhalb eines Event Handlers hat man über den Namen eines Parameter Zugriff auf seine Werte und kann diese abfragen, setzen oder ihre Sichtbarkeit bestimmen. Der Wert eines Parameters wird einfach über seinen Parameternamen gewonnen. Man kann auch die längere Variante schreiben, nämlich `parameter.Value()`.

Wertzuweisungen werden über den "="-Operator durchgeführt. Man kann Parameter in einer Property page aktivieren und deaktivieren. Deaktivierte Parameter sind grau unterlegt und lassen sich nicht verändern. Der Befehl dazu lautet `parameter.Enable(True|False)`. Weiters können Parameter ein- und ausgeblendet werden über `parameter.Show(True|False)`.

Für die Logik des Shaders werden allerdings nur minimale Funktionalitäten benötigt. Und zwar:

- Je nach eingestelltem Specular-Modell sollen die nicht benötigten Parameter deaktiviert werden. Im Falle des Phong-Modelles ist `roughness` und `specularRefraction` deaktiviert, im Fall des Blinn-Modells `specularDecay`. Wenn kein Specular Modell verwendet wird, sind alle Parameter inklusive Farbreger gesperrt.
- Wenn die Option `fadeOut` nicht aktiviert wurde, sollen die dazugehörigen Falloff-Parameter ebenfalls deaktiviert werden.
- Der Regler `nearDist` darf keinen größeren Wert annehmen als `farDist`.
- Genauso darf `farDist` nie kleiner als `nearDist` werden. Wird ein Regler auf einen unerlaubten Wert gestellt, reagiert der andere Regler damit, dass er denselben Wert annimmt und somit beide Werte gleich groß sind.

Alle Anforderungen werden durch die Verwendung des "OnChanged()" -Event Handlers erfüllt. Die Ausprogrammierung der einzelnen Handler ist im folgenden Ausschnitt aus der SPDL-Datei "lumaReflect.spdl" zu sehen.

```
Logic
{
    sub nearDist_onChanged()
        if(nearDist > farDist) then
            farDist = nearDist
        end if
    end sub

    sub farDist_onChanged()
        if(farDist < nearDist) then
            nearDist = farDist
        end if
    end sub

    sub fadeOut_onChanged()
        if (fadeout = FALSE) then
            inverseFade.Enable(False)
            squareFade.Enable(False)
            nearDist.Enable(False)
            farDist.Enable(False)
        else
            inverseFade.Enable(True)
            squareFade.Enable(True)
            nearDist.Enable(True)
            farDist.Enable(True)
        end if
    end sub

    sub specularMode_onChanged()
        if(specularMode = 0) then
            specularColor.Enable(True)
            specularDecay.Enable(False)
            rougness.Enable(True)
            specularRefraction.Enable(True)
        end if
        if(specularMode = 1) then
            specularColor.Enable(True)
            specularDecay.Enable(True)
            rougness.Enable(False)
            specularRefraction.Enable(False)
        end if
        if(specularMode = 2) then
            specularColor.Enable(False)
            specularDecay.Enable(False)
            rougness.Enable(False)
            specularRefraction.Enable(False)
        end if
    end sub

    sub onInit()
        fadeOut_onChanged
        specularMode_onChanged
        nearDist_onChanged
        farDist_onChanged
    end sub
}
```

Der Großteil der Arbeit ist mit dem Erstellen des Codes abgeschlossen. Im Normalfall schreibt ein Anwender einen Shader dieses Umfangs nicht ohne ihn zwischendurch im 3D-Programm zu testen. So werden Fehler rasch erkannt und können sofort beseitigt werden. Hat man jedoch mehrere Funktionalitäten auf einmal programmiert und stimmt dann, das Renderergebnis nicht ist es schwieriger das Problem ausfindig zu machen. Durch gezieltes Verändern der Parameterwerte in der Propertypage des

Shaders lassen sich die Fehlerquellen rasch eingrenzen und der Code an den entsprechenden Stellen überarbeiten.

Das Integrieren des Shaders in Softimage erfordert noch folgende Schritte: Eine letzte Kontrolle der SPDL-Datei und das Registrieren des Shaders. Die Registrierung des Shaders kann entweder von der Kommandozeile (für Shaderwriter die ihre Shader wiederholt testen empfohlen) aus erfolgen, oder in Softimage|XSI direkt vorgenommen werden.

## B.10 Kontrolle von SPDL-Dateien

Bevor der Programmierer seinen Shader das erste Mal registriert, soll er die Syntax der SPDL-Datei überprüfen. Dazu öffnet man die Kommandozeile, die im Startfolder von Softimage zu finden ist. Der Unterschied zur herkömmlichen Kommandozeile besteht darin, dass die Datei

`<XSIVerzeichnis>\Application\bin\setenv.bat` aufgerufen wird. Diese Datei setzt benötigte Umgebungsvariablen für das Bedienen der Kommandozeilentools.

Um die SPDL-Datei zu überprüfen, wechselt man in das Verzeichnis des Shaders und führt folgendes Kommando aus:

```
spdlcheck <shadername>.spdl
```

Darauf wird die SPDL-Datei geparkt und auf Fehler überprüft. Das Programm gibt eine entsprechende Meldung aus wenn die Datei erfolgreich geparkt wurde. Tritt ein Fehler auf, hält das Programm und gibt die Nummer der fehlerhaften Zeile bekannt. Trotz der Verwendung des Shader-Wizards kann es zu Fehlermeldungen kommen, wie der Autor und andere Shaderwriter aus eigener Erfahrung zu berichten wissen.

Mögliche Fehlerursachen:

- Bei der Definition der UI-Types "Check", "Radio" oder "Combo" verabsäumt der Shader-Wizard eine Zeile mit einem Strichpunkt abzuschließen, der an dieser Stelle erforderlich gewesen wäre. Außerdem wird ohne Rücksicht auf den gewählten UI-Typ immer "Combo" eingestellt. Wenn anders erwünscht, muss "Check" oder "Combo" stattdessen geschrieben werden.

- Das Tool `spdlcheck` beschwert sich über fehlerhafte GUIDs. Auch das dürfte seinen Ursache in den verschiedenen Java Runtime Environments haben. Auf manchen Rechnern erstellte SPDLs erhielten korrekte GUIDs. SPDLs, die mit dem Shader-Wizard auf anderen Rechnern erstellt wurden, wurden wiederum von `spdlcheck` aufgrund inkorrektur GUIDs zurückgewiesen. Man kann die fehlerhaften GUIDs mit dem Programm `guidgen.exe`, welches mit der Visual C/C++ Entwicklungsumgebung auf dem System installiert wird, erzeugen und die entsprechende GUID aus der SPDL Datei damit ersetzen. Auch Softimage|XSI kann GUID's erstellen. Dazu öffnet man das Skriptfenster (der Button mit der Schriftrolle links unten), fügt den Code `logmessage XSIFactory.CreateGuid` ein und drückt "run". Im Infofenster wird eine gültige GUID ausgegeben welche für die SPDL Datei verwendet werden kann. Bei einer weiteren Betätigung des "run"-Buttons wird eine neue GUID ausgegeben.

## B.11 Installation und Deinstallation von Shader

### In der Softimage-Kommandozeile:

Man wechselt in das Verzeichnis des Shaders und führt zur Installation folgendes Kommando aus:

```
xsi -i <shadername>.spdl
```

Das Kommando für die Deinstallation lautet:

```
xsi -u <shadername>.spdl
```

Daraufhin ist kurz Geduld gefragt. Man bekommt in einem zweiten sich öffnenden Kommandozeilenfenster die Bestätigung, dass der Shader registriert, beziehungsweise entfernt wurde.

