

Volumenvisualisierung im Rahmen des MRT

Diplomarbeit

vorgelegt von
Michael Pietsch

Institut für Informatik III
Rheinische Friedrich–Wilhelms–Universität Bonn

25. November 1997

Hiermit versichere ich, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Bonn, den 25. November 1997

Dank

Einleitend möchte ich folgenden Personen meinen Dank für ihre Unterstützung aussprechen: Stephan Schäfer und Gordon Müller danke ich für das Korrekturlesen der Arbeit und für ihre fachlichen Hilfestellungen. Norbert Schenk danke ich für viele interessante und fruchtbare Diskussionen. Auch möchte ich mich ausdrücklich bei Ute Simon und Manfred Berndtgen für ein weiteres Korrekturlesen der Arbeit bedanken. Besonderer Dank gilt auch meinen Eltern für ihren Glauben an mich und meine Arbeit.

Vor allem danke ich aber Herrn Professor Fellner für seine Betreuung und motivierende Unterstützung.

Inhaltsverzeichnis

1	Einleitung	1
2	Volumenvisualisierung	2
2.1	Grundlagen	2
2.1.1	Gitterstrukturen	3
2.1.2	Interpolationsmethoden	5
2.1.3	Gradientenberechnung	6
2.2	Isoflächen	7
2.2.1	Schnittpunktberechnung	7
2.2.2	Polygonale Approximation	9
2.3	Integrationsbasierte Visualisierung	12
2.3.1	Die Transportgleichung	12
2.3.2	Approximationen der Transportgleichung	15
2.3.3	Abbildung skalarer Daten auf Modellparameter	18
2.3.4	Visualisierungstechniken	21
3	Das MRT	25
3.1	Architektur	25
3.1.1	Szenenobjekte	26
3.1.2	Oberflächenbeschreibungen	28
3.1.3	Lichtquellen	29
3.1.4	Bildgenerierung	29
3.2	Strukturelle Veränderungen und Erweiterungen	30
3.2.1	Szenenobjekte	31
3.2.2	Schnittobjekte	32
3.2.3	Strahlobjekte	34

<i>INHALTSVERZEICHNIS</i>	iii
3.2.4 Basisklassen der Beleuchtungsrechnung	37
3.3 Volumenspezifische Klassen	40
3.3.1 Volumendaten und Traversierer	41
3.3.2 Isoflächen	44
3.3.3 Volumenobjekte	45
3.4 Zusammenfassung	48
4 Ergebnisse	51
5 Ausblick	60
A Szenenbeschreibungen	62
B Das Dateiformat von HP	68

Kapitel 1

Einleitung

Das *Minimal Rendering Toolkit* (MRT) [Fel94] ist ein an der Universität Bonn von der Fachgruppe Computergrafik entwickeltes Visualisierungssystem, welches primär im Rahmen von Forschung und Lehre eingesetzt wird, mittlerweile aber auch in der Industrie Anwendung findet. Als plattformunabhängiges, in der Programmiersprache C++ [Str91] implementiertes System, bietet es eine Vielzahl von Techniken zur Visualisierung dreidimensionaler Szenarien. So kann es zum Beispiel als Ray-Tracer, für die hardwareunterstützte interaktive Darstellung und Animation oder für Radiosity-Berechnungen benutzt werden.

Trotz der hohen Funktionalität und Flexibilität ist das MRT wie die meisten Systeme dieser Art jedoch nur für die reine Flächenvisualisierung konzipiert. Neben dieser gewinnt aber auch immer mehr die Visualisierung dreidimensionaler Datenfelder, sogenannter Volumendaten, an Bedeutung. Sei es zur photorealistischen Darstellung atmosphärischer Phänomene oder zur Darstellung von Meß- und Simulationsergebnissen aus wissenschaftlichen Bereichen.

Gegenstand dieser Arbeit ist die im Sinne des objektorientierten Paradigmas elegante Erweiterung des MRT für die ray-tracing- und polygonbasierte Visualisierung von Volumina. Der Schwerpunkt wurde insbesondere darauf gelegt, einen möglichst großen Teil der vorhandenen Systemfunktionalität zu nutzen und neben den hier realisierten Konzepten eine Basis für weitere Repräsentations- und Visualisierungstechniken zu etablieren.

Kapitel 2

Volumenvisualisierung

Im Rahmen dieses Kapitels sollen zunächst grundlegende Konzepte der Volumenvisualisierung, d.h. Konzepte zur Repräsentation und Visualisierung dreidimensionaler Datenfelder erläutert werden. Im Vordergrund standen dabei vor allem Techniken, welche sich besonders gut für eine Integration in das Minimal Rendering Toolkit eignen.

2.1 Grundlagen

Dreidimensionale Datensätze, sogenannte Volumendaten, definieren für bestimmte Punkte im Raum bestimmte Eigenschaften. Formal betrachtet entsprechen Volumendaten Funktionen der Form

$$f : \mathbf{R}^3 \rightarrow \mathbf{R}^n,$$

deren Definitionsbereich auf einen bestimmten Teilraum, typischerweise ein Quader, beschränkt ist. Wie es insbesondere bei Ergebnissen aus Messungen und Simulationen der Fall ist, sind Volumendaten in der Regel nun nicht in kontinuierlicher, sondern in diskreter Form durch eine endliche Menge von Datenwerten gegeben, die in einer bestimmten Gitterstruktur angeordnet sind. Im folgenden soll als Grundlage für die nächsten Abschnitte auf verschiedene Typen von Gitterstrukturen und darauf aufbauend auf Interpolationsmethoden sowie auf die für viele Visualisierungstechniken erforderliche Berechnung von Gradienten einge-

gangen werden.

2.1.1 Gitterstrukturen

In Abhängigkeit der räumlichen Anordnung und Verteilung der Datenwerte kann jedem Datensatz eine bestimmte Gitterstruktur zugeordnet werden. Benachbarte, d.h. durch Kanten verbundene Gitterpunkte, definieren dabei die Eckpunkte von Teilvolumina, die in der Literatur auch als *Zellen* bezeichnet werden. Der oft verwendete Begriff *Voxel* hingegen bezieht sich auf regulär strukturierte Gitter und bezeichnet einen um einen Gitterpunkt liegenden uniformen Bereich konstanten Wertes. Spray & Kennon [SK90] unterscheiden nun generell sieben Typen von Gitterstrukturen, welche hier nach aufsteigender Komplexität und Generalität aufgeführt sind. Abbildung 2.1 zeigt die Gitterstrukturen im Überblick.

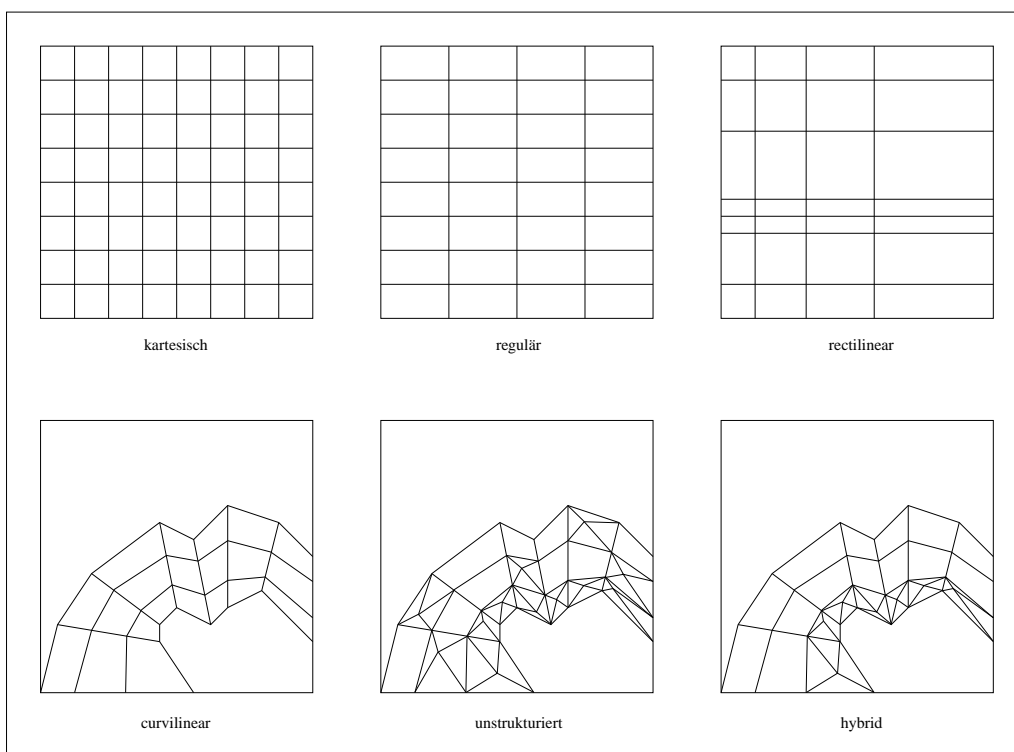


Abbildung 2.1: Gitterstrukturen

- **Kartesische Gitter**
Dies ist die einfachste Gitterstruktur. Die Zellen sind Würfel gleicher Größe und achsenparallel ausgerichtet.
- **Reguläre Gitter**
Reguläre Gitter sind ähnlich zu kartesischen Gittern. Nur sind hier die Zellen keine Würfel, sondern Quader. Datensätze aus der Computertomographie, Sonographie und Magnetresonanztomographie sind in der Regel in dieser Struktur gegeben.
- **Rectilineare Gitter**
Die Zellen sind wie bei regulären Gittern Quader. Allerdings sind die Abstände zwischen den Gitterpunkten entlang der Koordinatenachsen beliebig.
- **Curvilineare Gitter**
Curvilineare Gitter entsprechen kartesischen Gittern, auf deren Gitterpunkte eine nicht-lineare Transformation angewendet wurde. Die Gitterstruktur kann somit Zellen enthalten, deren Flächen nicht planar sind. In dieser Gitterstruktur sind üblicherweise Daten aus der Strömungssimulation gegeben.
- **Block-strukturierte Gitter**
Diese Gitter sind aus mehreren curvilinearen Gittern zusammengesetzt. Sie werden insbesondere für eine adaptive Darstellung von Volumendaten benutzt.
- **Unstrukturierte Gitter**
Dieser Struktur sind keine geometrischen Beschränkungen auferlegt. Die Zellen können Tetraeder, Hexaeder, Prismen, etc. sein. Diese Gitterstruktur ist insbesondere in der Finite-Element-Analyse von Bedeutung.
- **Hybride Gitter**
In manchen Fällen ist es wünschenswert, strukturierte und unstrukturierte Gitter zusammen zu benutzen. Die daraus resultierenden Gitter werden als hybride Gitter bezeichnet.

Die meisten in der Literatur bekannten Verfahren zur Visualisierung von Volumina gehen davon aus, daß die Daten in einem regulären oder rectilinearen Gitter

gegeben sind. Für die Visualisierung anders strukturierter Datensätze können die Daten in eine reguläre Gitterstruktur mittels *Resampling* überführt werden. Dies hat allerdings den Nachteil, daß für eine adäquate Repräsentation eine in der Regel sehr hohe Gitterauflösung zu wählen ist, wodurch ein enormer Speicherplatzbedarf entsteht. Aus diesem Grunde befassen sich viele Autoren (z.B. [WC90], [Gar90]) auch zunehmend mit Verfahren zur direkten Visualisierung nicht regulär strukturierter Datensätze.

2.1.2 Interpolationsmethoden

Für die Visualisierung eines in diskreter Form gegebenen Datensatzes ist es oft notwendig, die Daten der durch den Datensatz repräsentierten Funktion aus den diskreten Gitterwerten zu rekonstruieren. Dies geschieht unter Anwendung einer Interpolationsmethode. Die bekanntesten Interpolationsmethoden [WC90] sind *nearest Neighbor*, *trilineare Interpolation* und *inverse Distanzgewichtung*. Mit letzteren beiden Methoden wird der Datenwert eines Punktes in der Regel aus den Datenwerten der Eckpunkte der Zelle berechnet, in welcher der Punkt enthalten ist.

Die Nearest-Neighbor-Methode ist die einfachste. Der für einen Punkt zu bestimmende Datenwert wird einfach durch den Datenwert des am nächsten liegenden Gitterpunktes approximiert. Ist die Auflösung des Gitters geringer als die Auflösung der Bildebene, auf welche die Daten abzubilden sind, ist typischerweise die mit dem Gitter gegebene Blockstruktur zu erkennen. In der Regel bietet sich diese Methode nur für eine sehr approximative Visualisierung an.

Bei der trilinearen Interpolation wird angenommen, daß der Datenwert innerhalb einer Zelle linear entlang der Hauptachsen variiert. Für einen Punkt $p = (x, y, z)$ ergibt sich dessen Datenwert dann durch

$$f(p) = a + bx + cy + dz + exy + fxz + gyz + hxyz,$$

wobei a, b, \dots, h entsprechende Koeffizienten sind. Berechnet werden können diese, indem für acht Gitterpunkte der Zelle deren Datenwerte und Positionen eingesetzt werden. Dies ergibt ein aus acht Gleichungen bestehendes Gleichungssystem, dessen Lösung die gesuchte Interpolationsfunktion für eine Zelle liefert.

Bei der inversen Distanzgewichtung wird der Datenwert $f(p)$ eines gegebenen

Punktes p durch die gewichtete Summe der Datenwerte f_i der benachbarten n Punkte p_i berechnet:

$$f(p) = \sum_{i=0}^{n-1} w_i f_i.$$

Das Gewicht w_i eines Punktes p_i hängt dabei invers von der Distanz zwischen p und p_i ab. Je größer die Distanz, desto geringer dessen Einfluß. Berechnet werden können die Gewichte gemäß [BS84] durch

$$w_i = \prod_{k=0, k \neq i}^{n-1} [d_k(p)]^2 / \sum_{j=0}^{n-1} \prod_{l=0, l \neq j}^{n-1} [d_l(p)]^2,$$

wobei $d_j(p)$ die euklidische Distanz zwischen p und p_j ist.

2.1.3 Gradientenberechnung

Neben den reinen Datenwerten werden für die Visualisierung oft auch die Gradienten der Funktion benötigt. Bei der Darstellung von Iso- bzw. Konturflächen dienen diese zum Beispiel zur Approximation von Normalvektoren. Berechnet werden kann der Gradient an einem Punkt generell aus den Differenzen der Datenwerte und Koordinaten benachbarter Paare von Gitterpunkten. Wie der Gradient konkret berechnet wird, hängt von der gegebenen Gitterstruktur ab. Hier soll lediglich ein Standardverfahren für reguläre Gitter vorgestellt werden. Die Komponenten des Gradienten $g = (g_x, g_y, g_z)$ an einem Gitterpunkt (i, j, k) werden dabei mittels Differenzbildung entlang der drei Koordinatenachsen durch

$$\begin{aligned} g_x(i, j, k) &= \frac{f(i+1, j, k) - f(i-1, j, k)}{\Delta x} \\ g_y(i, j, k) &= \frac{f(i, j+1, k) - f(i, j-1, k)}{\Delta y} \\ g_z(i, j, k) &= \frac{f(i, j, k+1) - f(i, j, k-1)}{\Delta z} \end{aligned}$$

approximiert, wobei $f(i, j, k)$ der Datenwert am Gitterpunkt (i, j, k) ist und Δx , Δy , Δz die Längen der Kanten einer Zelle sind. Für einen nicht auf einem Gitterpunkt liegenden Punkt kann der Gradient dann aus den Gradienten der benachbarten Gitterpunkte unter Anwendung der gewünschten Interpolationsmethode ermittelt

werden.

2.2 Isoflächen

Die Darstellung bzw. Extraktion von Isoflächen ist ein relativ einfacher und auch oft praktizierter Ansatz zur Visualisierung dreidimensionaler Skalarfelder. Konkret ist eine Isofläche eine Menge von Punkten, deren Datenwerte gleich einem festen vorgegebenen Schwellwert sind. Bezüglich des vom Benutzer gegebenen Schwellwertes kann der Schnittpunkt eines Strahls mit der Isofläche als auch eine polygonale Approximation für diese berechnet werden, womit sich die Darstellung von Isoflächen auch leicht in einen konventionellen Ray-Tracer oder Polygon-Renderer integrieren läßt.

2.2.1 Schnittpunktberechnung

Wie in Abbildung 2.2 gezeigt, besteht ein möglicher Ansatz zur Berechnung des Schnittpunktes zwischen einem Strahl und einer Isofläche darin, das Volumen entlang des Strahls in regelmäßigen Abständen abzutasten, bis zwei aufeinanderfolgende Abtastwerte $f(p_i)$ und $f(p_{i+1})$ gefunden sind, zwischen denen der Schwellwert f_{thres} liegt, d.h. bis

$$f(p_i) \leq f_{thres} \leq f(p_{i+1}) \text{ oder } f(p_i) \geq f_{thres} \geq f(p_{i+1}),$$

wobei p_j gleich dem j -ten Abtastpunkt entspricht. Der gesuchte Schnittpunkt p_{thres} kann dann durch lineare Interpolation

$$p_{thres} = \frac{(f(p_{i+1}) - f_{thres})p_i + (f_{thres} - f(p_i))p_{i+1}}{f(p_{i+1}) - f(p_i)}$$

oder mittels binärer Suche approximiert werden. Die binäre Suche wird solange durchgeführt, bis die Differenz zwischen Abtastwert und Schwellwert kleiner einem vorgegebenen Epsilonwert ist.

Ist dem Volumendatensatz eine bestimmte Gittertopologie zugeordnet, so kann alternativ zum ersten Ansatz entsprechend Abbildung 2.3 das Volumen entlang des Strahls auch Zelle für Zelle traversiert werden, wobei eine Zelle einem (kon-

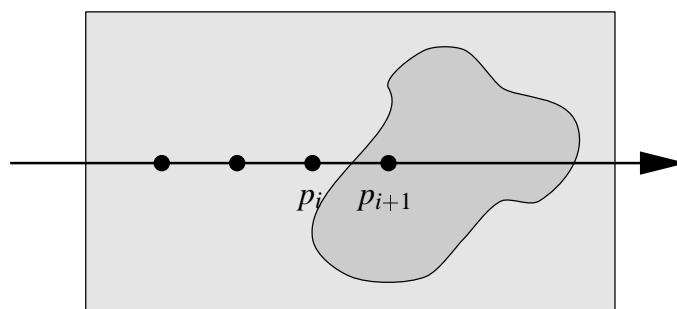


Abbildung 2.2: Abtasten in regelmäßigen Abständen

vexen) Polyeder entspricht, dessen Eckpunkte Gitter- bzw. Samplepunkte im Volumen sind. In jeder Zelle kann dann, basierend auf der verwendeten Interpolationsfunktion (trilinear, inverse Distanzgewichtung, etc.) und der parametrischen Form des Strahls, analytisch berechnet werden, ob und wenn ja, wo der Strahl die Isofläche schneidet. Der Vorteil dieses Ansatzes besteht darin, daß gemäß der Interpolationsfunktion der genaue Schnittpunkt berechnet werden kann. Allerdings können die Kosten der analytischen Berechnung für komplexe Interpolationsfunktionen recht hoch sein. Alternativ könnte der Schnittpunkt natürlich auch einfach durch lineare Interpolation zwischen den beiden Endpunkten des Strahlsegments innerhalb einer Zelle approximiert werden.

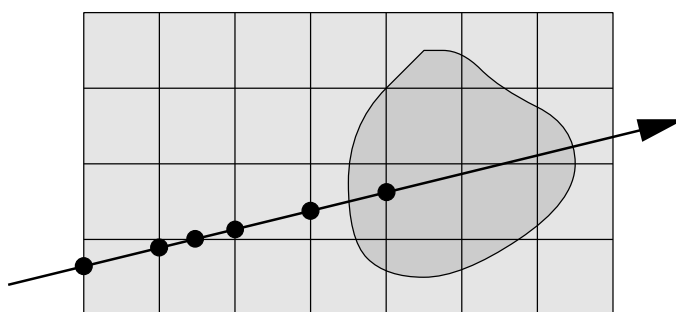


Abbildung 2.3: Traversierung der Zellen

Zu beachten ist, daß es insbesondere bei ersterem Ansatz im Rahmen eines Ray-Tracers zu Aliasing-Problemen kommen kann. Werden ausgehend von dem

approximierten Schnittpunkt Schatten- bzw. Sekundärstrahlen in die Szene geschickt, so könnte fälschlicherweise im Ursprung des Strahls eine Blockierung detektiert werden. Dieses Problem kann auf drei Arten verringert bzw. behoben werden [Gro96]:

- Es werden nur solche Schnittpunkte p_{thres} als gültig qualifiziert, die einen Mindestabstand $|p_{thres} - O| > \epsilon_{min}$ zum Strahlursprung O haben.
- Die Generierung von Schatten- und Sekundärstrahlen wird unterbunden.
- Die Isofläche wird zunächst durch Polygone approximiert und das daraus resultierende Polygonnetz dann mittels des Ray-Tracers dargestellt.

2.2.2 Polygonale Approximation

Im folgenden soll exemplarisch auf den von Lorensen & Cline vorgestellten *Marching Cubes Algorithmus* [LC87] eingegangen werden, welcher als eine Art Standardverfahren zur polygonalen Approximation bzw. zur Extraktion von Isoflächen bezeichnet werden kann. Der Algorithmus geht davon aus, daß die Daten in Form eines regulären Gitters gegeben sind, und nutzt dabei die Aufteilung des Volumens in Würfel, bzw. Zellen, welche jeweils durch acht benachbarte Gitterpunkte definiert sind. Das Prinzip besteht darin, das Volumen Zelle für Zelle zu durchwandern und für jede Zelle, die von der Isofläche geschnitten wird, die darin enthaltene Teilfläche durch Dreiecke zu approximieren. Das Ergebnis nach Abarbeitung aller Zellen ist dann eine polygonale Approximation in Form einer Dreiecksliste, welche mittels konventioneller Techniken dargestellt werden kann.

Grundlegend für das Verfahren ist die Überlegung von Lorensen & Cline, wie eine Fläche eine Zelle schneiden kann und wie sie innerhalb der Zelle zu approximieren ist. Unterschieden wird dafür zunächst zwischen Eckpunkten, deren Datenwert kleiner als der Schwellwert ist und Eckpunkten, deren Datenwert größer oder gleich dem Schwellwert ist. Die Eckpunkte werden diesbezüglich als *Weiß* bzw. *Schwarz* bezeichnet. Die Autoren gehen nun davon aus, daß in einer Zelle genau die Kanten von einer Fläche geschnitten werden, deren Eckpunkte verschiedene Farben haben und daß dann jeweils nur ein Schnitt mit diesen Kanten existiert. Die entsprechenden Schnittpunkte, welche durch lineare Interpolation berechnet

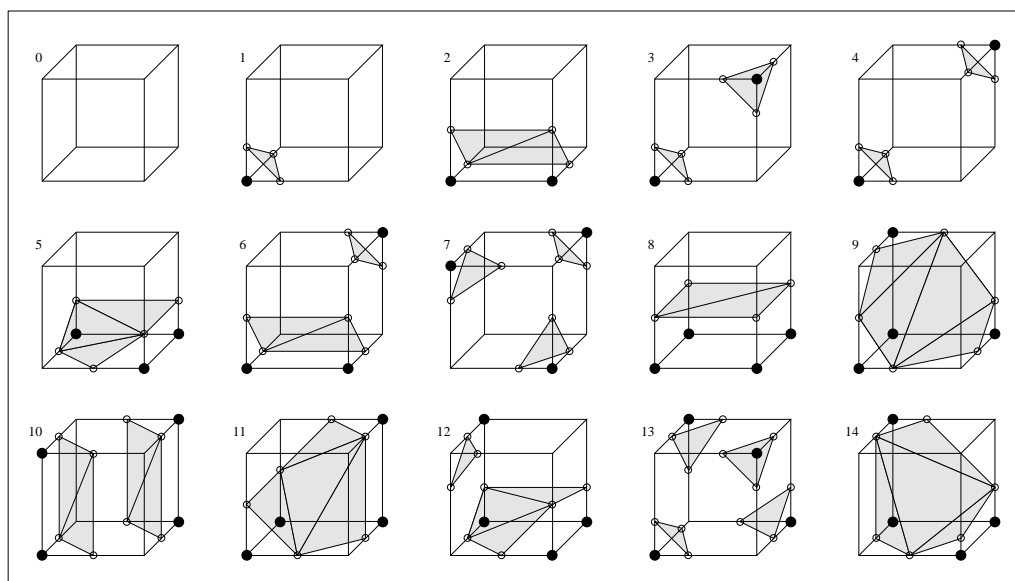


Abbildung 2.4: Die 15 Basis-Konfigurationen des Marching Cubes Algorithmus

werden können, sind dann so durch Kanten zu verbinden, daß die daraus resultierenden Polygone jeweils die weißen und die schwarzen Eckpunkte separieren. Das Ergebnis ist dann eine polygonale Approximation der in der Zelle enthaltenen Teilfläche.

Gemäß den acht Eckpunkten einer Zelle und den zwei möglichen Farben für jeden Eckpunkt, gibt es insgesamt $2^8 = 256$ mögliche Konfigurationen für eine Zelle, d.h. 256 Möglichkeiten, wie eine Fläche eine Zelle schneiden kann. Durch Ausnutzung von Symmetrie-Eigenschaften lassen sich diese Konfigurationen auf 15 topologisch verschiedene reduzieren, welche in Abbildung 2.4 dargestellt sind. Die einfachste Konfiguration, 0, ist dann gegeben, wenn die Eckpunkte entweder alle schwarz oder alle weiß sind. Konfiguration 1 tritt auf, wenn nur ein Eckpunkt schwarz bzw. nur ein Eckpunkt weiß ist. In diesem Fall beinhaltet die Zelle eine Fläche, welche den einen Eckpunkt von den restlichen trennt und durch ein Dreieck approximiert wird. Die weiteren Konfigurationen sind analog zu betrachten. Ausgehend von diesen 15 Basis-Konfigurationen kann eine Look-Up-Tabelle konstruiert werden, in welcher für jede der 256 Konfigurationen die entsprechende Flächentopologie gespeichert ist.

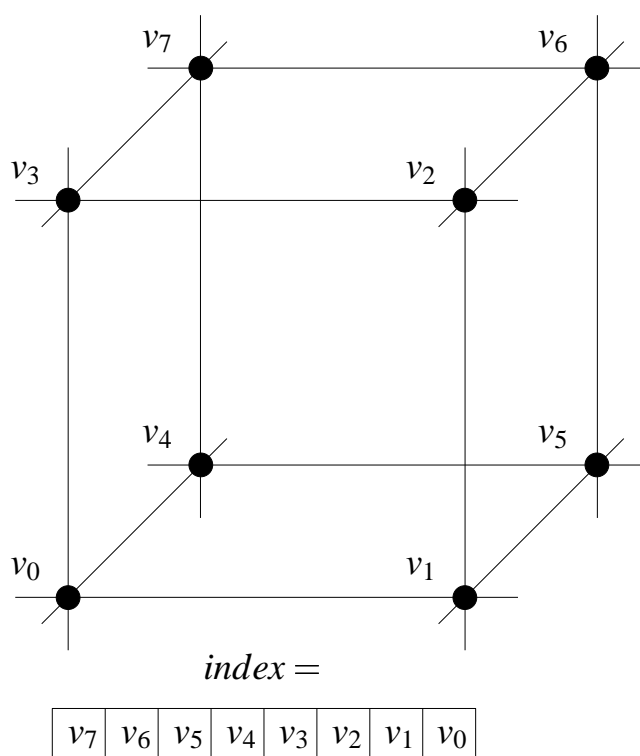


Abbildung 2.5: Index für den Zugriff auf die Look-Up-Tabelle

Für die Approximation einer Fläche innerhalb einer Zelle sind somit folgende Schritte auszuführen: Zunächst erfolgt eine Klassifizierung der Eckpunkte bzgl. des gegebenen Schwellwertes. Daraus ergibt sich ein achtstelliger binärer Vektor, welcher als Zahl interpretiert wird und als Index für den Zugriff auf die vorberechnete Look-Up-Tabelle dient (siehe Abbildung 2.5). Gemäß den in der Tabelle enthaltenen topologischen Informationen werden dann für die entsprechenden Kanten der Zelle die Schnittpunkte durch lineare Interpolation berechnet und die nötigen Dreiecke generiert. Lorensen & Cline berechnen desweiteren die Normalvektoren für jeden Schnittpunkt. Alternativ können aber auch die durch die Dreiecke implizit gegebenen Vektoren benutzt werden, was allerdings einen Qualitätsverlust zur Folge hat.

Der Marching Cubes Algorithmus besticht zwar durch seine Einfachheit, weist aber auch einige Schwachstellen auf. So sind zum Beispiel die 15 Basis-Konfigu-

rationen unvollständig in dem Sinne, daß Oberflächen generiert werden, die unter Umständen Lücken oder Löcher beinhalten können [Due88]. Wilhelms & van Gelder [WvG90] haben für dieses Problem verschiedene Lösungsansätze vorgestellt, welche aber zu teilweise ungewollten bzw. überflüssigen Flächenelementen führen. Eine weitere Schwachstelle besteht darin, daß für jede Zelle separat die Polygone erzeugt werden und somit keine Reduktion der Polygone in Abhängigkeit der globalen Flächenstruktur berücksichtigt wird. Müller & Stark [MS93] haben diesbezüglich einen alternativen Algorithmus zur adaptiven Generierung von Isoflächen vorgestellt, bei welchem Größe und Anzahl der Polygone an die Form der Fläche angepaßt werden.

2.3 Integrationsbasierte Visualisierung

Ein wesentlicher Nachteil bei der oben beschriebenen Darstellung von Isoflächen besteht darin, daß gemäß eines Schwellwertes eine binäre Klassifizierung der Daten gegeben ist und somit amorphe Strukturen, sowie atmosphärische Phänomene nicht adäquat dargestellt werden können. Bei integrierenden Verfahren hingegen wird im Prinzip jedes Volumenelement berücksichtigt und eine zweidimensionale Darstellung der Volumendaten durch Integration von Intensitätswerten berechnet. Grundlage für diese Verfahren sind physikalisch basierte Modelle, welche von der Transportgleichung der Transporttheorie abgeleitet sind und die Gesetzmäßigkeiten für die Interaktion des Lichts mit einem *partizipierenden Medium* beschreiben. Im folgenden sollen diese Modelle erläutert und danach auf die Darstellung atmosphärischer Phänomene sowie auf die flächenorientierte Visualisierung eingegangen werden.

2.3.1 Die Transportgleichung

Die hier vorgestellte Transportgleichung ist ein aus der Transporttheorie stammendes, von der *linearen Boltzmann-Gleichung* abgeleitetes, Modell zur Simulation des Partikeltransports in einem partizipierenden Medium. Glassner [Gla95], Cohen & Wallace [CW93] und Siegel & Howell [SH92] beinhalten Einführungen in die Transporttheorie sowie eine detailliertere Betrachtung der zugrundeliegenden Physik. Hier sollen nur die im Rahmen dieser Arbeit relevanten Ergebnisse auf-

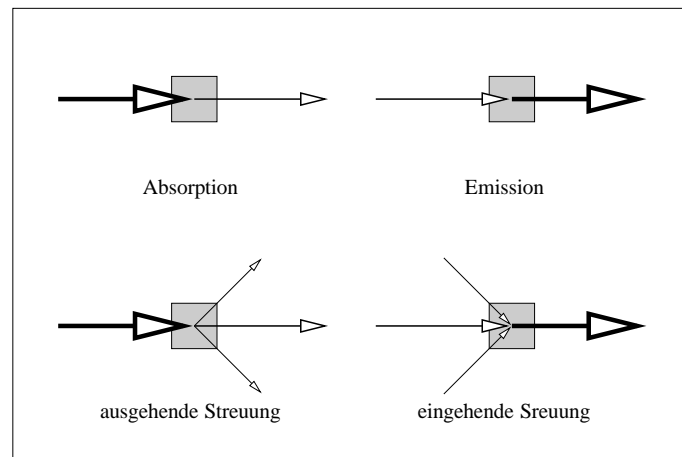


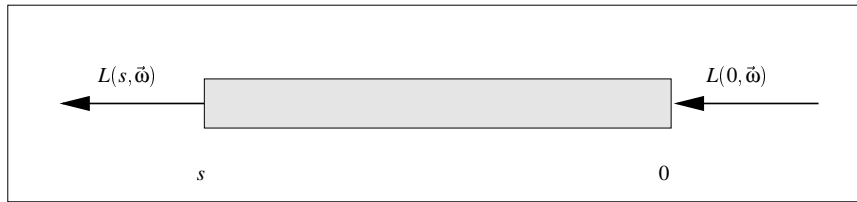
Abbildung 2.6: Phänomene in einem partizipierenden Medium

geführt werden, wobei in den Grundzügen einer in [Lac95] gegebenen Darstellung gefolgt wird.