### Registrieren des Shaders in Softimage:

Unter File>Addon>Plugins(SPDL) hat man die Möglichkeit bestehende SPDLs entweder zu installieren oder zu deinstallieren. Nach der Installation eines Shaders ist der Neustart von Softimage|XSI erforderlich. Entwicklern, die ihre Shader häufig testen, sei deshalb empfohlen, ein

Kommandozeilenfenster ständig geöffnet zu haben, von dem aus sie Shader installieren und deinstallieren können.

## B.12 Shader testen

Was zuletzt übrig bleibt ist, den Shader in Softimage zu testen. Dazu baut man sich eine einfache Testszene zusammen, die alle Fähigkeiten des Shaders ausnutzt. Natürlich kann auch eine ältere Szene geladen werden, die Anlass für die Programmierung des Shaders war.

Man öffnet den Rendertree der gewünschten Geometrie. Im Menü "Nodes" wählt man den Punkt "Illumination" aus und klickt in dem daraufhin erscheinenden Submenü auf "more...". Daraufhin erscheint ein Dateibrowser. Rechts oben befindet sich ein Button mit der Aufschrift "Paths". Klickt man auf diesen, erscheint ein weiteres Auswahlmenü, auf dem man "user" auswählt. Diese Verlinkung führt zu dem Pfad, in dem eigene Installationen des Users zu finden sind, darunter auch selbst installierte Shader. Man wählt den gewünschten Shader mit einem Doppelklick aus und erhält damit den Node im Rendertree. Ab diesem Zeitpunkt kann man wie gewohnt den Shader im Rendertree verwenden. Dank Shaderassignment ist es möglich, an diese Eingänge wiederum andere Shader anzuschließen.



*Abbildung 34: Kugeln in Kellergwölbe*

Reflexion befinden unangetastet bleibt.

Die blaue Kugel ist mit einem Inverse Falloff versehen, weshalb die Reflexionen in der nächsten Umgebung der Kugel nicht berücksichtigt werden.

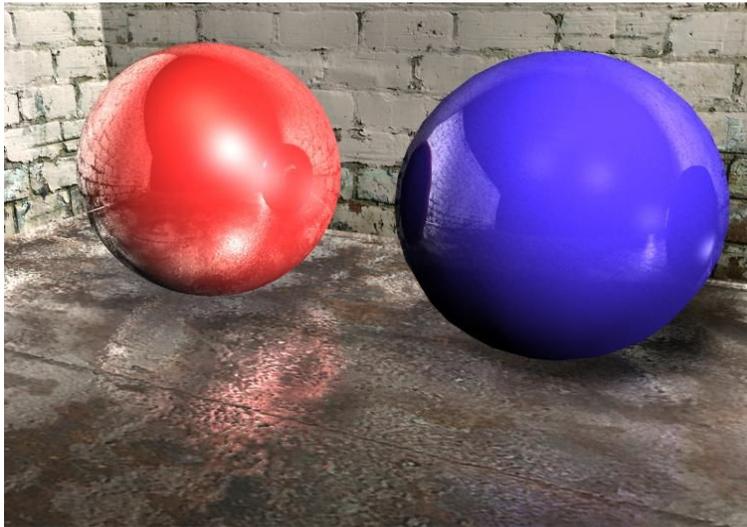
### B.12.1 Die Testbilder

Die Testbilder demonstrieren eine einfache Szene mit diversen Settings. Boden und Kugeln haben alle den neuen Shader zugewiesen und wurden mit diversen Settings versehen. Die rote Kugel präsentiert eine glatte Oberfläche mit Reflexion nach dem Luminanzmodell. Man kann sehr gut erkennen, dass trotz sichtbarer Reflexion die originale Materialfarbe dort, wo sich dunkle Bereiche in der



*Abbildung 35: Texturierte Kugeln mit dem lumaReflect-Shader*

Reflexionen diffus werden. Dank des neuen Reflexionsmodelles wird bei keiner einzigen Kugel die Originalfarbe abgeschwächt, sondern die Reflexion je nach Helligkeit des Reflexionsergebnisses zugemischt. Dieser Effekt ist am Boden ebenfalls erkennbar.



*Abbildung 36: Reflexion nach dem Luminanz-Prinzip, einmal ohne Falloff und einmal mit inversem Falloff*

Das nächste Bild zeigt sehr schön, dass man den Shader wie gewohnt texturieren und mit einer Bump aktivieren. Die rostige Metallkugel hat als Reflexionseingang eine gammakorrigierte SW-Version der Metalltextur. Daher reflektiert die Kugel nicht gleichmäßig, sondern erhält auf diese Weise ein abgenutztes Aussehen.

Die Goldkugel ist mit einer Bumpmap versehen, die eine raue Oberfläche erzeugt. Die Folge ist, dass die

## B.13 Zusammenfassung

*In diesem Kapitel werden die letzte Ergebnisse dieser Diplomarbeit zusammengefasst präsentiert. Es wird die forschungsleitende Fragestellung beantwortet, sowie ein Fazit aus dieser Arbeit gezogen. Anschließend folgt die persönliche Interpretation.*

### B.13.1 Beantwortung der forschungsleitenden Fragestellungen:

#### **Wie sind die beiden Programme Mental Ray und Softimage miteinander verflochten?**

Softimage stellt seine 3D Daten intern anders dar als Mental Ray. Mental Ray akzeptiert normalerweise .mi-Dateien als Input. Softimage kann zwar .mi-Dateien auf die Festplatte schreiben, umgeht aber den Schritt erst die .mi-Datei zu erzeugen und dann Mental Ray mit dieser .mi-Datei zu speisen. Wenn der Befehl zum Rendern kommt, übersetzt Softimage seinen Inhalt für den Renderer ohne ein Zwischenformat zu erzeugen.

#### **An welchen Schnittstellen muss man einen selbstprogrammierten Shader einbinden?**

3D-Animation wäre ohne einer grafischen Benutzeroberfläche heutzutage undenkbar. Mental Ray besitzt keine solche Benutzeroberfläche. Aber Softimage. Das Animationsprogramm zeigt änderbare Parameter in sogenannten "Propertypages" an, in denen der Benutzer Werte verändern und animieren kann. Auch Shaderparameter werden in Propertypages dargestellt.

Programmiert man Shader für Mental Ray und Softimage, muss dafür gesorgt werden, dass sowohl Mental Ray den Funktionscode des Shaders kennt, als auch Softimage über den Shader und seine Inhalte informiert wird. Das bedeutet, dass die beiden Programme verschiedene Informationen benötigen. Softimage kennt den Shadercode nicht, sondern lediglich die Shaderparameter welche der User einstellen kann. Diese Parameter und das Layout der Propertypage sind in einer SPDL-Datei gespeichert, die in Softimage registriert werden muss. Bei der Registrierung dieser Datei wird auch die DLL-Datei mit dem kompilierten Shadercode kopiert. Diese DLL wird nicht von Softimage, sondern von Mental Ray benötigt. Wird Mental Ray von Softimage aus aufgerufen, findet der Renderer in dieser DLL-Datei die Anweisungen wie der Shader zu berechnen ist.