Ein partizipierendes Medium kann als ein semitransparenter Block betrachtet werden, in welchem Partikel bzw. Photonen emittiert, absorbiert und gestreut werden (siehe Abbildung 2.6). Die Partikelanzahl entspricht dabei einer Energie, welche im Falle von Photonen als Lichtintensität interpretiert wird. Die Transportgleichung besagt nun, daß die Änderung der Intensität $L(\mathbf{x}, \vec{\omega})$ an einem Punkt \mathbf{x} entlang einer Richtung $\vec{\omega}$ gleich der Summe aus abgeschwächtem, emittiertem und gestreutem Licht ist. Zu beachten ist, daß die hier genannten Größen wellenlängenabhängig sind, und somit die Gleichung typischerweise für die drei Farbkanäle (rot, grün, blau) getrennt betrachtet werden muß:

$$\begin{aligned} \vec{\omega} \cdot \nabla L(\mathbf{x}, \vec{\omega}) &= -\sigma_t(\mathbf{x})L(\mathbf{x}, \vec{\omega}) \\ &+ \sigma_a(\mathbf{x})L_E(\mathbf{x}, \vec{\omega}) \\ &+ \sigma_s(\mathbf{x}) \int_{\Omega} \rho(\mathbf{x}, \vec{\omega}' \rightarrow \vec{\omega})L(\mathbf{x}, \vec{\omega}')d\vec{\omega}'. \end{aligned} \quad (2.1)$$

- $\sigma_t(\mathbf{x}) = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x})$ ist der Dämpfungskoeffizient, welcher gleich der Summe aus Absorptionskoeffizient und Streukoeffizient des Mediums am Punkt \mathbf{x} ist. Er beschreibt die Wahrscheinlichkeit pro Einheitsdistanz, daß ein Photon entlang des Pfades $\vec{\omega}$ entweder absorbiert oder in eine ande-

Abbildung 2.7: Integration über einen Pfad der Länge s

re Richtung gestreut wird.

- $L_E(\mathbf{x}, \vec{\omega})$ ist die Emissions-Funktion. Sie beschreibt, wieviel Photonen am Punkt \mathbf{x} des Mediums in Richtung $\vec{\omega}$ emittiert werden. Die Multiplikation mit dem Absorptionskoeffizienten $\sigma_a(\mathbf{x})$ ist eine Konsequenz von Reziprozitäts-Prinzipien aus der Thermodynamik [SP94].
- $\rho(\mathbf{x}, \vec{\omega})$ ist die Streu- oder Phasen-Funktion, welche die Wahrscheinlichkeit pro Einheitsraumwinkel pro Einheitsdistanz angibt, daß ein Photon aus Richtung $\vec{\omega}' \in \Omega$ am Punkt \mathbf{x} in Richtung $\vec{\omega}$ gestreut wird.

Gleichung 2.1 ist eine Differentialgleichung erster Ordnung, die durch Integration über einen Pfad der Länge s (siehe Abbildung 2.7) in folgende äquivalente integrale Form überführt werden kann:

$$L(s, \vec{\omega}) = e^{-t(0,s)} L(0, \vec{\omega}) + \int_0^s e^{-t(s',s)} Q(s', \vec{\omega}) ds'. \quad (2.2)$$

- $t(u, u')$ ist das Integral des Dämpfungskoeffizienten entlang des geradlinigen Pfades zwischen u und u' :

$$t(u, u') = \int_u^{u'} \sigma_t(u'') du''.$$

- $L(0, \vec{\omega})$ ist die Intensität des am Anfang des Pfades in Richtung $\vec{\omega}$ einfallenden Lichts.

- $Q(u, \vec{\omega})$ ist die Summe aus Emissions- und Streuungs-Term:

$$Q(u, \vec{\omega}) = \sigma_a(u)L_E(u, \vec{\omega}) + \int_{\Omega} \sigma_s(u)\rho(u, \vec{\omega}' \rightarrow \vec{\omega})L(u, \vec{\omega}')d\vec{\omega}'. \quad (2.3)$$

Diese Funktion wird in der Literatur auch als *Quellfunktion* bezeichnet.

Die integrale Form der Transportgleichung besagt nun, daß sich die Intensität an einem Punkt entlang eines Pfades aus dem am Anfang des Pfades einfallenden Licht und dem im Volumen entlang des Pfades emittierten und gestreuten Licht zusammensetzt. Das Licht wird dabei jeweils - in Abhängigkeit der Entfernung, die es im Medium zurücklegt - exponentiell abgeschwächt. Man beachte den auf beiden Seiten der Gleichung stehenden Term $L(s, \vec{\omega})$, wodurch eine Rekursion und damit auch die Komplexität der Gleichung gegeben ist.

Gleichung 2.2 ist eine Verallgemeinerung einer Vielzahl bekannter Beleuchtungs- und Visualisierungsmodelle aus der Computergrafik. So ist zum Beispiel Kajiya's bekannte *Rendering-Gleichung* [Kaj86] eine auf den Lichttransport zwischen Oberflächen spezialisierte Form der Transportgleichung.

Eine mögliche Methode zur Lösung von Gleichung 2.2 ist das *Zonal-Radiosity-Verfahren* [RT87], bei welchem allerdings davon ausgegangen wird, daß Emissions- und Streu-Funktionen *isotropisch*, d.h. von der Richtung $\vec{\omega}$ unabhängig sind.

2.3.2 Approximationen der Transportgleichung

Insbesondere im Bereich der wissenschaftlich orientierten Visualisierung geht das Bestreben weniger dahin, eine physikalisch korrekte, als vielmehr eine schnelle und informationsreiche Darstellung der Daten zu erhalten. Da die Auswertung der Transportgleichung in der Regel zu zeitaufwendig ist, gehen die meisten Verfahren von der vereinfachenden Annahme aus, daß das darzustellende Medium einen geringen Reflexionsgrad (Albedo) hat, sodaß nur Einfach-Streuungen auftreten. D.h. alle Photonen, die von einer Lichtquelle ausgehen, werden auf dem Weg zur Bildebene nur einmal im Volumen gestreut. Gemäß dieser Annahme wird das Integral des Streuungs-Terms oft durch eine Summe über eine endliche Anzahl externer Lichtquellen approximiert, wodurch der Quellterm 2.3 folgende verein-

fachte Form annimmt:

$$Q(u, \vec{\omega}) = \sigma_a(u)L_E(u, \vec{\omega}) + \sum_i \sigma_s(u)\rho(u, \vec{\omega}_i \rightarrow \vec{\omega})L_i, \quad (2.4)$$

wobei L_i der Intensität von Lichtquelle i und $\vec{\omega}_i$ der Richtung des von Lichtquelle i einfallenden Lichts entspricht. Für die adäquate Darstellung von Schatten muß auch noch für das von den Lichtquellen einfallende Licht entlang des Pfades, den es im Volumen zurücklegt, eine exponentielle Abschwächung berücksichtigt werden.

Berechnet werden können die verbleibenden Integrale generell durch numerische Integration, z.B. durch Anwendung der Trapez-Formel oder durch Bildung der Riemannschen Summe. Eine weitere bekannte Methode, auf die hier näher eingegangen werden soll, basiert auf der Tatsache, daß für einen konstanten Quellterm $Q(u, \vec{\omega})$ und einen konstanten Dämpfungskoeffizienten $\sigma_t(u)$, die Intensität $L(u, \vec{\omega})$ durch

$$\begin{aligned} L(u, \vec{\omega}) &= e^{-t(0,u)}L(0, \vec{\omega}) + \int_0^u e^{-t(u',u)}Q(u', \vec{\omega})du' & (2.5) \\ &= e^{-\int_0^u \sigma_t u}L(0, \vec{\omega}) - Q/\sigma_t \int_0^u -\sigma_t e^{-\int_{u'}^u \sigma_t du''} du' \\ &= e^{-\int_0^u \sigma_t u}L(0, \vec{\omega}) - Q/\sigma_t \int_0^u \frac{d}{du'} e^{-\int_{u'}^u \sigma_t du''} du' \\ &= e^{-\sigma_t u}L(0, \vec{\omega}) + (1 - e^{-\sigma_t u})Q/\sigma_t \end{aligned}$$

analytisch exakt berechnet werden kann. Einige Autoren wie z.B. [Sab88] vernachlässigen die Streuung bzw. modellieren diese mittels des Emissions-Terms. Unter diesen Voraussetzungen (d.h. $\sigma_s = 0$, $\sigma_t = \sigma_a$) vereinfacht sich obige Gleichung zu

$$L(u, \vec{\omega}) = e^{-\sigma_a u}L(0, \vec{\omega}) + (1 - e^{-\sigma_a u})L_E. \quad (2.6)$$

Diese Ableitung entspricht auch dem einfachen Modell, mit welchem Nebefeffekte in konventionellen Visualisierungssystemen erzielt werden.

Um nun für nicht konstante Q und σ_t die Intensität zu berechnen, kann der Pfad in Segmente unterteilt werden, auf welche jeweils obige Lösung 2.5 angewendet wird. Für n Segmente der Länge $\Delta s = s/n$ ergibt sich dann mit folgenden

Ersetzungen bzw. Vereinfachungen

$$\begin{aligned}\alpha_i &= 1 - e^{-\sigma_t(i\Delta s)\Delta s} && \text{die Opazität von Segment } i \\ c_i &= Q(i\Delta s, \vec{\omega}) / \sigma_t(i\Delta s) && \text{die Farbe von Segment } i \\ c_0 &= L(0, \vec{\omega}) && \text{die Hintergrundfarbe}\end{aligned}$$

die sogenannte *volumetrische Kompositionsapproximation*:

$$\begin{aligned}L(s, \vec{\omega}) &\approx c_0 \prod_{i=1}^n (1 - \alpha_i) + \sum_{i=1}^n \alpha_i c_i \cdot \prod_{j=i+1}^n (1 - \alpha_j) && (2.7) \\ &= c_0 (1 - \alpha_1) \cdots (1 - \alpha_n) \\ &\quad + \alpha_1 c_1 (1 - \alpha_2) \cdots (1 - \alpha_n) + \\ &\quad \cdots + \alpha_{n-1} c_{n-1} (1 - \alpha_n) + \alpha_n c_n.\end{aligned}$$

Damit lassen sich direkt der bekannte *Back-to-Front*-

```
color = c[0];
for ( i = 1; i <= n; i++ )
    color = (1-a[i])*color + a[i]*c[i];
```

und der äquivalente *Front-to-Back*-Kompositionsalgorithmus

```
color = 0.0; alpha = 1.0; i = n;
while ( alpha > small.threshold && i >= 1 )
{
    color = alpha*a[i]*c[i] + color;
    alpha = alpha*(1-a[i]);
    i--;
}
if ( i == 0 )
    color = color + alpha*c[0];
```

formulieren [Max95]. Letzterer bietet den Vorteil, die Komposition bei Erreichen einer bestimmten Opazität in Abhängigkeit eines vom Benutzer vorgegebenen Schwellwertes abbrechen zu können.

2.3.3 Abbildung skalarer Daten auf Modellparameter

Für die Anwendung eines der genannten Modelle zur Visualisierung dreidimensionaler Datensätze gilt es zunächst festzulegen, wie die Daten auf die Modellparameter abzubilden sind, d.h. welche optischen Eigenschaften das darzustellende Medium haben soll. Im folgenden soll diesbezüglich auf zwei bekannte, im Rahmen dieser Arbeit implementierte Ansätze, eingegangen werden, welche zur Darstellung atmosphärischer Phänomene bzw. zur Darstellung von innerhalb des Volumens befindlichen Konturflächen geeignet sind.

Atmosphärische Phänomene

Blinn [Bli82] sowie Kajiya & von Herzen [KvH84] haben ein auf Gleichung 2.2 basierendes Modell vorgestellt, welches sie zur Visualisierung atmosphärischer Phänomene, wie Rauch, Nebel und Dunst benutzt haben. Sie gehen davon aus, daß das Medium aus einer Menge sphärischer Partikel gleicher Größe besteht, welche Licht absorbieren und streuen. Ein Skalarwert $f(\mathbf{x})$ an einem Punkt \mathbf{x} im Volumen wird dabei als Partikeldichte interpretiert. Gemäß diesem Modell sind Dämpfungs- und Streukoeffizient durch

$$\begin{aligned}\sigma_t(\mathbf{x}) &= \tau f(\mathbf{x}) \\ \sigma_s(\mathbf{x}) &= f(\mathbf{x})\end{aligned}$$

gegeben, wobei τ eine beliebige Skalierungs-Konstante zur Festlegung der “optischen Tiefe” des Mediums ist. Die Emission wird in diesem Modell vernachlässigt und die Phasen- oder Streu-Funktion ist eine der folgenden physikalisch basierten Funktionen, welche in Abhängigkeit des darzustellenden Mediums zu wählen sind:

$$\begin{aligned}\rho_{\text{Rayleigh}} &= \frac{3}{4}(1 + (\vec{\omega}' \cdot \vec{\omega})^2) \\ \rho_{\text{HazyMie}} &= 1 + 9 \left(\frac{1 + (\vec{\omega}' \cdot \vec{\omega})}{2} \right)^8 \\ \rho_{\text{MurkyMie}} &= 1 + 50 \left(\frac{1 + (\vec{\omega}' \cdot \vec{\omega})}{2} \right)^{32}.\end{aligned}$$

Die Raleigh–Funktion eignet sich besonders für die Darstellung von Rauch und Staub, die Mie–Funktionen für die Darstellung von Dunst ($\rho_{HazyMie}$) und Nebel ($\rho_{MurkyMie}$).

Konturflächen

Die in der Literatur bekannten Verfahren zur Darstellung von Konturflächen basieren in der Regel auf Gleichung 2.7. Mittels sogenannter *Transferfunktionen* werden die gegebenen Daten auf Opazitäten und ggf. weitere Materialparameter abgebildet und unter Verwendung eines Phong–ähnlichen Beleuchtungsmodells wird an jedem Samplepunkt eine Intensität bzw. Farbe berechnet. Die für die Beleuchtungsrechnung notwendigen Normalvektoren werden dabei jeweils durch den lokalen Gradienten approximiert. Grundlegend für die Visualisierung ist die Wahl der Transferfunktion mit der eine Klassifizierung der Daten realisiert wird. Levoy [Lev88] hat in diesem Zusammenhang ein Standardverfahren entwickelt, welches für die Visualisierung medizinischer Daten gedacht ist aber auch auf Datensätze aus anderen Bereichen angewendet werden kann. Er macht dabei folgende Annahmen über das darzustellende Volumen:

- Das Volumen repräsentiert eine beliebige Anzahl von Gewebeschichten, die jeweils einen charakteristischen Skalarwert besitzen, der a priori bekannt sein muß.
- Jede Gewebeschicht grenzt höchstens an zwei andere Schichten.
- Sind die Schichten nach ihren charakteristischen Skalarwerten geordnet, so grenzt eine Schicht im Volumen nur an die ihr in der gegebenen Ordnung benachbarten Schichten.

Formal ausgedrückt, bedeutet dies für n Gewebeschichten mit Skalarwerten

$$f_{v_1} < f_{v_2} < \dots < f_{v_n},$$

daß eine Schicht i nicht an eine Schicht j grenzt, wenn $|i - j| > 1$. Treffen diese Kriterien zu, kann jeder Gewebeschicht eine Opazität zugeordnet und eine stückweise lineare Abbildung konstruiert werden, mit welcher Skalarwerte f_{v_i} auf Opazitätswerte α_{v_i} und dazwischenliegende Skalarwerte auf dazwischenliegende Opa-

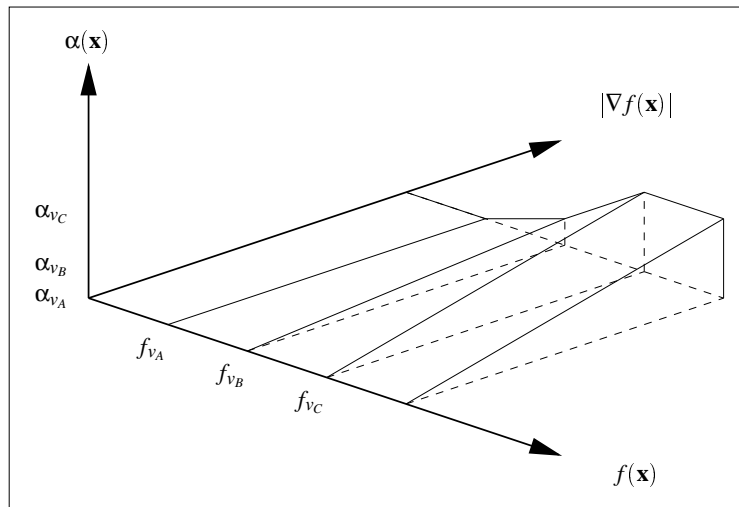


Abbildung 2.8: Ein Graph von $\alpha(\mathbf{x})$ für drei Gewebeschichten

zitätswerte abgebildet werden. Weiterhin ist es wünschenswert, Übergänge zwischen verschiedenen Gewebeschichten hervorzuheben und die inneren Bereiche der Schichten bei der Darstellung zu unterdrücken, um somit u.a. auch “verdeckte” Strukturen sichtbar zu machen. Erreicht werden kann dies, indem die berechnete Opazität durch die (normierte) Länge des lokalen Gradienten skaliert wird. Die resultierende Transferfunktion hat dann folgende Form:

$$\alpha(\mathbf{x}) = |\nabla f(\mathbf{x})| \begin{cases} \alpha_{v_{i+1}} \left(\frac{f(\mathbf{x}) - f_{v_i}}{f_{v_{i+1}} - f_{v_i}} \right) & \text{wenn } f_{v_i} \leq f(\mathbf{x}) \leq f_{v_{i+1}} \\ + \alpha_{v_i} \left(\frac{f_{v_{i+1}} - f(\mathbf{x})}{f_{v_{i+1}} - f_{v_i}} \right) & \\ 0 & \text{sonst} \end{cases}$$

für $i = 1, \dots, n - 1, n \geq 1$. Ein Graph von $\alpha(\mathbf{x})$ als Funktion von $f(\mathbf{x})$ und $|\nabla f(\mathbf{x})|$ für drei Gewebeschichten A, B, C mit charakteristischen Skalarwerten $f_{v_A}, f_{v_B}, f_{v_C}$ und zugeordneten Opazitäten $\alpha_{v_A}, \alpha_{v_B}, \alpha_{v_C}$ ist zur Veranschaulichung in Abbildung 2.8 dargestellt.

Um verschiedene Gewebeschichten farblich hervorzuheben, können analog zu den Opazitäten die Skalarwerte zusätzlich auch auf Farbwerte bzw. Parameter des

zu benutzenden lokalen Beleuchtungsmodells abgebildet werden. Ein möglicher Ansatz, der im Rahmen dieser Arbeit implementiert wurde, basiert auf einem Vorschlag von Upson & Keeler [UK88] und besteht darin, einen Skalarwert auf eine Objektfarbe abzubilden, die im Rahmen eines vereinfachten Phong-Modells als diffuse Farbkomponente benutzt wird.

$$I_\lambda = I_{a,\lambda} k_a O_{d,\lambda}(f(\mathbf{x})) + \sum_i I_{l,\lambda,i} k_d O_{d,\lambda}(f(\mathbf{x}))(N \cdot L_i), \quad \lambda \in \{r, g, b\}.$$

- I_λ ist die Intensität des reflektierten Lichts. Die Farbe c aus Gleichung 2.7 entspricht I .
- $I_{a,\lambda}$ ist die Intensität des ambienten Lichts.
- k_a ist der ambiente Reflexionskoeffizient.
- $O_{d,\lambda}(f(\mathbf{x}))$ ist die diffuse Objektfarbe, die in Abhängigkeit des Skalarwertes an der Position \mathbf{x} definiert ist.
- k_d ist der diffuse Reflexionskoeffizient.
- $I_{\lambda,l,i}$ ist die Intensität von Lichtquelle i .
- N ist der durch den Gradienten approximierte Normalvektor.
- L_i ist der Vektor zur Lichtquelle.

Auf den spiegelnden Term wurde verzichtet, da dieser nur relativ wenig zur Aussagekraft des visualisierten Datensatzes beiträgt.

2.3.4 Visualisierungstechniken

Die gesamte Bandbreite der aus der Literatur bekannten Techniken zur integrationsbasierten Visualisierung dreidimensionaler Datensätze lässt sich grob in folgende fünf Ansätze unterteilen:

- Radiosity mit partizipierenden Medien
- Cell Projection

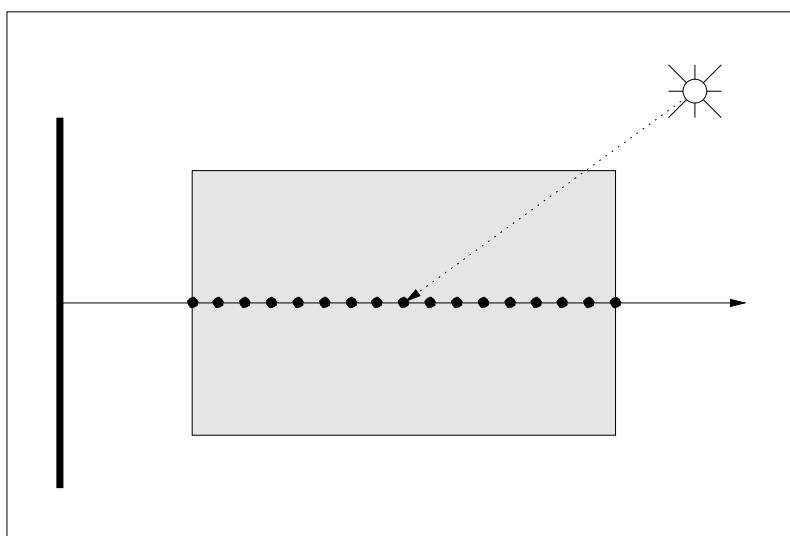


Abbildung 2.9: Visualisierung mittels Ray Integration

- Splatting
- Ray Integration
- Texturbasierte Darstellung

Abgesehen von den Radiosity-Verfahren, bei welchen immer eine Mehrfach-Streuung des Mediums berücksichtigt wird, können mit den übrigen Verfahren qualitativ gleichwertige Ergebnisse erzielt werden. Die beiden Charakteristika, durch welche sich die Verfahren voneinander unterscheiden ist die Art der Daten-traversierung und die Methode, mit welcher die Daten auf die Bildebene projiziert werden. Da der Schwerpunkt dieser Arbeit darin besteht, Volumina im Rahmen des Ray-Tracers und des Polygon-Renderers zu visualisieren, soll im folgenden lediglich auf die letzten beiden Ansätze genauer eingegangen werden.

Ray Integration

Algorithmen, die auf dem Ray-Integration-Ansatz basieren, erzeugen eine zweidimensionale Darstellung eines Volumens, indem für jeden Bildpunkt ein Strahl durch das Volumen geschickt wird und entlang dieses Strahls Farb- und Opazitätswerte aufintegriert werden. Wie in Abbildung 2.9 gezeigt, werden die

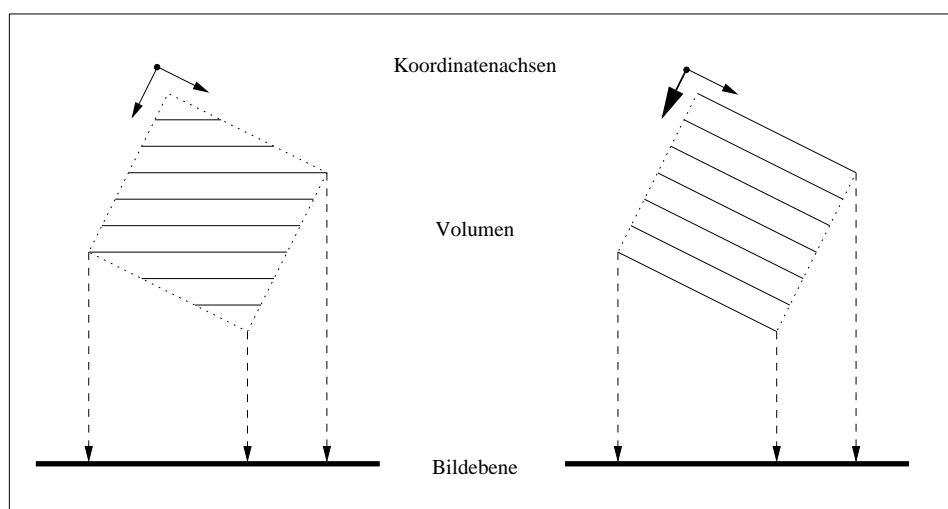


Abbildung 2.10: Zwei Ansätze zur texturbasierten Darstellung

Daten dabei typischerweise in regelmäßigen Abständen abgetastet und die für jeden Punkt berechneten Farb- und Opazitätswerte mittels des in Abschnitt 2.3.2 genannten *Front-to-Back*-Verfahrens aufintegriert bis die Opazität einen vorgegebenen Schwellwert erreicht hat. Diese Technik ist einfach zu implementieren und eignet sich offensichtlich sehr gut für eine Intergration in ein Ray-Tracing-System. Im Vergleich zu anderen Techniken (Cell Projection, Splatting) besitzt sie allerdings den Nachteil, daß auf die Daten nicht in “storage order” zugegriffen wird und somit Kohärenzeigenschaften von Gitterstrukturen weniger gut ausgenutzt werden können.

Texturbasierte Darstellung

Bestimmte Grafikbibliotheken zur polygonbasierten Flächen-Visualisierung (wie z.B. OpenGL [Sil93]) erlauben es unter Ausnutzung spezieller Hardware, einzelnen Flächen transparente Texturen zuzuordnen. Jedes Element einer Textur (Texel) kann dabei neben den drei RGB-Farbwerten auch einen Alpha-Wert beinhalten, welcher die Transparenz bzw. Opazität für einen Punkt auf einer darzustellenden Fläche definiert. Bei der Abbildung der Szene auf die Bildebene werden dann unter Verwendung eines sogenannten Alpha-Puffers die RGBA-Komponenten “hintereinanderliegender” Flächen mittels des in Abschnitt 2.3.2 ge-

nannten Front-to-Back-Verfahrens akkumuliert.

Gemäß diesen Voraussetzungen lassen sich Volumina auch im Rahmen eines konventionellen Polygon-Renderers “scheibchenweise” durch eine Folge von entsprechend texturierten Flächen darstellen. Jede Fläche, typischerweise ein planares Rechteck, repräsentiert dabei einen Schnitt durch das Volumen. Für eine adäquate Darstellung sollten diese Schnittflächen in Abhängigkeit des Augvektors gewählt werden. Abbildung 2.10 zeigt diesbezüglich zwei mögliche Ansätze. Der erste Ansatz besteht darin, das Volumen durch senkrecht zum Augvektor stehende Flächen darzustellen. Der zweite Ansatz besteht darin, das Volumen durch Flächen gleicher Größe darzustellen, die entlang jener Achse im Volumen gestaffelt sind, welche den kleinsten Winkel mit dem Augvektor bildet, d.h. für welche das Skalarprodukt mit dem Augvektor am größten ist. Dieser zweite Ansatz bietet den Vorteil, daß die Flächen leicht zu bestimmen sind und ggf. eine den Daten zugrundeliegende reguläre Gitterstruktur besser ausgenutzt werden kann.

Kapitel 3

Das MRT

Im Rahmen dieses Kapitels soll nach einer Einführung in die Architektur des ursprünglichen, flächenorientierten Systems auf die durchgeführten strukturellen Veränderungen und Erweiterungen sowie auf die konkrete Integration der in Kapitel 2 genannten Verfahren zur Volumenvisualisierung eingegangen werden. Wie einleitend zu dieser Arbeit erwähnt, wurde insbesondere darauf geachtet, einen Großteil der vorhandenen Konzepte zu nutzen und eine Basis für weitere Verfahren zur Volumenvisualisierung zu schaffen.

3.1 Architektur

Das MRT ist unter der Zielsetzung entstanden, ein möglichst flexibles und leicht erweiterbares System zu schaffen. Erreicht wurde dies durch den gewählten *objektbasierten* Ansatz und durch konsequente Umsetzung des objektorientierten Paradigmas. Das System kann dabei als eine Menge miteinander kommunizierender Objekte verstanden werden, die jeweils eine wohldefinierte Menge von Operationen ausführen können. Die diese Objekte repräsentierenden Basisklassen und deren Beziehungen zueinander sind in Abbildung 3.1 wiedergegeben.

Eine darzustellende Szene (`t_IllumScene`) besteht aus einer Menge von Lichtquellen (`t_Light`) sowie aus einem in der Regel hierarchisch aufgebauten geometrischen Szenenobjekt (`t_Object` bzw. `t_Scene`). Jedem primitiven Szenenobjekt (Kugel, Zylinder, etc.) ist eine Oberflächenbeschreibung (`t_Surface`) zugeordnet, welche das Material bzw. die optischen Eigenschaften des Objektes

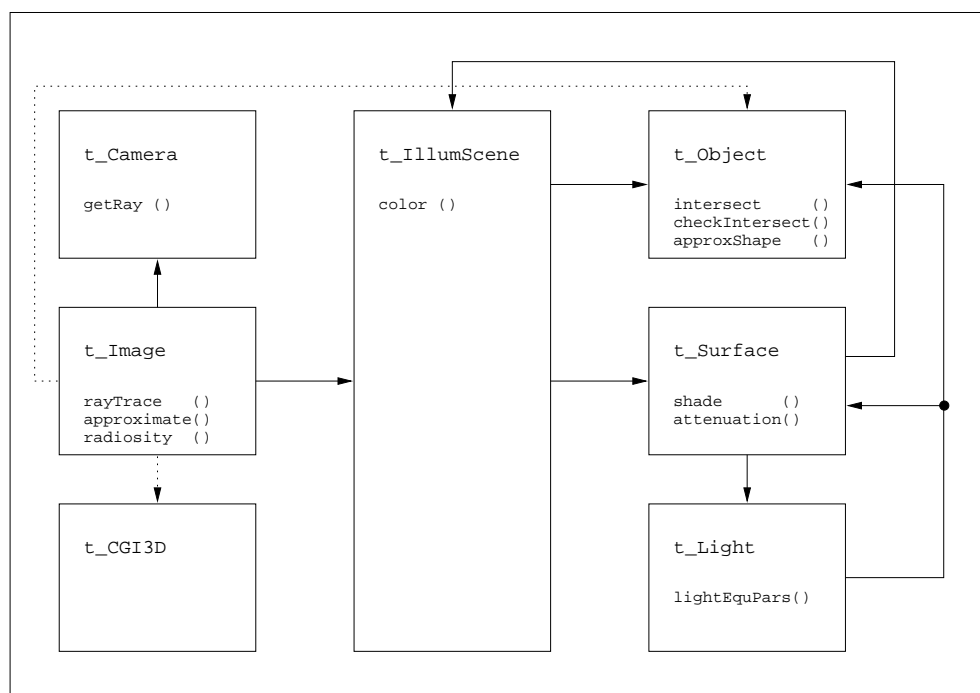


Abbildung 3.1: Komponenten des MRT

definiert. Die Kamera (`t_Camera`) legt fest, wie die Szene auf die Bildebene zu projizieren ist, das (virtuelle) Bild (`t_Image`), wie eine zweidimensionale Darstellung der Szene zu erzeugen ist. Diese kann generell mittels Ray-Tracing, unter Verwendung des Radiosity-Verfahrens oder approximativ unter Verwendung eines lokalen Beleuchtungsmodells visualisiert werden.

Gruppirt nach ihrer Funktionalität sollen im folgenden die einzelnen Klassen und deren Beziehungen zueinander genauer beschrieben werden.

3.1.1 Szenenobjekte

Die Klasse `t_Object` ist die Basisklasse für alle geometrischen Szenenobjekte. Sie verfügt im wesentlichen über folgende virtuelle Methoden:

- `boundingVolume()`
Jedes Objekt kann einen sich selbst einschließenden Hüllkörper berechnen, welcher einem achsenparallelen Quader entspricht.

- `intersect()`
Jedes Objekt kann den Schnitt mit einem Strahl berechnen. Die Methode liefert genau dann den Wert `True`, wenn ein Schnittpunkt existiert, dessen Distanz positiv und kleiner als ein vorgegebener Wert ist. In diesem Fall wird die berechnete Entfernung übergeben.
- `checkIntersect()`
Diese Methode ist äquivalent zu `intersect()`, mit dem Unterschied, daß hier nur auf die *Existenz* eines Schnittes geprüft wird.
- `surfaceNormal()`
Bezüglich eines Punktes auf der Oberfläche kann jedes Objekt den entsprechenden Normalvektor berechnen.
- `mapInvers()`
Diese Methode bildet einen Punkt auf der Oberfläche des Objektes in einen zweidimensionalen Texturraum ab, welcher über dem Bereich $[0, 1] \times [0, 1]$ definiert ist. Standardmäßig liefert diese Methode den Wert `False` zurück, um anzuzeigen, daß eine Texturierung nicht unterstützt wird.
- `inside()`
Diese Methode liefert ein Maß für die Position eines Punktes zur Objektoberfläche.
- `approxShape()`
Jedes Objekt kann eine polygonale Approximation seiner Oberfläche berechnen. Die Qualität der Approximation kann dabei durch einen zu übergebenden Qualitätswert flexibel gesteuert werden. Diese Methode wird insbesondere für eine Visualisierung im Rahmen des Polygon-Renderers und im Rahmen des Radiosity-Verfahrens benutzt.