Die Fähigkeit des Mental Ray Renderers selbst programmierten Shadercode ausführen zu lassen, birgt auch Gefahren. Es gibt verschiedene Arten von Shader. Jeder Shader darf nur bestimmte Operationen durchführen. Programmiert ein unerfahrener User einen Shader der unerlaubte Funktionsaufrufe tätigt, kann das zum Absturz des Programmes führen.

### **Welche Tools stehen zum Shaderwriting zur Verfügung und was ist bei ihrer Verwendung zu beachten?**

Die Installation eines Shaders erfolgt einfach. Indem man die SPDL-Datei registriert, wird die DLL-Datei mit dem Shadercode in ein Verzeichnis kopiert, in dem Mental Ray diese findet. Man registriert den Shader von der Softimage-Kommandozeile aus oder direkt in Softimage. Bei letzterem Fall muss man das Programm beenden und wieder hochfahren um den neuen Shader zu verwenden.

Der Prozess der Shaderkreation wird durch zwei Tools wesentlich erleichtert. Das wohl wichtigste ist der Shader-Wizard. Dieses Tool erzeugt die zum Shaderschreiben wichtigsten Dateien. Der Wizard richtet die C/C++-Dateien so her, dass sie die Funktionen und Parameter schon beinhalten. Man kann sich dadurch voll auf das Schreiben des Funktionscodes konzentrieren und muss nicht bei jedem Shader ständig wiederkehrende Aufgaben erledigen.

Weiters legt der Shader-Wizard eine funktionstüchtige SPDL-Datei für Softimage an. In dieser sind alle Shaderparameter so angelegt, wie sie im Shader-Wizard definiert wurden. Will man in die Property page noch Logik einprogrammieren, hat das händisch zu geschehen. Die Möglichkeiten sind zu groß, dass der Shader-Wizard diese Funktionalität nicht mehr abdeckt.

Leider hat das Programm einen kleinen Bug (Siehe Kapitel B.5.2.2) der einen falschen SPDL-Code erzeugt. Außerdem gibt es Schwierigkeiten mit verschiedenen JAVA-Runtimes und Webbrowsern. Ein Wizard der als Executable ausgeführt wird wäre wünschenswert.

Das zweite Tool ist das Programm "spdlcheck.exe", welches SPDL-Dateien auf ihren fehlerfreien Aufbau überprüft. Wurde ein Fehler gefunden, kann man anhand der ausgegebenen Zeilennummer diesen rasch ausbessern. Dieses kleine Programm erleichtert das Korrigieren enorm.

Das Schreiben des Programmcodes erledigt man am besten in einer Entwicklungsumgebung welche Syntaxhighlighting bietet. Das Microsoft Visual Studio hat sich für diese Diplomarbeit als eine gute Lösung herausgestellt. Die Umgebung ist einfach zu bedienen und den Bedürfnissen eines Programmierers angepasst.

### **Wie programmiert man Shader die über den Funktionsumfang der Standardshader hinausgehen?**

Dank der offenen Schnittstelle lassen sich völlig neue Shader für Mental Ray programmieren. Die Basisshader die im Installationspaket von Softimage schon enthalten sind, sind sehr vielseitig. Aber in manchen Situationen kann es passieren, dass man einen besonderen Shader benötigt, der nicht über den Rendertree realisiert werden kann.

Indem man den Code eines Shaders von Grund auf selber gestaltet selbst, lässt sich für beinahe jedes mögliche rendertechnische Problem eine Lösung finden. Die Bibliothek an Hilfsfunktionen in Mental Ray ist enorm, daher lässt sich fast jeder Shader Modular aus kleineren Funktionsaufrufen zusammenbauen. Man hat Zugriff auf alle Variablen des States und kann diese einfach manipulieren. Dabei muss darauf geachtet werden, keine Daten zu bearbeiten die keinesfalls verändert werden dürfen, weil sie der Renderer unbedingt benötigt. Genaueres ist der Mental Ray Dokumentation zu entnehmen.

### **B.13.2 Zusammenfassung der Resultate**

3D-Animation ist ein komplexes Fachgebiet. Allein das Rendern von 3D-Szenen lässt sich in weitere Fachbereiche unterteilen wenn man sich um spezielle Probleme kümmert. Man darf nicht jedes Mal das Rad neu erfinden, da man sonst keinen Schritt weiterkommt. Es gibt nur mehr wenige Menschen die sich mit diesem Thema auseinandersetzen, zum einen, weil es schon scheinbar genug Shader gibt, zum anderen, weil die Materie viel Einarbeitungszeit erfordert. Die meisten Shaderwriter arbeiten in Animationsstudios und sind dort als Technical Directors tätig. Sie wissen über fast jeden Aspekt von Mental Ray Bescheid und überwachen das Rendern bei großen Projekten. Als Technical Director muss man mit der Software die man bedient vertraut sein und Lösungen für Probleme bieten können. Man muss Wege finden, spezielle Effekte umzusetzen oder Szenen zu optimieren, damit sie sich in einer annehmbaren Zeit rendern lassen.

Mental Ray bietet die Möglichkeit dazu. Sobald man weiß, dass Softimage Mental Ray Shader über SPDL-Dateien integriert, kann man sich mit dem Interface vertraut machen und eigene Shader auf diese Weise einbauen. Der Shader-Wizard ist ein praktisches Tool das diesen Prozess beschleunigt.

Man muss beim kreieren neuer Shader immer ein Ziel vor Augen haben und genau planen, welchen Anforderungen der Shader genügen muss. Ein Bedürfnis muss erst erkannt werden bevor es befriedigt werden kann. Gründliche Recherche vor der Arbeit bleibt einem meist nicht erspart. Man sollte die Handbücher von Softimage und Mental Ray stets zur Hand haben, um bei speziellen Problemen mögliche Hilfestellungen zu finden.

Beim Programmieren muss man sich auf sein Können verlassen. Ist man man mit der Programmiersprache C/C++ vertraut, kann man äußerst effizienten Programmcode schreiben. Je besser man seinen Code gestaltet, desto schneller kann er ausgeführt werden und desto kürzer wird die Renderzeit für den Shader. Es lässt sich nicht immer vermeiden, dass ein Effekt leicht zu berechnen ist. Man muss manchmal

Abstriche machen. Regelmäßiges Testen während der Entwicklung eines Shaders beugt bösen Überraschungen vor.

### **B.13.3 Persönliche Interpretation**

Als ich mit dieser Diplomarbeit anfing, wusste ich nicht auf was ich mich da einlassen würde. Ich war stark an den Hintergründen von Mental Ray und der Verbindung des Renderprogrammes zu Softimage interessiert und wollte genauer untersuchen, welches Potenzial dahinter steckt, selber neue Shader zu programmieren. Allerdings wurde die Freude rasch gedämpft, da die Ressourcen zu diesem Thema sehr rar sind. Es bleiben nur die Handbücher von Softimage und die Bücher des Vaters von Mental Ray, Thomas Driemeyer. Letzterer ließ mich mit dem Buch "Programming Mental Ray" im Stich. Die Bestellung im Jänner 2005 aufgebend, erhielt ich einen Brief, der besagte dass das Buch im ersten Quartal 2005 herauskäme. Ein paar Monate später folgte ein zweiter Brief des Verlages, der den Termin auf das zweite Quartal 2005 verschob. Ich war auf andere alternativen angewiesen. Als eine der besten Ressourcen erwies sich die Homepage und die Mailinglist von Mental Images. Dort kann der Sourcecode einiger Standardshader heruntergeladen und unter die Lupe genommen werden. In der Mailinglist tauschen die Technical Directors der großen Animationshäuser Wissen, Fragen und Erfahrung aus. Auch Thomas Driemeyer ist am Briefverkehr beteiligt und hilft bei besonders kniffligen Problemen aus. Das Niveau dieser Mailinglist ist dementsprechend hoch, aber je besser man sich mit Mental Ray auskennt, desto mehr Information erhält man aus dem Wissensaustausch.