Es ist erwähnenswert, daß nicht nur alle primitiven Szenenobjekte von der Klasse `t_Object` abgeleitet sind, sondern auch die Szene `t_Scene` selbst, womit sich beliebig komplexe Szenenhierarchien leicht aufbauen und handhaben lassen. Die Klasse `t_Scene` dient desweiteren als Basisklasse für eine Reihe verschiedener Beschleunigungsstrukturen, die im Rahmen der ray-tracing-basierten Visualisierung benutzt werden können [Mue97].

3.1.2 Oberflächenbeschreibungen

Jedem primitiven geometrischen Szenenobjekt ist eine Oberflächenbeschreibung zugeordnet, welche das Material bzw. die optischen Eigenschaften des darzustellenden Objektes definiert. Die Basisklasse für alle Oberflächenbeschreibungen ist die Klasse `t_Surface`.

- `shade()`
Diese Methode berechnet unter Verwendung des Phong-Modells die Farbe eines Punktes auf der Oberfläche. Handelt es sich um eine transmittierende oder spiegelnd reflektierende Oberfläche, so wird auch das entlang des reflektierten bzw. transmittierten Strahls einfallende Licht berücksichtigt. Der für die Beleuchtungsrechnung notwendige Normalvektor und die für eine Texturierung notwendigen Texturkoordinaten werden durch Aufruf der entsprechenden Methoden des zugeordneten geometrischen Objekts berechnet.
- `attenuation()`
Diese Methode wird für die Abschwächung von Schattenstrahlen benutzt. Sie berechnet zu einer gegebenen Distanz, die der Strahl im zugeordneten Objekt zurücklegt, eine Abschwächung der Lichtintensität.
- `perturbNormal()`
Diese als `protected` deklarierte Methode bietet für abgeleitete Klassen eine Möglichkeit, gemäß der zu simulierenden Oberfläche, den Normalvektor zu verändern.
- `diffuseCoefficient()`
Analog zu `perturbNormal()` bietet diese Methode die Möglichkeit für abgeleitete Klassen, die diffusen Reflexionseigenschaften der Oberfläche zu verändern.

Abgeleitet von `t_Surface` ist eine Vielzahl weiterer Klassen, die allein durch Redefinition der beiden letztgenannten Methoden, eine Vielzahl verschiedener Oberflächentypen realisieren.

3.1.3 Lichtquellen

Die Basisklasse für alle Lichtquellen ist `t_Light`, welche standardmäßig eine Punktlichtquelle realisiert. Mittels der (virtuellen) Methode `lightEquPars()` wird bezüglich eines Objektes, eines Punktes und eines Normalvektors das auf den Punkt einfallende Licht berechnet. Die Methode liefert genau dann den Wert `False` zurück, wenn der Punkt bzgl. der Lichtquelle im Schatten liegt, ansonsten `True` und die den Punkt erreichende Lichtintensität.

Die Berechnung der Lichtintensität kann auf zweierlei Weise erfolgen. Zum einen kann lediglich getestet werden, ob sich ein blockierendes Objekt zwischen dem Oberflächenpunkt und der Lichtquelle befindet. Ist dies der Fall, so ist der Punkt im Schatten, andernfalls wird der Punkt von der Lichtquelle beleuchtet und deren Intensität wird als Ergebnis der Berechnung übergeben. Die Lichtquelle muß somit nur (mittels der Methode `checkIntersect()` von `t_Object`) testen, ob ein Schnitt mit der Szene existiert.

Zum anderen kann zusätzlich auch eine Lichtabschwächung durch transparente Objekte berücksichtigt werden, wobei hier eine Abschwächung in Abhängigkeit der Distanz, welche der Lichtstrahl in einem Objekt zurücklegt, berechnet wird. Realisiert wird dies, indem mittels der Methode `intersect()` die Schnittintervalle mit den Szenenobjekten ermittelt werden. Für das jeweilige Objekt wird dann eine Abschwächung durch Aufruf der Methode `attenuation()` von der ihm zugeordneten Oberfläche berechnet.

Gemäß obiger Unterscheidung wird auch zwischen den von `t_Light` abgeleiteten Klassen unterschieden. Das heißt Punktlichtquellen, gerichtete Lichtquellen, Spotlights und verteilte Lichtquellen sind jeweils gemäß dem ersten Ansatz als “nicht abschwächende” Lichtquellen und gemäß dem zweiten Ansatz als “abschwächende” Lichtquellen implementiert.

3.1.4 Bildgenerierung

Die Klasse `t_Image` ist die zentrale Einheit, welche die Bildgenerierung kontrolliert. Sie verfügt im wesentlichen über drei (virtuelle) Methoden:

- `rayTrace()`

Diese Methode erzeugt eine Darstellung der Szene unter Anwendung des

Ray-Tracing-Verfahrens. Für jede (abzutastende) Position auf der Bildebene wird ein Strahl in Objektraumkoordinaten von der Kamera angefordert. Dieser Strahl wird an die Methode `color()` von `t_IllumScene` übergeben, welche das auf den Strahlursprung einfallende Licht berechnet. Dafür wird zunächst durch Aufruf der Methode `intersect()` nach dem nächstgelegenen Objektschnitt gesucht. Existiert kein Schnitt, so wird die Hintergrundfarbe als Ergebnis der Berechnung zurückgeliefert. Andernfalls wird die Methode `shade()` der Oberfläche des geschnittenen Objektes aufgerufen. Diese wiederum fragt die Lichtquellen durch Aufruf der Methode `lightEquPars()` nach dem in diesem Punkt einfallenden Licht und führt gemäß des verwendeten Beleuchtungsmodells eine lokale Beleuchtungsrechnung durch. Abhängig von den Oberflächeneigenschaften wird dann ggf. ein reflektierter und/oder transmittierter Strahl ausgesandt, dessen Intensität jeweils durch rekursiven Aufruf der Methode `color()` von `t_IllumScene` berechnet wird.

- `approximate()` und `radiosity()`

Mittels diesen Methoden wird eine polygonbasierte Darstellung der Szene erzeugt. Die Methode `approximate()` berechnet eine approximative Darstellung unter Anwendung eines lokalen Beleuchtungsmodells, die Methode `radiosity()` eine photorealistische Darstellung unter Anwendung des Radiosity-Verfahrens. Durch Aufruf der Methode `approxShape()` wird jedem geometrischen Objekt signalisiert, eine polygonale Randdarstellung (Boundary Representation) zu berechnen [Ben97]. Die Randdarstellungen werden dann zusammen mit der übrigen Szenenbeschreibung (Kamera, Lichtquellen, Oberflächeneigenschaften) gemäß des gewählten Visualisierungsverfahrens weiterverarbeitet. Die eigentliche Darstellung erfolgt durch die Grafikbibliothek CGI3D, welche als plattformunabhängige Schnittstelle zu OpenGL [Sil93] oder XGL [Sun93] dient.

3.2 Strukturelle Veränderungen und Erweiterungen

Der erste Schritt, die Visualisierung von Volumina in das MRT zu integrieren, bestand darin, einige strukturelle Veränderungen und Erweiterungen am System

durchzuführen. Diese bezogen sich weniger auf die Darstellung von Isoflächen als auf die integrationsbasierte Visualisierung. Einem objektbasierten Ansatz von [Gro96] folgend, wurden dafür Volumina als eine neue Klasse darstellbarer Szenenobjekte eingeführt und die Komponenten des Systems derart angepaßt, daß eine einheitliche Schnittstelle für die gleichzeitige Darstellung von Oberflächen und Volumina gegeben ist. Dadurch wurde insbesondere erreicht, daß eine Vielzahl bereits bestehender Konzepte ohne (bzw. ohne große) Änderungen auch für die Darstellung von Volumina genutzt werden kann. Die für die Anpassung des Systems durchgeführten Änderungen bezogen sich im wesentlichen auf die ray-tracing-basierte Visualisierung und betrafen neben den geometrischen auch die für die Beleuchtungsrechnung relevanten Komponenten.

3.2.1 Szenenobjekte

Wie einleitend erwähnt, wurden für die integrationsbasierte Visualisierung dreidimensionaler Datensätze zunächst Volumina als eine neue Klasse darstellbarer Szenenobjekte eingeführt. Die diesbezüglich durchgeführten Änderungen sind in Abbildung 3.2 wiedergegeben.

Die Basisklasse aller Szenenobjekte ist nach wie vor die Klasse `t_Object`, die Basisklasse aller Materialien und Beleuchtungsmodelle ist `t_Shader`. Abgesehen von einigen namentlichen Änderungen sind die Methoden der alten Klassen (`t_Object` und `t_Surface`) im wesentlichen erhalten geblieben. Lediglich die nur für Oberflächen relevanten Methoden sind in die entsprechenden abgeleiteten Klassen verlegt worden.

Die Klasse `t_SurfaceObject` ist nun die abstrakte Basisklasse für alle primitiven ursprünglichen Szenenobjekte, die Klasse `t_SurfaceShader` ist die Basisklasse für alle oberflächenspezifischen Materialien und Beleuchtungsmodelle. Die Klasse `t_VolumeObject` dient als Basisklasse für volumenspezifische Szenenobjekte, welche im Kontext des MRT gleichermaßen Shader sind. Erwähnenswert ist, daß die Basisklasse `t_Scene` weiterhin von `t_Object` abgeleitet ist, wodurch sich nach wie vor beliebig komplexe Szenenhierarchien mit beliebigen Szenenobjekten aufbauen lassen. Insbesondere ist damit auch weiterhin eine elegante Kombination verschiedener Beschleunigungsstrukturen für die ray-tracing-basierte Visualisierung möglich.

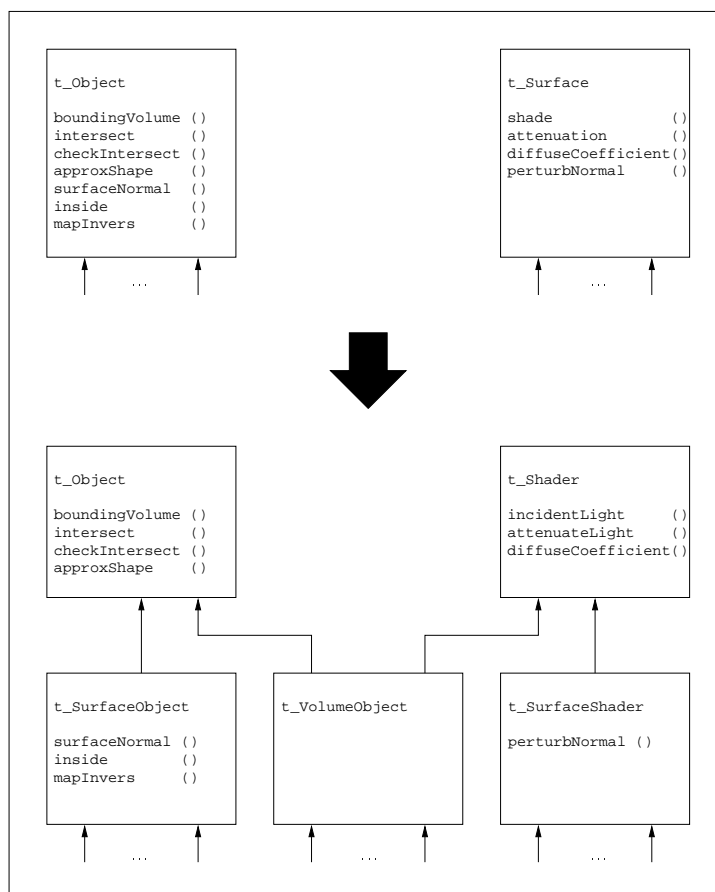


Abbildung 3.2: Änderungen in der Klassenhierarchie

3.2.2 Schnittobjekte

Im Rahmen des Ray-Tracers werden bzgl. der Beleuchtungsrechnung für Oberflächen und Volumina unterschiedliche Schnittinformationen benötigt. Für Oberflächen sind Informationen bzgl. eines *Schnittpunktes*, für Volumina Informationen bzgl. eines *Schnittintervalls* notwendig. Gemäß dieser Unterscheidung wird nun - anstelle eines zuvor übergebenen Distanzwertes und eines Zeigers auf das geschnittene Objekt - von Flächen- und Volumen- Schnittobjekten Gebrauch gemacht, in welche die Szenenobjekte gemäß ihres Typs die entsprechenden Informationen über den Schnitt mit einem Strahl speichern können.

Flächenschnitte

Die Klasse für Flächenschnitte ist `t_SrfIntersec`, deren wesentliche Elemente in nachfolgender Abbildung dargestellt sind. Der Zugriff auf diese Elemente erfolgt über entsprechende Methoden.

```
class t_SrfIntersec

protected:
    t_Real          dist;
    t_3DVector      point;
    const t_SurfaceObject* hitObject;
    int             number;
    t_3DVector      vector;
    t_SrfIntersec* subIntersec;
```

Existiert ein Schnitt mit einem Strahl, so muß die entsprechende Oberfläche den Schnittpunkt (`point`), dessen Distanz zum Strahlursprung (`dist`) und einen Zeiger auf sich selbst (`hitObject`) speichern. Zudem kommt es vor, daß neben den genannten Informationen weitere Ergebnisse aus der Schnittpunktberechnung für eine (spätere) Berechnung des Normalvektors verwendet werden können. Diesbezüglich kann jede Fläche optional eine beliebige Zahl (`number`) und/oder einen beliebigen Vektor (`vector`) im Schnittobjekt speichern. Für die Berechnung des Normalvektors wird der Methode `surfaceNormal()` der geschnittenen Fläche dann das Schnittobjekt mit den zuvor von der Fläche gespeicherten Informationen übergeben. Um diesen Callback-Mechanismus konsequent und sauber auch für zusammengesetzte Szenenobjekte (z.B. CSG-Objekte) nutzbar zu machen, kann an jedes Schnittobjekt auch das Schnittobjekt eines untergeordneten Szenenobjektes (`subIntersec`) angehängt werden. Das Ergebnis einer Schnittberechnung kann somit eine einfach verkettete Liste von Schnittobjekten sein, deren Speicherverwaltung mittels eines Referenzzähler-Mechanismus vollständig von den Schnittobjekten selbst kontrolliert wird.

Volumenschnitte

Die Klasse für Volumenschnitte ist `t_VolIntersec`, auf deren Elemente analog zur Klasse für Flächenschnitte mit entsprechenden Methoden zugegriffen werden kann.

```
class t_VolIntersec

protected:
    t_Real          entryDist;
    t_Real          exitDist;
    const t_VolumeObject* hitObject;
    t_3DVector      rayLocalPoint;
    t_3DVector      rayLocalDir;
```

Das Schnittintervall eines Strahls mit einem Volumen ist definiert durch eine Eintrittsdistanz (`entryDist`) und eine Austrittsdistanz (`exitDist`). Neben diesen Distanzwerten beinhaltet ein Schnittobjekt noch einen Zeiger auf das Volumen (`hitObject`) sowie den Strahlursprung (`rayLocalPoint`) und die Strahlrichtung (`rayLocalDir`) in lokalen Koordinaten. Wie im Abschnitt über Volumina noch erläutert wird, werden die lokalen Koordinaten des Strahls immer mitberechnet und können im Rahmen der Beleuchtungsrechnung weiter verwendet werden.

3.2.3 Strahlobjekte

Um bzgl. der Schnitt- und Beleuchtungsrechnung gleichermaßen eine gemeinsame Schnittstelle für alle Szenenobjekte als auch eine übergeordnete Instanz zur Koordination von Flächen- und Volumenschnitten zu haben, wurde die Verwendung von Strahlobjekten eingeführt. Die Klasse für alle Strahlobjekte ist `t_Ray`, deren wesentliche Elemente und Methoden in Abbildung 3.3 dargestellt sind.

Definiert wird ein Strahl durch einen Strahlursprung (`point`), eine Strahlrichtung (`dir`), ein Szenenobjekt (`fromObject`), von welchem der Strahl ausgesandt wird und eine Suchdistanz (`dist`). Weiterhin besitzt jedes Strahlobjekt

```
class t-Ray
public:
    void          update          (const t_SrfIntersec&);
    void          update          (const t_VolIntersec&);
    const t_Object* intersecObject ();

protected:
    t_3DVector    point;
    t_3DVector    dir;
    const t_Object* fromObject;
    t_Real        dist;
    int           id;
    t_SrfIntersec si;
    t_VolIntersec vi;
```

Abbildung 3.3: Klasse für Strahlobjekte

eine eindeutige Identifikationsnummer (*id*), welche im Rahmen bestimmter Beschleunigungsstrukturen Verwendung findet, um effizient unterschiedliche Strahlen unterscheiden zu können.