Als ich im Internet ein Tutorial fand, das einen Workaround zeigte wie man Reflexionen mit der Distanz ausfaden lassen kann, wusste ich was ich für diese Diplomarbeit programmieren würde. "See a need, fill a need<sup>10</sup>". User die sich danach sehnen, nicht in Compositingprogrammen 3 separat gerenderte Passes zusammenzubauen, sollten mit diesem Shader ihre Freude haben. Denn dieser Shader erledigt diese Arbeit schon während des Renderns. außerdem wollte ich ein neues Reflexionsmodell testen. Anstoß dazu gab mir ein Arbeitskollege in meiner Praktikumsfirma.

Es ist gerade wenig Arbeit wenn man einen Shader schreibt. Die unzählbaren Ray-Types sind sehr verwirrend und auch das Datenpaket des States gibt einem immer wieder Rätsel auf. Umso erfreulicher ist es, wenn man den Shader soweit hat, dass er in Softimage funktioniert. Das Debuggen des Shaders ist allerdings keine einfache Aufgabe. Wenn sich der Shader anders als geplant verhält, kommt eine äußerst zeitintensive und frustrierende Tätigkeit:

Der alte Shader muss in Softimage deinstalliert werden. Jede Codezeile des Shaders muss untersucht und

---

<sup>10</sup> Leitspruch des großen Erfinders Bigweld im Film "Robots" von Blue Sky Studios

der Code an der verdächtigen Stelle ausgebessert werden. Danach kompiliert man den Shader neu, registriert ihn, startet Softimage neu und rendert einen Bildausschnitt in der Testszene. Hat man den Fehler behoben, ist alles gut. Andernfalls beginnt die Prozedur von vorne. Dass dieser Prozess ermüdend werden kann, ist leicht verständlich.

Die SPDL-Logik ist ein VB-Skript, eine weit verbreitete Scriptingsprache, die zusätzlichen Einarbeitungsaufwand erfordert.

Es gibt genug Hürden. Hat man den Dreh aber heraus, wird der Prozess des Shaderschreibens eine interessante Aufgabe, welche die eigenen mathematischen und logischen Fähigkeiten weiter verbessert. Wer sich in Softimage und Mental Ray gut auskennt und schon einige gute 3D-Projekte gemacht hat, hat gute Chancen in diesem Bereich einen Job zu bekommen. Personen mit derartigen Fertigkeiten sind rar und der 3D-Markt wächst ständig.

Will man mit dem Shaderwriting anfangen, sollte man sich viel Zeit nehmen. Ein umfassendes Studium der Bücher "Rendering with Mental Ray®" und "Programming Mental Ray®" von Thomas Driemeyer ist Pflicht um den Einstieg ins Shaderwriting zu finden. Der erste Teil hat zwar noch nichts mit dem Shaderschreiben zu tun, ist aber vom Informationsgehalt über die Funktionsweise von Mental Ray enorm wichtig. Hier bekommt man einmal eine Ahnung vom Umfang dieser Software. Die Bücher handeln ausschließlich von "Mental Ray". Sie erläutern die Funktionsweise des Renderers bis ins kleinste Detail, erläutern Algorithmen und die Arbeitsweise von Mental Ray und zeigen Wege wie man mit dem Renderer am besten arbeitet, um das meiste aus der Software herauszuholen. Im Shaderschreiben verschmelzen die Bereiche Informatik, Mathematik und Kunst. Man sollte in jedem dieser Bereiche begabt sein. In der Mathematik sollte man sich besonders mit Vektorrechnung gut auskennen, da man beim Shaderwriting ständig mit Punkten und Vektoren im 3-Dimensionalen Raum arbeitet.

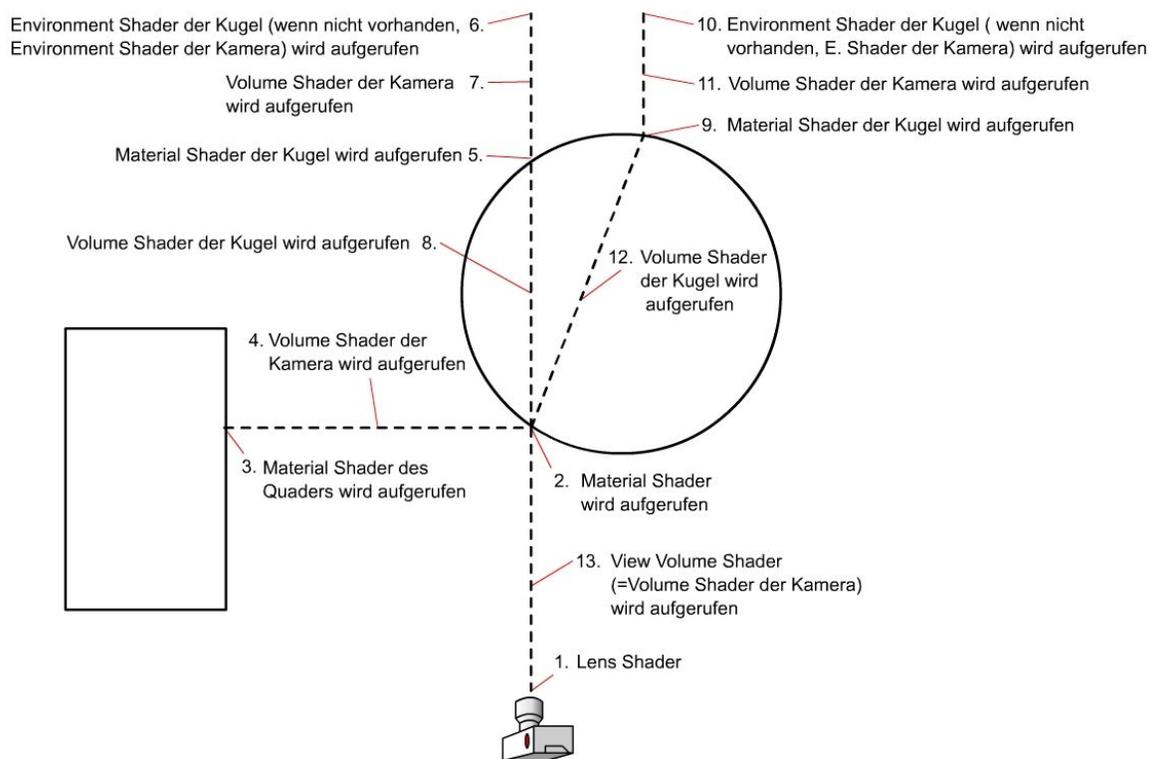
### **B.13.4 Diplomarbeitsthemen, die auf dieser Arbeit aufbauen können**

- Falls in Zukunft Mental Ray intensiveren Gebrauch von 3D-Hardware macht, kann dieser Bereich weiter untersucht werden. 3D-Hardware beherrscht zwar bislang kein Raytracing, dafür sind die 3D-Prozessoren für bestimmte Aufgaben optimiert und Dank der höheren Rechengenauigkeit neuer Grafikkarten für Rendersoftware interessant geworden.

- Die Untersuchung aktueller und neuer Shader. Gibt es noch ein verborgenes Potenzial an Shadingtechniken, die erst in Entwicklung sind? Äußerst interessante Shader wie Daniel Rind's Dirtmap Shader zeigen, dass es immer noch neue Ideen auf diesem Gebiet gibt. Auch der jüngste Hype des Subsurface-Scattererings kann genauer untersucht werden. Diese Shadingtechnik ermöglicht bislang nur schwer nachzuempfindende Effekte wie Licht, welches in durchscheinenden Materialien gestreut wird. Man kann untersuchen, ob der Einsatz solcher Shader in Animationen Sinn macht, wie rechenaufwändig diese sind und ob der Endverbraucher den Unterschied überhaupt erkennt beziehungsweise zu schätzen weiß.
- Die Implementierung der Cg-Sprache in Softimage. Cg ist eine Shadinglanguage für Hardware, die von Nvidia entwickelt wurde. Sie ist für den Gebrauch mit den neuesten und schnellsten Grafikkarten für die Spieleindustrie vorgesehen. Softimage hat das Potenzial der Spieleentwicklung erkannt und DirectX und Cg in seine Software integriert. Der Vorteil von Cg ist, dass man direkt in Softimage Cg-Shader schreiben und das Resultat in Echtzeit von der Grafikkarte berechnet bewundern kann. Es stellt sich auch die Frage, ob es sich nicht schon auszahlt, bestimmte Renderings von der Hardware erledigen zu lassen. Die neuesten Grafikkarten schaffen einfache Szenen in einem Bruchteil der Renderzeit und liefern ähnlich gute Bilder wie die wesentlich trägeren Software Renderer. Es mag sich durchaus bezahlt machen, Szenen in Hardware und Software-Passes zu unterteilen und zu compositen. Eine Untersuchung, ob neueste Hardware der alten Software Konkurrenz zu machen beginnt, lässt Ausblicke auf die Zukunft zu.

# C Anhang:

## C.1 Aufrufsreihenfolge von Shader in Mental Ray



**Abbildung37:** Aufrufsreihenfolge von Shadern entlang eines Rays.

An dieser Abbildung ist die Aufrufsreihenfolge der einzelnen Shader entlang eines Rays zu erkennen. Der Material Shader der Kugel erzeugt sogar 3 Secondary Rays: Einen Transparency-Ray, einen Reflection-Ray und einen Refraction-Ray. Dieses Verhalten ist eher außergewöhnlich und dient hier nur zur Veranschaulichung.

Man sieht, dass die Rays der Reihe nach abgearbeitet werden. Die letzten Shader für jeden Ray sind die Volume Shader. Erst wenn der letzte Shader ausgewertet wurde, liegt der endgültige Farbwert für das Sample vor. (Sampling, siehe Seite 32)

Es wurde außer acht gelassen, dass Material Shader meistens auch noch Light Shader und diese wiederum Shadow Shader aufrufen. Deswegen würde als Punkt 2.1 der Lichtshader einer oder mehrerer Lichtquellen aufgerufen. Als Punkt 2.2 würden die Shadow Shader von Objekten, die sich im

Strahlengang von der Lichtquelle zur Objektoberfläche der Geometrie befinden, aufgerufen werden. Dasselbe gilt auch für Punkt 3, 5 und 9.

## C.2 Quickstart zum Shaderwriting

Shader-Wizard starten und auf die Sicherheitsstufe achten. Unter Windows 2000 veranlasst die Installation der jüngsten JAVA J2RE, dass das Applet sich weigert die Dateien trotz gelockerter Restriktionen abzuspeichern. In diesem Fall empfiehlt sich die Deinstallation der neuen JAVA Version ebenso wie das Arbeiten mit der Standardversion, die bei der Windows 2000 Installation integriert ist. Unter Windows XP scheint dieses Problem behoben zu sein. Auf jeden Fall zahlt sich eine genaue Untersuchung der Browsereinstellungen aus. Das gilt auch für andere Browser als den Internet Explorer.

Man wählt den Shadernamen und Shadertyp, sowie den Ergebnistyp des Shaders aus. Utility-Shader, also Shader die lediglich die Ergebniswerte anderer Shader kombinieren oder mathematisch bearbeiten, sollen immer als Textureshader definiert werden.

Die Variablennamen (=Parameternamen) und der Typ werden anschließend auf der nächsten Seite eingestellt. Es können Standardwerte vordefiniert werden, was empfohlen wird. Weiters kann bei Parametern vom Typ "Scalar" eine Unter- und eine Obergrenze eingestellt werden. Wurde der Parameter mit "Add" hinzugefügt, kann man das genaue Erscheinungsbild des Parameters unter "UI Type" noch einmal genau bestimmen. Es gibt Kombinationen, die keinen Sinn machen (zum Beispiel einen Color-Wert vom Typ "Combo" zu machen und ähnliches). Bei einigen Parametern zahlt es sich auf jeden Fall aus, noch einmal explizit das richtige Verhalten einzustellen. Zum Beispiel bekommen Boolean Datentypen auch den UI-Typ "Boolean". Auswahllisten hingegen werden über Integers gemanagt und so kann man in den UI-Type Einstellungen die Benennung der einzelnen Einträge vornehmen.

Danach kann das Layout der Property page in Softimage genau eingestellt werden, indem man Tabs und Groups erstellt und die Parameter den Rubriken wie gewünscht zuweist.

Die Dateien werden im letzten Schritt in das anzugebende Verzeichnis geschrieben.

Danach öffnet man das Visual C Project File, das der Shader ebenfalls erstellt hat. Man sucht die Funktion, die den Namen des Shaders trägt und fügt eigenen Shadercode ein. Um die Parameterwerte des

Shaders zu importieren, erstellt man eine Variable vom richtigen Datentyp und weist ihr mit `mi_eval_*` ( ) den gewünschten Wert zu. Zu beachten ist, dass Pointer und Werte auf richtige Art und Weise zugewiesen werden.

## C.3 Referenz zu wichtigen `shader.h` – Funktionen und Datentypen

Die Datei "`shader.h`" ist voller Wertvoller Information über die Struktur von Mental Rays und den verwendeten Datentypen. Funktionen und Datentypen, die man bei fast jedem Shader benötigt, sind hier zum raschen Nachschlagen aufgelistet. Die Funktionen sind kurz erklärt und mit Tipps und Hinweisen ergänzt.