Mittels den `update()`-Methoden kann jedes Szenenobjekt sein Schnittergebnis dem Strahl übergeben, welcher dieses dann speichert und u.a. seine Suchdistanz entsprechend anpaßt. Dabei wird insbesondere berücksichtigt, daß eine Fläche innerhalb eines Volumens liegen kann. Konkret sind die Methoden der Klasse wie folgt definiert.

- `void update (const t_SrfIntersec& isec)`
Der Flächenschnitt `isec` wird im Strahl gespeichert und die Suchdistanz `dist` auf die Schnittdistanz `isec.dist` gesetzt. Ist zuvor ein Volumenschnitt gefunden worden, so wird dieser als ungültig markiert, falls dessen Eintrittsdistanz `vi.entryDist` größer als die gefundene Schnittdistanz ist. Ansonsten wird die Austrittsdistanz `vi.exitDist` auf die gefundene Schnittdistanz gesetzt.

- void update (const t_VolIntersec& isec)

Der Volumenschnitt `isec` wird im Strahl gespeichert und die Suchdistanz `dist` auf die Austrittsdistanz `isec.exitDist` gesetzt. Ist zuvor ein Flächenschnitt gefunden worden, so wird dieser als ungültig markiert, wenn dessen Distanz `si.dist` größer als `isec.exitDist` ist.

Der Strahl enthält also immer das Schnittobjekt des am nächsten liegenden Szenenobjektes. Ist dies ein Volumen mit einer darin enthaltenen Fläche, so sind die Schnittobjekte beider Szenenobjekte im Strahl enthalten. Abbildung 3.4 zeigt noch einmal anschaulich für vier verschiedene Strahlen, wie die entsprechenden Distanzwerte gesetzt werden.

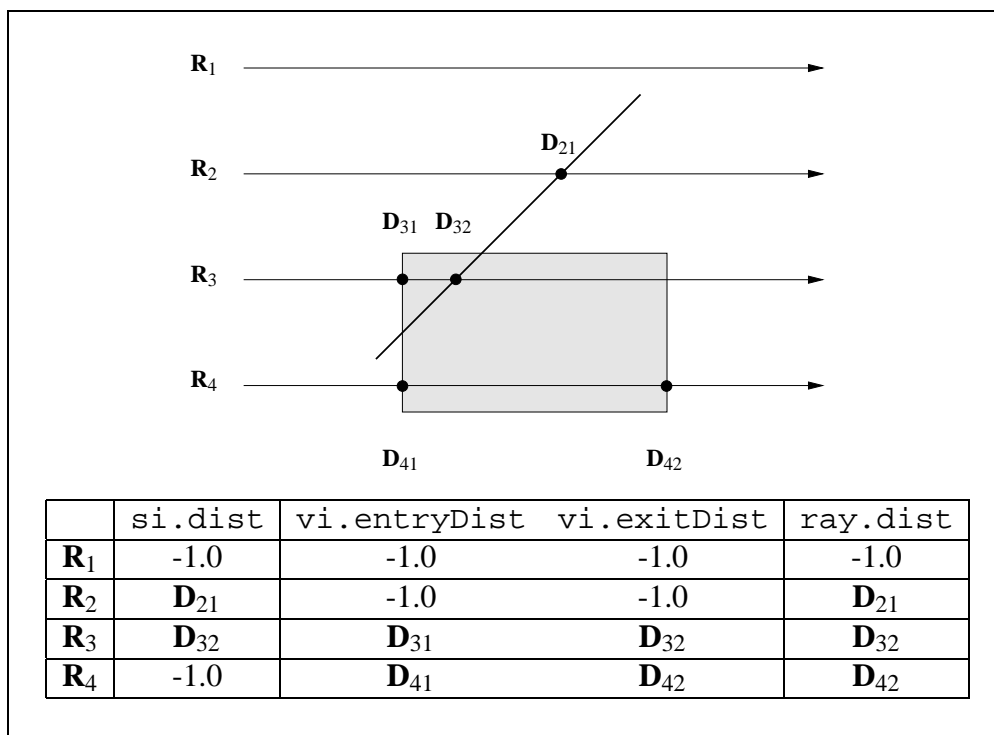


Abbildung 3.4: Schnitte mit Szenenobjekten

Nach erfolgter Schnittpunktberechnung kann der Strahl mittels der Methode `intersectObject()` nach dem am nächsten liegenden Szenenobjekt gefragt werden. Für die Beleuchtungsrechnung ist dann der Strahl mit den Schnittinformationen an den Shader des gefundenen Objektes zu übergeben.

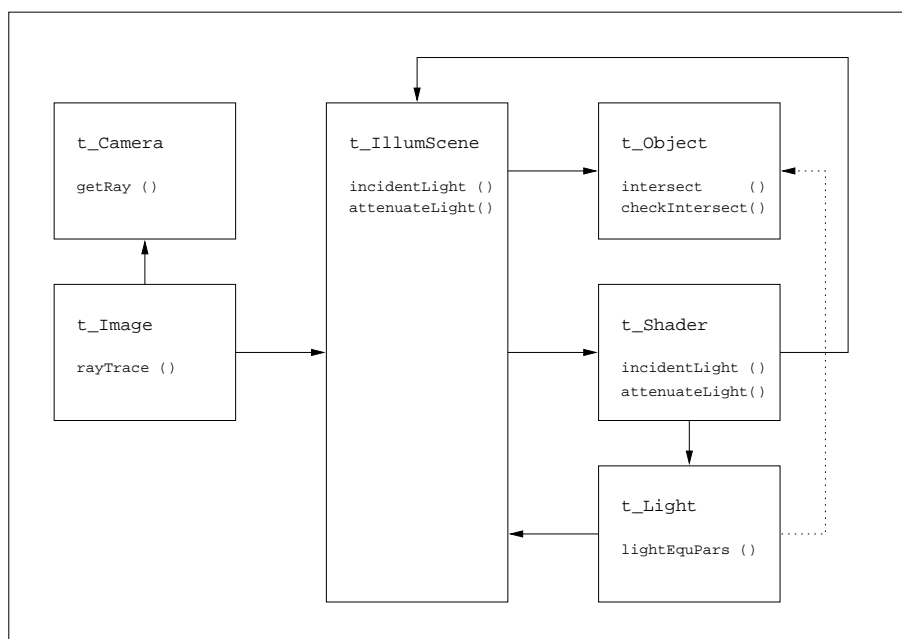


Abbildung 3.5: Die wesentlichen Klassen und deren Kommunikationspfade

3.2.4 Basisklassen der Beleuchtungsrechnung

In diesem Abschnitt soll auf Änderungen und Erweiterungen bzgl. der für die Beleuchtungsrechnung relevanten Basisklassen und derer Methoden eingegangen werden. Neben einer Anpassung an das eingeführte Strahlobjekt und den aus Analogiegründen durchgeführten Namensänderungen waren insbesondere die Klassen derart zu verändern, daß eine adäquate Abschwächung von Schattenstrahlen durch Volumina realisiert werden kann. Abbildung 3.5 stellt die wesentlichen Klassen und deren Kommunikationspfade anschaulich dar.

Die beleuchtete Szene

Die bezüglich der ray-tracing-basierten Visualisierung wesentlichen Methoden der Klasse `t_IllumScene` sind nun analog zur Klasse `t_Shader` die Methoden `incidentLight()` und `attenuateLight()`.

- `virtual`
`t_Color incidentLight (t_Ray& ray, int level) const;`

Diese Methode ist äquivalent zur ursprünglichen Methode `color()` und berechnet bezüglich des Strahls `ray`, das von der gesamten Szene auf den Strahlursprung einfallende Licht. Dafür wird durch Aufruf der Methode `intersect()` des (in der Regel hierarchisch aufgebauten) geometrischen Szenenobjekts zunächst nach dem(den) am nächsten liegenden Schnitt(en) gesucht. Existiert kein Schnitt, so wird die Hintergrundfarbe als Ergebnis zurückgeliefert. Ansonsten wird der Strahl mit den darin enthaltenen Schnittinformationen an die gleichnamige Methode des Shaders des geschnittenen Szenenobjektes zur Beleuchtungsrechnung weitergereicht. Die Variable `level` dient dabei zur Kontrolle der Rekursionstiefe bzgl. der von den Shadern zu generierenden Sekundärstrahlen.

- `virtual`

```
void attenuateLight ( t-Ray& ray,
                    t-Color& intensity ) const;
```

Mittels dieser Methode wird für einen Strahl `ray` eine Abschwächung von dessen Intensität `intensity` in Abhängigkeit der Szenenobjekte berechnet. Analog zur Methode `incidentLight()` wird zunächst nach dem nächstgelegenen Schnitt gesucht und das Schnittergebnis an die gleichnamige Methode des entsprechenden Shaders weitergereicht.

Shader

Analog zur Klasse `t-IllumScene` beinhaltet die Klasse `t-Shader` nun auch die Methoden `incidentLight()` und `attenuateLight()`. Diese sind hier allerdings nicht szenen-, sondern objektspezifisch.

- `virtual`

```
t-Color incidentLight ( const t-Ray& ray,
                      const t-IllumScene* iScene,
                      int level ) const;
```

Die Methode `incidentLight()` entspricht der ursprünglichen Methode `shade()` der Klasse `t-Surface`. Bezüglich der im Strahl enthaltenen Schnittinformationen und den mit `iScene` sowie `level` gegebenen weiteren globalen Parametern kann jeder Shader das auf den Strahlursprung von `ray` einfallende Licht berechnen. Ein Shader führt dafür eine lokale

Beleuchtungsrechnung durch und generiert ggf. Sekundärstrahlen, deren Intensität jeweils durch rekursiven Aufruf der Methode `incidentLight()` von `iScene` berechnet wird.

- `virtual`

```
void attenuateLight ( const t-Ray&          ray,
                    t-Color&             intensity,
                    const t-IllumScene* iScene,
                    t-Real                 attDist ) const;
```

Diese Methode ersetzt die ursprüngliche Methode `attenuation()`. Jeder Shader kann für einen Strahl `ray` mit den darin enthaltenen Schnittinformationen eine Abschwächung von dessen Intensität `intensity` berechnen. Im Gegensatz zur Methode `attenuation()` wird nun eine Abschwächung in Abhängigkeit eines konkreten Schnittes und nicht nur in Abhängigkeit eines Distanzwertes berechnet. Einerseits können damit nun Schatten von Volumina, andererseits auch Schatten von transparenten texturierten Flächen adäquat dargestellt werden. Die Shader sind selbst dafür verantwortlich, ausgehend vom aktuellen Schnitt, fortgesetzte Strahlen zu generieren, deren weitere Abschwächung durch rekursiven Aufruf der Methode `attenuateLight()` von `iScene` berechnet wird. Die Variable `attDist` beinhaltet dafür die Distanz vom Strahlursprung zur Lichtquelle, entlang welcher eine Abschwächung zu berücksichtigen ist.

Lichtquellen

Gemäß den bzgl. der Schattenberechnung durchgeführten Änderungen an oben genannten Klassen, war es auch nötig die Lichtquellen entsprechend zu verändern. Dies betraf weniger die Basisklasse als die davon abgeleiteten Klassen vom Typ `t-AttenuatedLight`, welche eine Abschwächung der Strahlen berücksichtigen. Die dafür relevante, in der Basisklasse `t-Light` deklarierte Methode, ist `lightEquipars`, welche der Vollständigkeit halber hier aufgeführt ist.

```
virtual
t_Bool lightEquPars (  const t_IllumScene*  iScene,
                      const t_3DVector&    point,
                      const t_3DVector&    normal,
                      const t_Object*      obj,
                      int                  level,
                      t_3DVector&         ldir,
                      t_Color&            intensity ) const;
```

Durch Aufruf dieser Methode kann ein Shader das an einem Punkt `point` auf ein Objekt `obj` einfallende Licht `intensity` ermitteln. Der Vektor `normal` definiert den Halbraum, aus welchem der Punkt beleuchtet werden kann. Liegt der Punkt bzgl. der Lichtquelle im Schatten, so wird als Ergebnis der Wert `False` zurückgeliefert, ansonsten der Wert `True` und neben der Intensität der Vektor `ldir` zur Lichtquelle.

Die Lichtquellen, welche keine Abschwächung des Lichtes berücksichtigen, testen nach wie vor mittels Aufruf der Methode `checkIntersect()` des geometrischen Szenenobjektes, ob sich ein blockierendes Objekt zwischen Punkt und Lichtquelle befindet.

Die Lichtquellen, welche eine Abschwächung berücksichtigen (abgeleitet von `t_AtenuatedLight`) ermitteln das auf einen Punkt einfallende Licht, indem sie jetzt für jeden generierten Strahl zwischen Punkt und Lichtquelle die Methode `attenuateLight()` von `iScene` aufrufen. Lichtquellen dieses Typs dienen im wesentlichen also nur noch der Generierung von Strahlen. Für deren Abschwächung ist die Szene `iScene` sowie der Shader eines jeden Szenenobjektes zuständig.

3.3 Volumenspezifische Klassen

Nach der Vorbereitung bzw. Anpassung des Systems zur Visualisierung von Volumina, soll in diesem Abschnitt nun auf die konkrete Integration der in Kapitel 2 genannten Konzepte und Verfahren eingegangen werden. Wert wurde auch darauf gelegt, eine Basis für die Realisierung weiterer Repräsentations- und Visualisierungstechniken zu etablieren.

3.3.1 Volumendaten und Traversierer

Volumendaten sind im Kontext des MRT dreidimensionale, durch einen achsenparallelen Quader begrenzte Skalarfelder, die in einer beliebigen Form repräsentiert sein können. Die Klassenhierarchie für Volumendaten ist in Abbildung 3.6 wiedergegeben. Unterschieden wird zwischen Daten, die in kontinuierlicher Form und Daten, die in diskreter Form gegeben sind. Momentan implementiert sind eine Rauschfunktion, eine Turbulenzfunktion, eine Funktion, mit welcher aufsteigender Rauch simuliert wird, sowie ein generisches reguläres Gitter, welches mit einem beliebigen anderen Volumendaten-Objekt oder mit einem auf einem Festspeicher befindlichen Datensatz initialisiert werden kann.

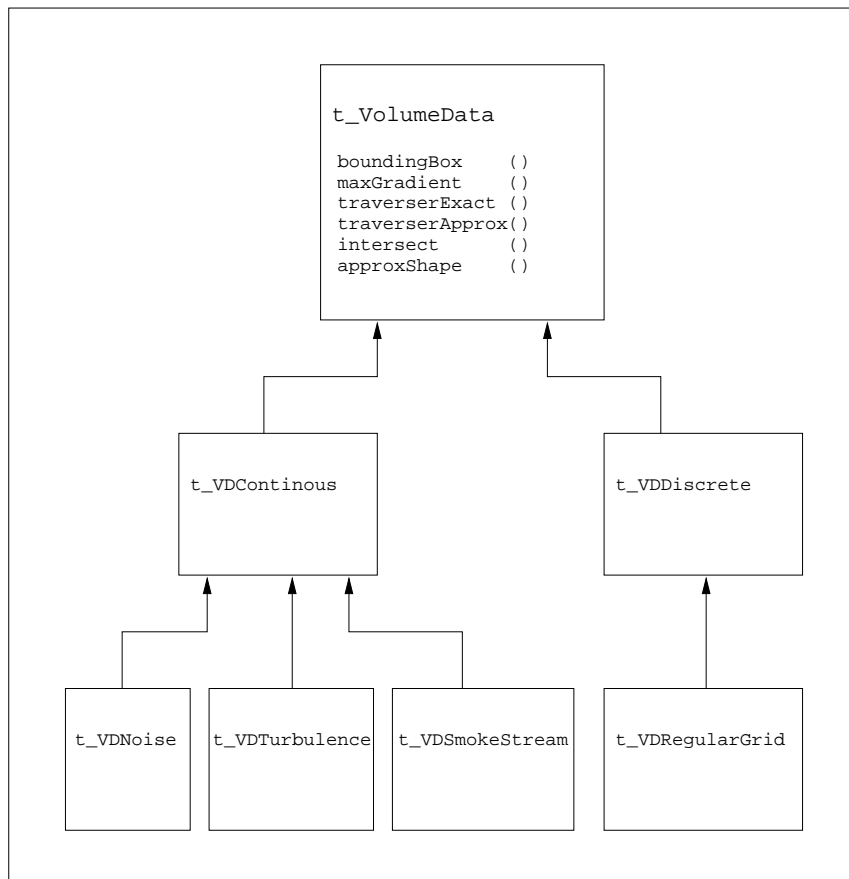


Abbildung 3.6: Klassenhierarchie für Volumendaten

Volumendaten

Momentan beinhaltet die Basisklasse `t_VolumeData` im wesentlichen vier Methoden.

- `const t_AA_Box& boundingBox () const;`

Von jedem Objekt kann dessen Definitionsbereich in Form eines achsenparallelen Quaders abgefragt werden.