### C.3.1 Raytypes

Dank der genauen Unterscheidung zwischen den verschiedenen Raytypes können Shader genau an ihren Zweck angepasst geschrieben werden. Viele Shader sind Material, Shadow und Photon Shader in einem. Man kann im Funktionscode einfach unterscheiden, um welchen Raytype es sich handelt, wenn die Shaderfunktion aufgerufen wird. Man überprüft den Wert der Statevariablen `miRay_type`. Somit ließe sich auch ein Shader schreiben, der für alle möglichen Shadingzwecke eingesetzt werden kann. Von dieser Idee wird aber abgeraten, weil eine solche Methode gegen das Prinzip der Verständlichkeit und Interpretierbarkeit von Shader Nodes in Softimage|XSI verstößt. Prinzipiell ist aber alles möglich.

```
typedef enum miRay_type {
    miRAY_EYE,                /* eye ray */
    miRAY_TRANSPARENT,       /* transparency ray */
    miRAY_REFLECT,           /* reflection ray */
    miRAY_REFRACT,          /* refraction ray */
    miRAY_LIGHT,            /* light ray */
    miRAY_SHADOW,           /* shadow ray */
    miRAY_ENVIRONMENT,      /* ray only into environment/volume */
    miRAY_NONE,             /* other ray */
    miPHOTON_ABSORB,        /* photon is absorbed (RIP) */
    miPHOTON_LIGHT,        /* photon emitted from a light source*/
    miPHOTON_REFLECT_SPECULAR, /* specular reflection of a photon */
    miPHOTON_REFLECT_GLOSSY, /* glossy reflection of a photon */
    miPHOTON_REFLECT_DIFFUSE, /* diffuse reflection of a photon */
    miPHOTON_TRANSMIT_SPECULAR, /* specular transmission of a photon */
    miPHOTON_TRANSMIT_GLOSSY, /* glossy transmission of a photon */
    miPHOTON_TRANSMIT_DIFFUSE, /* diffuse transmission of a photon */
    miRAY_DISPLACE,         /* displacement during tessellation */
    miRAY_OUTPUT,          /* output shader */
    miPHOTON_SCATTER_VOLUME, /* volume scattering of a photon */
    miPHOTON_TRANSPARENT,  /* transparency photon */
    miRAY_FINALGATHER,     /* final gather ray */
    miRAY_LM_VERTEX,       /* light map vertex rendering */
    miRAY_LM_MESH,         /* light map mesh rendering */
    miPHOTON_EMIT_GLOBILLUM, /* globillum photons (emitters only) */
    miPHOTON_EMIT_CAUSTIC, /* caustic photons (emitters only) */
    miRAY_PROBE,           /* mi_trace_probe, ignores vis/trace */
    miRAPID_TILES,        /* tiles rendered. */
}
```

```

miRAPID_LEAVES_T,          /* leaves examined. */
miRAPID_LEAVES,           /* leaves passed on. */
miRAPID_SUBLEAVES_T,     /* subleaves examined. */
miRAPID_SUBLEAVES,       /* subleaves passed on. */
miRAPID_TRIS_T,          /* source triangles examined. */
miRAPID_TRIS,            /* source triangles passed on. */
miRAPID_SUBTRIS_G,       /* sub-triangles generated. */
miRAPID_SUBTRIS,         /* sub-triangles passed on. */
miRAPID_SAMPLES_T,       /* subtri-sample tests. */
miRAPID_SAMPLES_H,       /* subtri-sample hits. */
miRAPID_SHADING,         /* shading calls. */
miRAPID_SAMPLES,         /* subpixel (collect) samples. */
miRAY_HULL,               /* hull ray */
miPHOTON_HULL,           /* photon hull ray */
miRAY_NO_TYPES
} miRay_type;

```

Um einfach zu unterscheiden, ob ein Ray ein Primary- oder ein Secondary Ray ist, wurde in "shader.h" folgende Funktion definiert:

```

#define miRAY_PRIMARY(r)      ((r) == miRAY_EYE)
#define miRAY_SECONDARY(r)   ((r) > miRAY_EYE &&\
                               (r) < miPHOTON_ABSORB ||\
                               (r) == miRAY_FINALGATHER ||\
                               (r) == miRAY_HULL)

```

### C.3.2 Vektorrechnung:

Achtung, die Funktionen arbeiten mit Pointern vom Typ „miVector“ oder „miGeoVector“. Ist die Variable kein Pointer, so muss die Funktion mit der Adresse der Variable aufgerufen werden, also mit vorangestelltem „&“-Zeichen! Der dazugehörige Code ist in „shader.h“ zu finden.

Die Kürzel:

r Ergebnispointer, manchmal auch Argument vom Typ „miVector“ bzw. „miGeoVector“  
a, b, c Argumente vom Typ „miVector“ bzw. „miGeoVector“ - deren Werte nie verändert werden  
f Argumente vom Typ int, float, double, miInteger, miScalar oder miGeoScalar

Dreht die Vorzeichen von r um

mi\_vector\_neg(\*r)

Schreibt die Addition von a und b in die Variable r

mi\_vector\_add(\*r, \*a, \*b)

Subtrahiert a von b und schreibt das Ergebnis in r

mi\_vector\_sub(\*r, \*a, \*b)

Multipliziert r mit f und schreibt das Ergebnis in r

mi\_vector\_mul(\*r, f)

Dividiert r durch f und schreibt das Ergebnis in r

mi\_vector\_div(\*r, f)

Wie mi\_vector\_div(r,f), nur dass die Division mit double-Genauigkeit durchgeführt wird

mi\_vector\_div\_d(\*r, f)

Berechnet des Kreuzprodukt von a und b und schreibt das Ergebnis in r

```
mi_vector_prod(*r, *a, *b)
```

Berechnet das Skalarprodukt (im englischen als Dot-Product bekannt) und liefert dieses als Returnwert

```
mi_vector_dot(*a, *b)
```

Das Skalarprodukt mit double-Genauigkeit

```
mi_vector_dot_d(*a, *b)
```

Liefert den Betrag des Vektors r (also seine Länge) als Rückgabewert

```
mi_vector_norm(*r)
```

Liefert den Betrag des Vektors r als Rückgabewert mit double-Genauigkeit

```
#define mi_vector_norm_d(r)
```

Normalisiert r, ( $\text{SQRT}(a^2+b^2+c^2) = 1$ ), Ergebnis wird in r geschrieben

```
mi_vector_normalize(r)
```

Normalisierung mit double-Genauigkeit

```
mi_vector_normalize_d(*r)
```

Schreibt die jeweils niedrigere Komponente von a oder b in r

```
mi_vector_min(*r, *a, *b)
```

Schreibt die jeweils größere Komponente von a oder b in r

```
mi_vector_max(*r, *a, *b)
```

Liefert die Determinante der 3 Vektoren als Rückgabewert

```
mi_vector_det(*a, *b, *c)
```

Liefert die Distanz zwischen zwei Punkten als Rückgabewert

```
mi_vector_dist(*a, *b)
```

### C.3.3 Specular Highlights

```
miScalar mi_phong_specular ( miScalar spec_exp, miState *state, miVector *dir);  
  
miScalar mi_blinn_specular ( miVector *di, miVector *dr, miVector *n, miScalar roughness,  
                             miScalar ior);  
  
miBoolean mi_cooktorr_specular ( miColor *result, miVector *di, miVector *dr,  
                                 miVector *n, miScalar roughness, miColor *ior);  
  
miScalar mi_ward_glossy ( miVector *di, miVector *dr, miVector *n,  
                          miScalar shiny);
```

### C.3.4 Ray Direction verändernde Funktionen

*Folgende Seiten sind der Dokumentation von Mental Ray entnommen. Sie sollen zum Nachschlagen der Funktionen dienen, welche die Richtung eines Rays ändern [vgl. MRDOC1, 2005, node143.html].*

The functions in this section compute ray or photon directions. In both photon and ray tracing, shaders normally first compute a direction for secondary photons or rays using one of the functions with `_dir` in the name, and then call the corresponding photon or ray tracing function with the resulting direction.

The functions `mi_choose_scatter_type`, `mi_reflection_dir_*`, `mi_transmission_dir_*`, and `mi_scattering_dir_*` can also be used in other contexts than photon tracing, but they are listed in the photon tracing column above because they are most often used for that purpose. However, the glossy direction functions, for example, can just as well be used for ray tracing if the rendering quality settings (contrast and sampling limits) are high enough to avoid noisy images. Using diffuse directions for ray tracing is unlikely to produce acceptable results due to noise.

```
void mi_reflection_dir(
    miVector    *dir,
    miState     *state);
```

Calculate the reflection direction based on the `dir`, `normal`, and `normal_geom` state variables. The returned direction `dir` can be passed to `mi_trace_reflection`. It is returned in internal space.

```
void mi_reflection_dir_specular(
    miVector    *dir,
    miState     *state);
```

Same as `mi_reflection_dir`: computes the mirror direction. Created for symmetry with the similar functions for glossy and diffuse reflection.

```
void mi_reflection_dir_glossy(
    miVector    *dir,
    miState     *state,
    miScalar    shiny);
```

Choose a direction near the direction of ideal specular reflection (mirror direction). If shiny is low (for example, 5), a wide distribution of directions results; if shiny is high (for example, 100), a narrow distribution results.

```
void mi_reflection_dir_anisglossy(
    miVector    *dir,
    miState     *state,
    miVector    *u,
    miVector    *v,
    miScalar    shiny_u,
    miScalar    shiny_v);
```

Like `mi_reflection_dir_glossy`, but with different shinynesses in different directions. The `u` and `v` vectors specify the local surface orientation.

```
void mi_reflection_dir_diffuse(
    miVector    *dir,
    miState     *state)
```

Choose a direction with a distribution according to Lambert's cosine law for diffuse reflection.

```
miBoolean mi_refraction_dir(
    miVector    *dir,
    miState     *state,
    miScalar    ior_in,
    miScalar    ior_out);
```

Calculate the refraction direction in internal space based on the interior and exterior indices of refraction `ior_in` and `ior_out`, and on `dir`, `normal`, and `normal_geom` state variables. The returned direction `dir` can be passed to `mi_trace_refraction`. Returns `miFALSE` and leaves `* dir` undefined in case of total internal reflection.

```
miBoolean mi_transmission_dir_specular(
    miVector    *dir,
    miState     *state,
    miScalar    ior_in,
    miScalar    ior_out);
```

Same as `mi_refraction_dir`, since specular transmission occurs in the refraction direction. Created for symmetry with the similar functions for glossy and diffuse transmission.

```

miBoolean mi_transmission_dir_glossy(
    miVector    *dir,
    miState     *state,
    miScalar    ior_in,
    miScalar    ior_out,
    miScalar    shiny);

```

Choose a direction near the direction of ideal specular transmission (the refraction direction). If shiny is low, a very wide distribution of directions results; if shiny is high, a narrow distribution results.

```

miBoolean mi_transmission_dir_anisglossy(
    miVector *dir,
    miState *state,
    miScalar ior_in,
    miScalar ior_out,
    miVector *u,
    miVector *v,
    miScalar shiny_u,
    miScalar shiny_v);

```

Choose a direction for anisotropic glossy transmission. The u and v vectors specify the local surface orientation.

```

void mi_transmission_dir_diffuse(
    miVector    *dir,
    miState     *state)

```

Choose a direction with a distribution according to Lambert's cosine law for diffuse transmission (also known as "diffuse translucency").

```

void mi_scattering_dir_diffuse(
    miVector    *dir,
    miState     *state)

```

Choose a direction with a uniform probability over the whole sphere. The returned dir vector is normalized. This is useful for volume scattering.

```

void mi_scattering_dir_directional(
    miVector    *dir,

```

```
miState *state
miScalar directionality)
```

Choose a direction with a probability determined by directionality. For values between -1 and 0 it models volume backscattering (with -1 being the most directional), for a value of 0 it models diffuse (isotropic) volume scattering, and for values between 0 and 1 it models forward volume scattering.

```
miScalar mi_scattering_pathlength(
    miState *state,
    miScalar k);
```

Based on probability and exponential falloff, select a path length for a photon in a participating medium with density k.

## C.4 Literaturverzeichnis:

[LUEF03] Barbara Luef, High Dynamic Range Image Lighting, Diplomarbeit an der FH St. Pölten 2003

[BIB1] <http://www.cl.uni-heidelberg.de/kurs/ss04/prog2/html/page042.html>

Verwendung von Bibliotheken I

Last Modified 16.4.05

[BPSMP] Alexander Keller. To Trace or Not To Trace. Seminarvideo, Breakpoint '05  
[http://www.scene.org/file.php?file=%2Fparties%2F2005%2Fbreakpoint05%2Fseminars%2Fbp05\\_seminars\\_-\\_prof\\_dr\\_rer\\_nat\\_alexander\\_keller\\_-\\_to\\_trace\\_or\\_not\\_to\\_trace\\_-\\_that\\_is\\_the\\_question\\_-\\_xvid.avi&fileinfo](http://www.scene.org/file.php?file=%2Fparties%2F2005%2Fbreakpoint05%2Fseminars%2Fbp05_seminars_-_prof_dr_rer_nat_alexander_keller_-_to_trace_or_not_to_trace_-_that_is_the_question_-_xvid.avi&fileinfo)

[BSR] G. Scott Owen. Jim Blinn Model for Specular Reflection

[http://www.siggraph.org/education/materials/HyperGraph/illumin/specular\\_highlights/blinn\\_model\\_for\\_specular\\_reflect\\_1.htm](http://www.siggraph.org/education/materials/HyperGraph/illumin/specular_highlights/blinn_model_for_specular_reflect_1.htm) Last changed September 06, 1999,

[DLLEX] Microsoft Corporation, "MSDN Library - dllexport, dllimport",

[http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/msmod\\_20.asp](http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/msmod_20.asp)

Built on Thursday, May 11, 2000

[JAVA01] Scott Oaks. Java™ Security. O'Reilly & Associates, Auszug aus der Online-Ausgabe unter  
[http://www.unix.org.ua/oreilly/java-ent/security/ch01\\_02.htm](http://www.unix.org.ua/oreilly/java-ent/security/ch01_02.htm)

Last modified 5.01. 2005

[LAMRUG01] Los Angeles mental ray® User Group. Samples Tips

<http://www.lamrug.org/resources/samplestips.html>, last Modified 24.2.05

[MEY] Scott Meyers. Mehr Effektiv C++ programmieren. Addison-Wesley-Longman 1997 ISBN 6-8273-1275-2

[MRB] Th. Driemeyer. Rendering with Mental Ray, Springer Verlag 2001, ISBN 3-211-83403-6

[MRSMP, 2005, scn\_example.mi] Mental Images, Scene Example (mit Änderungen des Autors),  
[ftp://ftp.mentalimages.com/pub/books/book2\\_data.zip](ftp://ftp.mentalimages.com/pub/books/book2_data.zip) (scn\_example.mi)

[MRDOC1] Mental Ray Documentation, zu finden im Softimage-Installationsverzeichnis unter  
"<XSI\_VERZEICHNIS>\Doc\mental\_ray\manual"

[MRDOC2] Mental Ray Shaderdocumentation, zu finden im Softimage-Installationsverzeichnis unter  
"<XSI\_VERZEICHNIS>\Doc\mental\_ray\shaders"

[NAN2, 2003, <http://www.nanomation.co.uk/Programming-102.html>] D. Rowntree, XSI Shader Programming 102, (C) Copyright Nanomation Limited 2003,  
<http://www.nanomation.co.uk/Programming-102.html>