- `t_3DVector maxGradient () const;`

Zu Normierungszwecken kann zu jedem Datenfeld der maximale Gradient, d.h. genauer, der Gradientenvektor maximaler Länge abgefragt werden.

- `virtual t_VDTraverserPtr traverserApprox () const;`
`virtual t_VDTraverserPtr traverserExact () const;`

Von jedem Objekt können zwei Typen von Traversierern angefordert werden, mit denen die Daten in regelmäßigen Abständen abgetastet werden können. Für die Berechnung eines Datenwertes ist es in der Regel nötig, diesen aus diskreten Gitterwerten zu rekonstruieren. Unterschieden wird diesbezüglich zwischen Traversierern, die eine approximative Rekonstruktion benutzen und Traversierern, die eine möglichst exakte Rekonstruktion benutzen.

Erwähnenswert ist weiterhin, daß anstelle eines gewöhnlichen Zeigers ein Referenz-Zeiger [Fis95] auf einen Traversierer geliefert wird, der sich selbständig um die Destruktion des dynamisch erzeugten Traversierers kümmert.

Für die Darstellung von Isoflächen ließen sich noch zwei weitere Methoden hinzufügen.

- `virtual`
`t_Bool intersect (const t_3DVector& point,`
`const t_3DVector& dir,`
`t_Real threshold,`
`t_Real& dist) const;`


```

class t_IsoSurface : public t_SurfaceObject

public:
    t_BVol      boundingVolume () const;
    t_Bool      intersect      (t_Ray& ray);
    t_Bool      checkIntersect (t_Ray& ray);
    t_3DVector  surfaceNormal  (const t_SrfIntersec& si) const;

protected:
    t_VolumeDataPtr  volumeData;
    t_Real           threshold;
    t_Real           sampleStepSize;
    t_4x3Matrix      matrix;
    t_SurfaceShader* shader;

```

Abbildung 3.7: Die Klasse für Isoflächen

Liefert den Datenwert an der aktuellen Position. Der Datenwert ist normiert und liegt immer im Bereich zwischen Null und Eins.

- virtual
t_3DVector currGradient () const;

Liefert den Gradienten an der aktuellen Position.

Aus Effizienzgründen wird (in den abgeleiteten Klassen) nicht getestet, ob sich die aktuelle Abtastposition innerhalb des Definitionsbereiches der Volumendaten befindet, d.h. der Anwender ist selbst für eine korrekte Traversierung verantwortlich.

3.3.2 Isoflächen

Eine Isofläche kann im Kontext des MRT als ein gewöhnliches Oberflächenobjekt betrachtet werden. Insofern lag es nahe, für Isoflächen eine von der Klasse `t_SurfaceObject` abgeleitete Klasse `t_IsoSurface` einzuführen, womit alle schon vorhandenen Konzepte zur Darstellung von Flächen auch zur Darstellung von Isoflächen benutzt werden können.

Abbildung 3.7 zeigt die Klasse für Isoflächen. Definiert wird ein Objekt vom Typ `t_IsoSurface` durch einen (Referenz-)Zeiger auf einen Volumendatensatz, einen Schwellwert, eine für die Schnittpunktberechnung benötigte Abtastschrittweite, eine Matrix zur Transformation der Volumendaten in Weltkoordinaten und einen Zeiger auf einen (beliebigen) Flächenshader. Jede in einem Skalarfeld zu betrachtende Isofläche kann somit individuell dargestellt und mittels CSG-Operationen (diese sind im Rahmen des MRT auch mit nicht geschlossenen Flächen möglich) manipuliert werden.

Die Schnittpunktberechnung erfolgt gemäß des ersten in Abschnitt 2.2.1 genannten Verfahrens, indem mittels des vom Volumendaten-Objekt angeforderten Traversierers die Daten in regelmäßigen Abständen solange abgetastet werden, bis zwei aufeinanderfolgende Abtastwerte gefunden wurden, zwischen denen der gegebene Schwellwert liegt. Der Schnittpunkt bzw. dessen Distanz zum Strahlursprung wird dann durch lineare Interpolation bestimmt. Die polygonale Approximation ist zur Zeit nicht implementiert und sollte Aufgabe des Volumendaten-Objektes sein, d.h. es sollte, wie in Abschnitt 3.3.1 beschrieben, eine Methode `approxShape()` beinhalten, mit welcher eine polygonale Approximation berechnet werden kann. Das Polygonnetz ist dann nur noch gemäß der gegebenen Transformationsmatrix in Weltkoordinaten zu transformieren. Die Realisierung der übrigen Methoden dürfte offensichtlich sein.

3.3.3 Volumenobjekte

Volumenobjekte entsprechen im Kontext des MRT wie schon erwähnt dreidimensionalen Datensätzen, die mittels integrierender Verfahren dargestellt werden. Die (abstrakte) Basisklasse für alle Volumenobjekte, gleichermaßen abgeleitet von `t_Object` als auch von `t_Shader`, ist `t_VolumeObject`. Abbildung 3.8 zeigt die wesentlichen Elemente und Methoden, sowie die momentan abgeleiteten bzw. realisierten Klassen. Die Klasse `t_BlinnVolume` realisiert das Modell von Blinn/Kajiya & von Herzen zur Darstellung atmosphärischer Phänomene, die Klasse `t_LevoyVolume` das Modell von Levoy zur Darstellung von Konturflächen. Beide Klassen unterscheiden sich lediglich in den für die Beleuchtungsrechnung relevanten Methoden. Die Berechnung der übrigen Methoden erfolgt in der Basisklasse.

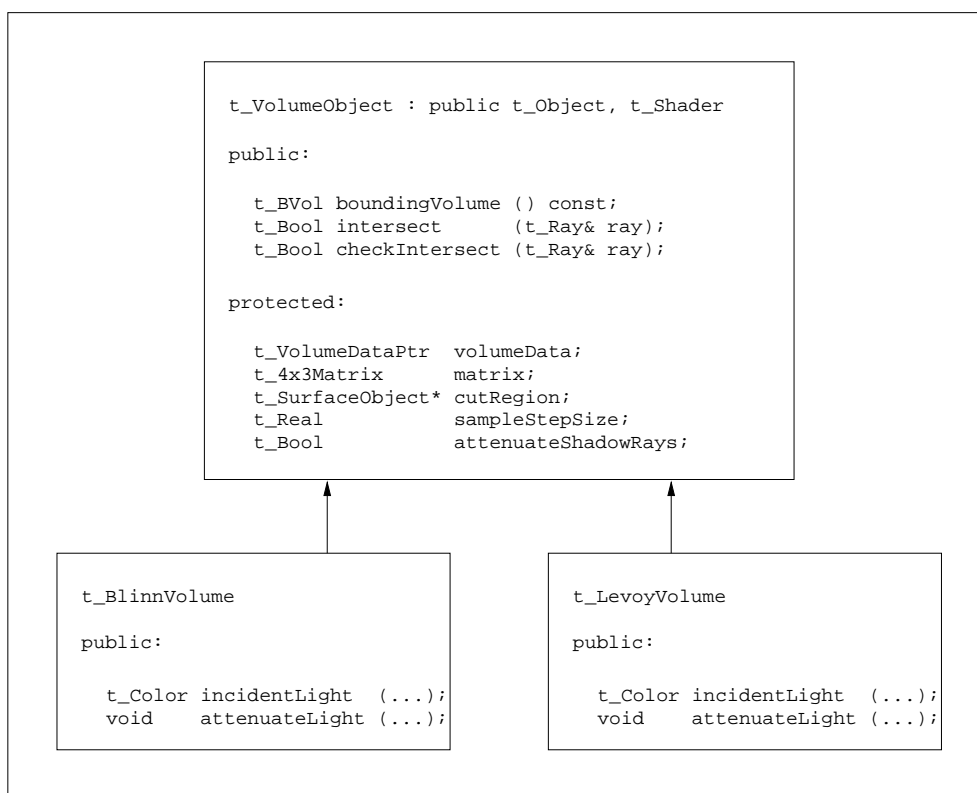


Abbildung 3.8: Klassenhierarchie der Volumenobjekte

Definiert wird ein Volumenobjekt generell durch einen Volumendatensatz, eine Matrix zur Transformation der Volumendaten in Weltkoordinaten, eine optionale Schnittfläche, welche typischerweise ein Festkörper ist, eine Abtastschrittweite, sowie durch diverse vom verwendeten Shadingverfahren abhängige Materialparameter. Weiterhin kann festgelegt werden, welche Art von Traversierer (approximativ oder exakt) zu benutzen ist und ob Schattenstrahlen abgeschwächt werden sollen.

Der Hüllkörper der Volumendaten und die Transformationsmatrix definieren zusammen die geometrische Begrenzungsfläche bzw. die Volumenregion der Daten in Weltkoordinaten, aus welcher mittels der Schnittfläche ein beliebiger Bereich herausgeschnitten werden kann. Für die im Rahmen des Ray-Tracers zu berechnenden Schnittintervalle wurden die entsprechenden Klassen (`t_AABOx` und `t_SurfaceObject`) jeweils um eine entsprechende Schnitt-Methode erweitert,

welche für Oberflächenobjekte in der Default-Implementierung ein Schnittintervall unter Anwendung der ursprünglichen Schnittpunkt-Berechnungsmethode ermittelt.

Visualisierung mittels Ray-Tracing

Die Visualisierung mittels Ray-Tracing läuft allgemein wie folgt ab: Zur Schnittberechnung wird zunächst die Methode `intersect()` aufgerufen. Diese fragt den Hüllkörper der Volumendaten und ggf. die Schnittfläche nach dem jeweiligen Schnittintervall. Existiert ein Schnitt mit dem Hüllkörper so wird dessen Schnittintervall bzw. die Differenz zwischen diesem und des Intervalls der Schnittfläche mit den in Abschnitt 3.2.2 genannten Informationen in einem Volumenschnitt gespeichert und an den Strahl übergeben. Für die Beleuchtungsrechnung wird dann der Strahl an die Methode `incidentLight()` weitergereicht. Mittels des entsprechenden Traversierers werden die Volumendaten entlang des Schnittintervalls in regelmäßigen Abständen abgetastet und eine Beleuchtungsrechnung an jedem Punkt (in Weltkoordinaten) durchgeführt, bis die aufintegrierte Opazität einen vorgegebenen Schwellwert unterschritten hat oder bis der Endpunkt des Schnittintervalls erreicht ist. Ist der Endpunkt des Schnittintervalls erreicht und die Opazität größer als der gegebene Schwellwert, so ist noch das auf den Endpunkt von "außerhalb" einfallende Licht zu berücksichtigen und entsprechend abzuschwächen. Diesbezüglich werden zwei Fälle unterschieden: Beinhaltet der übergebene Strahl einen Flächenschnitt, so wird die Methode `incidentLight()` des Flächen-shaders aufgerufen. Beinhaltet der Strahl keinen weiteren Schnitt, so wird vom Endpunkt ein fortgesetzter Strahl gleicher Richtung generiert und die Methode `incidentLight()` der beleuchteten Szene (`t_IllumScene`) aufgerufen. In beiden Fällen wird die berechnete Lichtintensität gemäß des verwendeten Shadingmodells dann abgeschwächt und zu der zuvor ermittelten Intensität addiert. Dieses für die Berechnung des einfallenden Lichts beschriebene Prinzip gilt analog für die Abschwächung von Schattenstrahlen.

Visualisierung im Polygon-Renderer

Die approximative Visualisierung im Rahmen des Polygon-Renderers wird momentan noch nicht unterstützt, stellt aber prinzipiell kein Problem dar. Ein mögli-

cher Ansatz bestünde darin, gemäß der in Abschnitt 2.3.4 erläuterten texturbasierten Darstellung, das Volumen durch eine Folge von transparenten, texturierten Flächen zu repräsentieren, welche dann einzeln vom Polygon-Renderer zu visualisieren sind. Konkret könnte dies wie folgt realisiert werden.

Bei Aufruf der Methode `approxShape()` wird dem Volumenobjekt signalisiert, die besagte Folge von transparenten, texturierten Flächen zu generieren. Jede Fläche wird dabei durch ein Oberflächenobjekt vom Typ `t_Parallel` (ein Parallelogramm) sowie durch einen ihm zugeordneten Oberflächenshader repräsentiert, welcher die entsprechenden Texturdaten beinhaltet. Für die zu benutzenden Shader sollte eine neue Klasse (z.B. `t_2DVOLTexture`) implementiert werden, welche von der für Oberflächen realisierten Basisklasse `t_2DTexture` abgeleitet ist. Shader dieses Typs könnten dann zum Beispiel die benötigten Texturdaten erst bei Bedarf berechnen, womit eine Speicherung der gesamten Textur entfällt.

Die einzelnen erzeugten Oberflächenobjekte können zwar mittels bestehender Techniken im Rahmen des Polygon-Renderers dargestellt werden, allerdings fehlt noch eine elegante Schnittstelle für den Zugriff auf die Objekte. Etabliert werden kann diese, indem die Klasse `t_VolumeObject` nicht mehr von `t_Object`, sondern von `t_Scene` abgeleitet wird. Ein Volumenobjekt wird damit aus Sicht des Polygon-Renderers als eine aus gewöhnlichen Oberflächenobjekten bestehende Szene verstanden, auf deren Elemente mittels der bestehenden Methoden zugegriffen werden kann.

Insgesamt ist durch den beschriebenen Ansatz gewährleistet, daß *alle* bestehenden Konzepte des Polygon-Renderers *ohne* Änderungen oder Erweiterungen auch für die Visualisierung von Volumina genutzt werden können.

3.4 Zusammenfassung

Trotz der genannten Änderungen ist die Grundstruktur des Systems erhalten geblieben. Es wurden lediglich die wesentlichen (Basis)-Klassen an den entscheidenden Stellen abstrahiert. Insbesondere wurde damit erreicht, daß ohne große Änderungen eine Vielzahl bereits bestehender Konzepte für die gleichzeitige Visualisierung von Oberflächen und Volumina nutzbar ist. Die konkret realisierten Klassen können dabei als Beispiele verstanden werden. Weitere Repräsentations- und Visualisierungstechniken lassen sich leicht durch Ableiten neuer Klassen im-

plementieren.

Abschließend sind hier noch einmal die wesentlichen Eigenschaften des realisierten Entwurfs zusammenfassend dargestellt.

- Es werden nun generell zwei Typen von Szenenobjekten unterschieden: Oberflächen- und Volumenobjekte. Oberflächenobjekte entsprechen den ursprünglichen Szenenobjekten, Volumenobjekte entsprechen dreidimensionalen Datensätzen, die mittels integrierenden Verfahren visualisiert werden.
- Oberflächen- und Volumenobjekte sind gleichwertige Szenenobjekte. Sie können in beliebiger Anzahl in einer Szene enthalten sein und bieten eine einheitliche und transparente Schnittstelle zu den übrigen Komponenten des Systems.
- Oberflächen können sich mit Volumina überlappen. Sich überlappende Volumina hingegen sind nicht möglich, da es nicht klar, bzw. nicht definiert ist, wie verschiedene Materialien miteinander zu verknüpfen sind. Aus streng physikalischer Sicht ist es auch nicht sinnvoll, einem Teilraum mehrere Materialien mit unterschiedlichen optischen Eigenschaften zuzuordnen.
- Alle vorhandenen Typen von Lichtquellen lassen sich auch für die Visualisierung von Volumina einsetzen. Insbesondere wird berücksichtigt, daß Volumina Schatten auf andere Szenenobjekte wie auch auf sich selbst werfen können. Unterstützt wird dies allerdings nur bei Lichtquellen vom Typ `t_AtenuatedLight`. Bei herkömmlichen Lichtquellen, die nur auf die Existenz eines Schnittes testen, werden Schatten von Volumina ignoriert.
- Bezüglich der ray-tracing-basierten Visualisierung können alle bereits zuvor implementierten Beschleunigungsstrukturen (abgeleitet von `t_Scene`) sowohl für Oberflächen als auch für Volumina genutzt werden.
- Für die darzustellenden Volumendaten existiert eine eigene Klassenhierarchie, welche beliebige Repräsentationen bzw. Gitterstrukturen erlaubt. Für den iterativen Zugriff auf die Daten können von den Objekten Traversierer angefordert werden, die auf die jeweilige Datenstruktur optimiert sein können.

- Eine Instanz eines Volumendatensatzes kann von verschiedenen Szenenobjekten (Objekte vom Typ `t_IsoSurface` als auch Objekte vom Typ `t_VolumeObject`) referenziert und jeweils beliebig affin transformiert werden. Auch lassen sich (optional) beliebige Bereiche mittels CSG-Operationen wegschneiden. Ein Datensatz kann somit auf unterschiedlichste Art und Weise (auch mehrfach) in einer Szene dargestellt werden.