[OGL\_R] M. Woo, J. Neider, T. Davis, D. Shreiner. OpenGL® Programming Guide, Third Edition,

Addison-Wesley ISBN 0-201-60458-2

[PROVID] U. Schmidt. Professionelle Videotechnik, 2. Auflage Springer Verlag ISBN 3-540-66854-3

[SPDLREF] Softimage / Avid, SPEDL-Reference, zu finden im Softimage-Installationsverzeichnis unter "<XSI\_VERZEICHNIS>\Doc\XSISDK\spdlref"

[QMC1] Stefan Heinrich, Alexander Keller. Quasi-Monte Carlo Techniques in Computer Graphics, Part 1: The QMC-Buffer. Technical Report 242/94, <http://graphics.uni-ulm.de/QMCBuffer.pdf>

[WHP053] A.Roberts. The Film Look – It's Not Just Jerky Motion... Research & Development British Broadcast Corporation 2002  
<http://www.bbc.co.uk/rd/pubs/whp/whp-pdf-files/WHP053.pdf>

Helmut Herold. C Kompaktreferenz. Addison-Wesley-Longman 1999 ISBN 3-8273-1480-1

## Abbildungsverzeichnis

- Abbildung 1: Eine Projektionsmatrix welche eine Orthografische Projektion durchführt.  
Universität von Baireuth, <http://www.rz.uni-bayreuth.de/lehre/dibito/vorlesung/node48.html> 10
- Abbildung 2: Ein Apfel auf einem Schachbrettmuster. Der Apfel und der Boden werden mit einem Lambertshader berechnet. Rüdiger Raab 2005 11
- Abbildung 3: Texturierter Apfel auf altem Parkettboden. Rüdiger Raab 2005 12
- Abbildung 4 : Lampe, die einen Raum ausleuchtet. Ohne Global Illumination wäre nur das beleuchtete Dreieck auf dem Boden zu sehen. Rüdiger Raab 2004 14
- Abbildung 5: Geometrische Objekte von selbstleuchtenden Flächen mit der Final Gathering-Technik beleuchtet. Rüdiger Raab 2005 15
- Abbildung 6: Kugel mit Bump Mapping. Rüdiger Raab 2005 15
- Abbildung 7: Kugel mit Displacement Mapping. Rüdiger Raab 2005 15
- Abbildung 8: Szene mit Nebel. Die Strahlen reisen durch ein Volumen mit ungleichmäßiger Dichte. Rüdiger Raab 2005 16
- Abbildung 9: UV-Map auf einer vorbereiteten Textur. Rüdiger Raab 2005 20
- Abbildung 10: Texturiertes Objekt mit sichtbaren Polygonen. Rüdiger Raab 2005 20
- Abbildung 11: In Softimage erstellte Geometrie mit noch nicht triangulierten Polygonen. Rüdiger Raab 2005 26
- Abbildung 12: Dieselbe Geometrie, von Mental Ray trianguliert. Rüdiger Raab 2005 26
- Abbildung 13: Virtuelle Kamera, die auf einige Objekte gerichtet ist. Die blauen Linien zeigen die Sichtpyramide an. Rüdiger Raab 2005 27
- Abbildung 14: Blick aus der Virtuellen Kamera auf die Objekte. Aus dieser Perspektive wird Mental Ray das Bild berechnen. Rüdiger Raab 2005 28
- Abbildung 15: Ein Ray reist durch ein Volumen mit inhomogener Dichte. Die roten Punkte zeigen die Unterteilungen des Rays an. Rüdiger Raab 2005 29
- Abbildung 16: Pixelraster mit gekennzeichneten Pixelmittelpunkten. Los Angeles Mental Ray User Group 30
- Abbildung 17: Samplingpunkte liegen an den Ecken der Pixel. Los Angeles Mental Ray User Group 30
- Abbildung 18: Oversampling mit Minimum-Maximum 1 1. Los Angeles Mental Ray User Group

- 30
- Abbildung 19: Oversampling mit Minimum-Maximum 2 2. Los Angeles Mental Ray User Group 30
- Abbildung 20: Samples bei Adaptive Sampling mit Minumun-Maximum 0 3. Los Angeles Mental Ray User Group 31
- Abbildung 21: Modell der Mental-Ray Fabrik. Schritt 2 bis 5 wird so lange durchlaufen, bis alle Pixel des Bildes berechnet wurden. Die Schritte 1, 6 und 7 sind einmalige Aktionen. Rüdiger Raab 2005 33
- Abbildung 22: Beispiel eines Rendertrees in Softimage. Rüdiger Raab 2005 39
- Abbildung 23: Beispiel eines Phenomenons das im Rendertree als Shader auftaucht. Der Endbenutzer kann keinen Unterschied erkennen. Rüdiger Raab 2005 41
- Abbildung 24: Zuweisen der RGB-Werte an eine Propertypage. Rüdiger Raab 2005 43
- Abbildung 25: Ein Material in einer .mi wird aus Shadern zusammengestellt die zuvor deklariert und definiert wurden. Die Namen im Declare-Statement müssen dem Funktionsnamen in der C/C++-Datei entsprechen. Der Name bei der Definition darf frei gewählt werden, soll aber der C-Syntax entsprechen. Rüdiger Raab 2005 48
- Abbildung 26: Perlin Noise ist in der Mental Ray Bibliothek gebrauchsfertig implementiert. Rüdiger Raab 2005 65
- Abbildung 27: Caipirinhaglas mit Eiswürfeln. An den Glasrändern ist deutlich zu erkennen, dass der Renderer die maximale Raytracingdepth erreicht hat. Unschöne schwarze Bereiche sind die Folge. Rüdiger Raab 2004 76
- Abbildung 28: Reflexionen auf einem Auto. RsportsCars.com, [http://www.rsportscars.com/foto/05/focusrs02\\_08\\_1024.jpg](http://www.rsportscars.com/foto/05/focusrs02_08_1024.jpg) 80
- Abbildung 29: Die Startseite des Shader-s. Hier muss man die Entscheidung treffen, für welche Shadingzwecke der Shader eingesetzt wird. Spätere Änderungen sind nur mehr durch Editieren der SPDL-Datei möglich. Rüdiger Raab 2005 84
- Abbildung 30: Erstellen der einzelnen Parameter im Shader-Wizard. Rüdiger Raab 2005 86
- Abbildung 31: Berechnung des Falloff unter Berücksichtigung der vorderen und der hinteren Falloffgrenze. Rüdiger Raab 2005 100
- Abbildung 32: Graphen verschiedener Einstellungen für Threshold kombiniert mit s-curve. Rüdiger Raab 2005 101
- Abbildung 33: Diese Rechnung gewichtet das herkömmliche und das im lumaReflect-Shader eingeführte Reflexionsmodell zueinander. Rüdiger Raab 2005 102
- Abbildung 34: Kugeln in Kellergwölbe. Rüdiger Raab 2005 110
- Abbildung 35: Texturierte Kugeln mit dem lumaReflect-Shader. Rüdiger Raab 2005 110
- Abbildung 36: Reflexion nach dem Luminanz-Prinzip, einmal ohne Falloff und einmal mit inversem Falloff. Rüdiger Raab 2005 111
- Abbildung 37: Aufrufsreihenfolge von Shadern entlang eines Rays. MRDOC1, node102.html Mental Images – Zeichnung neu entworfen von Rüdiger Raab 2005 114