Kapitel 4

Ergebnisse

Die folgenden fünf Abbildungen zeigen dreidimensionale Szenarien, die mittels der Ray-Tracing-Komponente des MRT visualisiert wurden. Als Systemplattform diente ein handelsüblicher IBM PC kompatibler Rechner mit 200MHz Pentium CPU und 32MB Hauptspeicher. Darauf installiert war das Betriebssystem Linux. Die Berechnungszeiten der Bilder beziehen sich auf eine Auflösung von 300×300 Bildpunkte und sind in Tabelle 4.1 zusammengefaßt. Die den Bildern zugrundeliegenden Szenenbeschreibungen sind im Anhang A aufgeführt.

Abbildung	Zeit in Minuten
4.1	1:16
4.2	3:06
4.3	4:53
4.4	4:46
4.5	20:06
4.6	1:20

Tabelle 4.1: Berechnungszeiten

Abbildung 4.1 zeigt die Isofläche eines menschlichen Kopfes mit freigelegtem Gehirn. Der Datensatz wurde durch eine MRI-Messung gewonnen und hat eine Auflösung von $256 \times 256 \times 109$ Gitterpunkten. In der Szene befinden sich neben der Isofläche eine spiegelnde ebene Fläche sowie zwei Punktlichtquellen. Eine Lichtquelle ist an der Betrachterposition definiert, die andere befindet sich hinter

der Isofläche.

In Abbildung 4.2 wurde der mit einer Auflösung von $256 \times 256 \times 109$ Gitterpunkten gegebene CT-Datensatz einer Maschine mittels des Verfahrens von Levoy visualisiert. Die Ventile und die Rückwand sind hier mit hoher Opazität und rötlicher Farbe, der Maschinenblock mit geringer Opazität und bläulicher Farbe dargestellt. In der Szene befinden sich desweiteren eine Punktlichtquelle an der Betrachterposition sowie eine spiegelnde Fläche unterhalb des Volumens, welche einen Einblick in das Innere der Maschine gewährt.

In den Abbildungen 4.3 und 4.4 wurde der CT-Datensatz eines menschlichen Kopfes mittels des Verfahrens von Levoy visualisiert. Der Haut ist eine geringe Opazität und eine bläuliche Farbe zugeordnet, den Knochen eine hohe Opazität und eine gelbliche Farbe. Die Auflösung des Datensatzes beträgt wieder $256 \times 256 \times 109$ Gitterpunkte. Abbildung 4.3 zeigt den kompletten Kopf, der sich in der darunterliegenden Fläche spiegelt und von einer an der Betrachterposition befindlichen Punktlichtquelle beleuchtet wird. Abbildung 4.4 zeigt die hintere Hälfte des Kopfes, die einen Schatten auf die darunterliegende Fläche wirft. Die vordere Hälfte wurde mittels eines Quaders als Differenzobjekt weggeschnitten. Der Schatten resultiert aus einer sich oberhalb des Volumens befindlichen Punktlichtquelle.

Abbildung 4.5 zeigt eine Szene, die von einem Rauchfeld umschlossen ist. Zur Modellierung des Rauches diente eine Rauschfunktion, die Visualisierung erfolgte nach dem Verfahren von Blinn/Kajiya & von Herzen. Neben einer Kugel und einer darunterliegenden ebenen Fläche befinden sich auch der Betrachter, eine Punktlichtquelle und ein Spotlight innerhalb des Volumens. Die Punktlichtquelle hat eine blaue Farbe und befindet sich links neben der Kugel, das weiße Spotlight befindet sich überhalb der Kugel und wirft einen Lichtkegel auf diese. Die im Vergleich zu den anderen Bildern relativ hohe Berechnungszeit läßt sich u.a. dadurch erklären, daß die komplette Szene von einem Volumen umhüllt ist und somit entlang eines jeden Strahls über die ganze Länge eine Integration durchgeführt werden muß. Desweiteren ist auch die Abtastschrittweite sehr klein gewählt, um Schatten- und Lichtkegel entsprechend akkurat darzustellen.

In Abbildung 4.6 wurde aufsteigender Rauch simuliert. Definiert wurde dieser gemäß einer in [Ebe94] gegebenen Prozedur, visualisiert wurde er nach dem Verfahren von Blinn/Kajiya & von Herzen. Die Beleuchtung und der Schatten re-

sultieren aus einer rechts oberhalb des Betrachters positionierten Punktlichtquelle. Die relativ geringe Berechnungszeit ist u.a. dadurch zu erklären, daß einerseits eine hohe Abtastschrittweite gewählt wurde und andererseits der prozedural definierte (kontinuierliche) Datensatz für eine effizientere Berechnung der Dichtewerte vor der Bildgenerierung in ein regulär strukturiertes Gitters überführt wurde. Alternativ kann der Datensatz natürlich auch direkt visualisiert werden, was einen geringeren Speicherplatzbedarf aber eben auch eine höhere Berechnungszeit zur Folge hat.

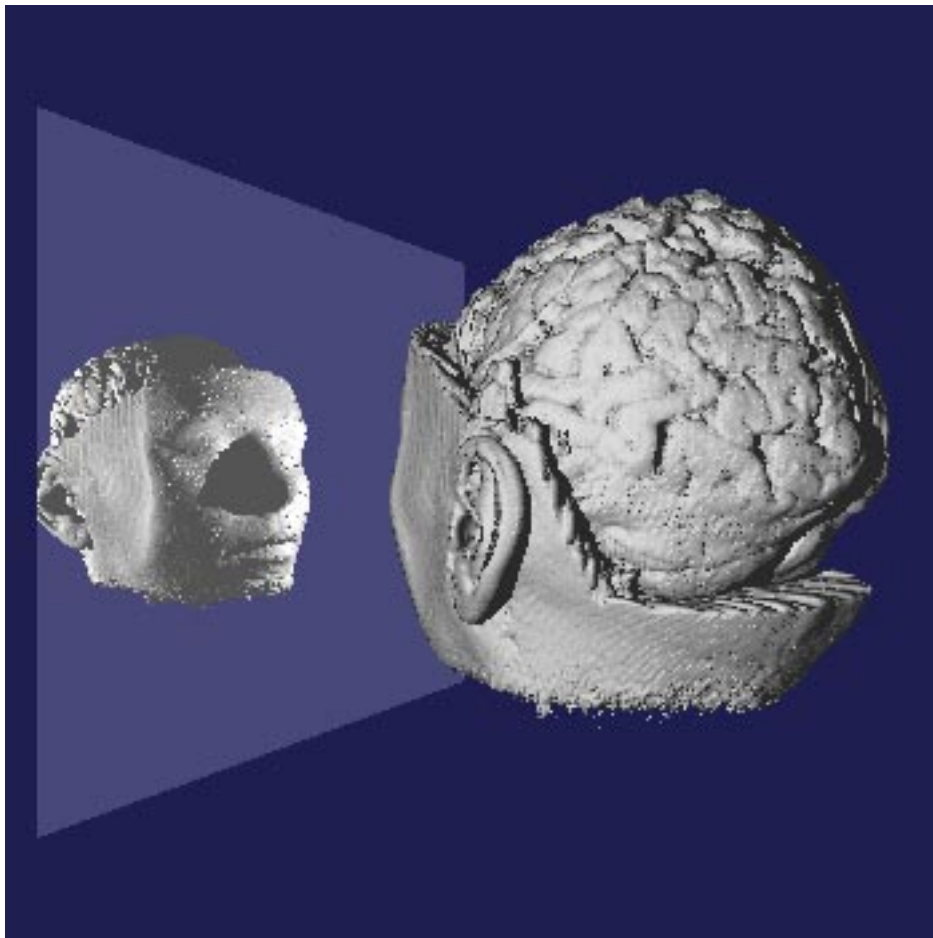


Abbildung 4.1: Visualisierung einer Isofläche aus dem MRI-Datensatz eines menschlichen Kopfes.

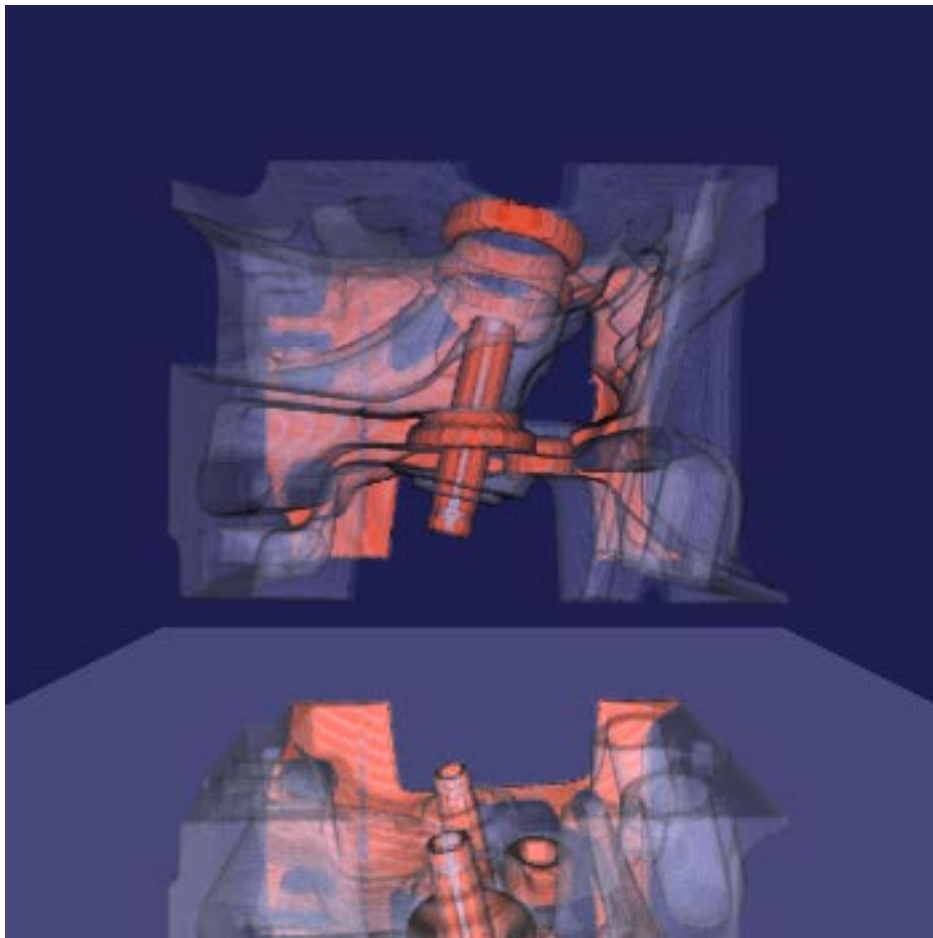


Abbildung 4.2: CT-Datensatz einer Maschine, visualisiert nach dem Verfahren von Levoy.

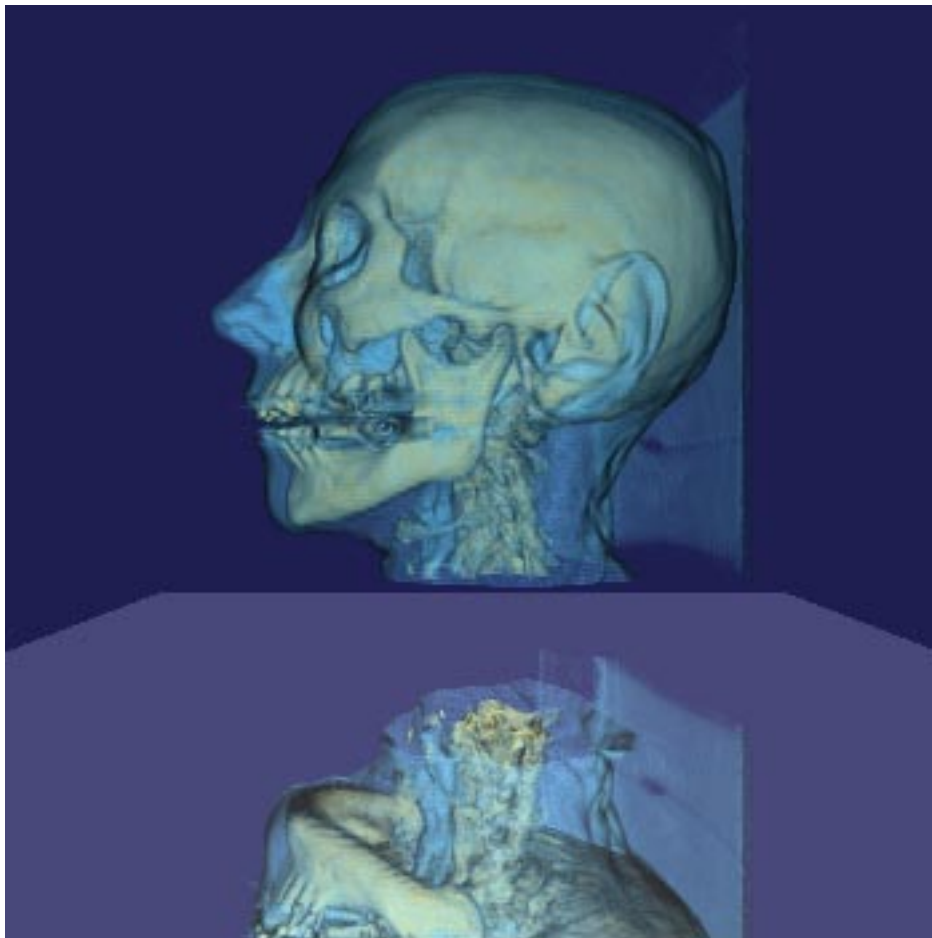


Abbildung 4.3: CT-Datensatz eines menschlichen Kopfes, visualisiert nach dem Verfahren von Levoy.

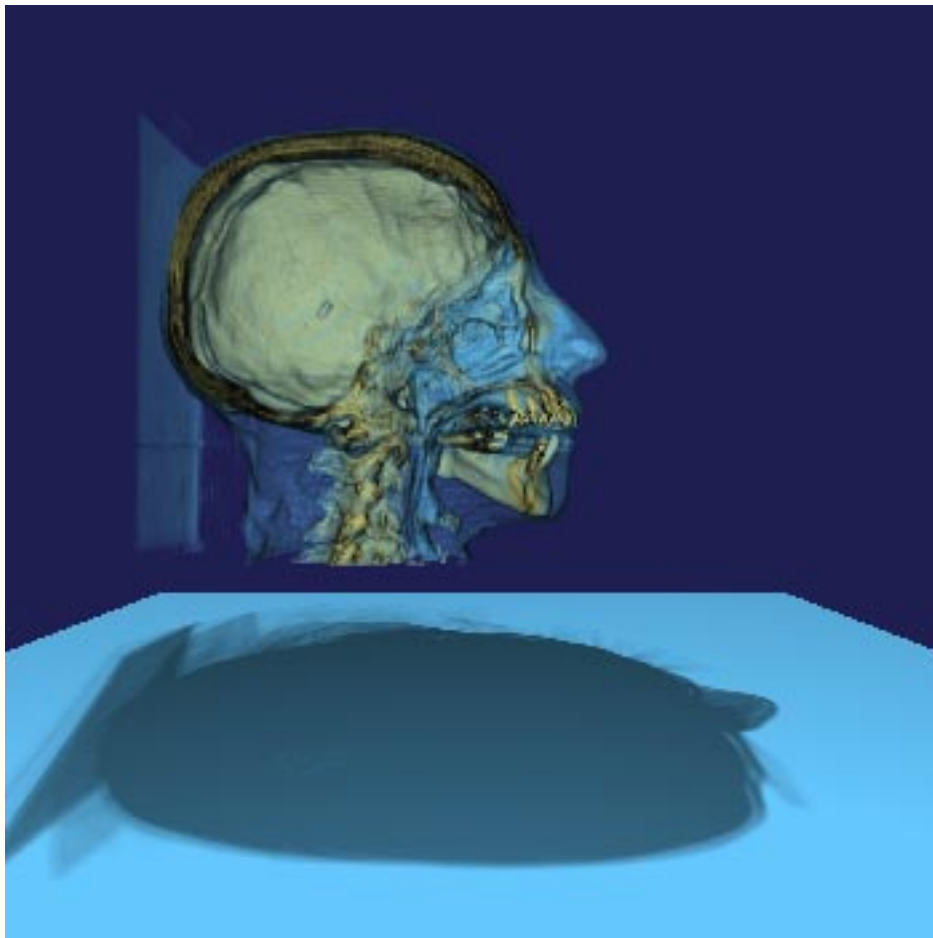


Abbildung 4.4: CT-Datensatz eines menschlichen Kopfes, visualisiert nach dem Verfahren von Levoy.

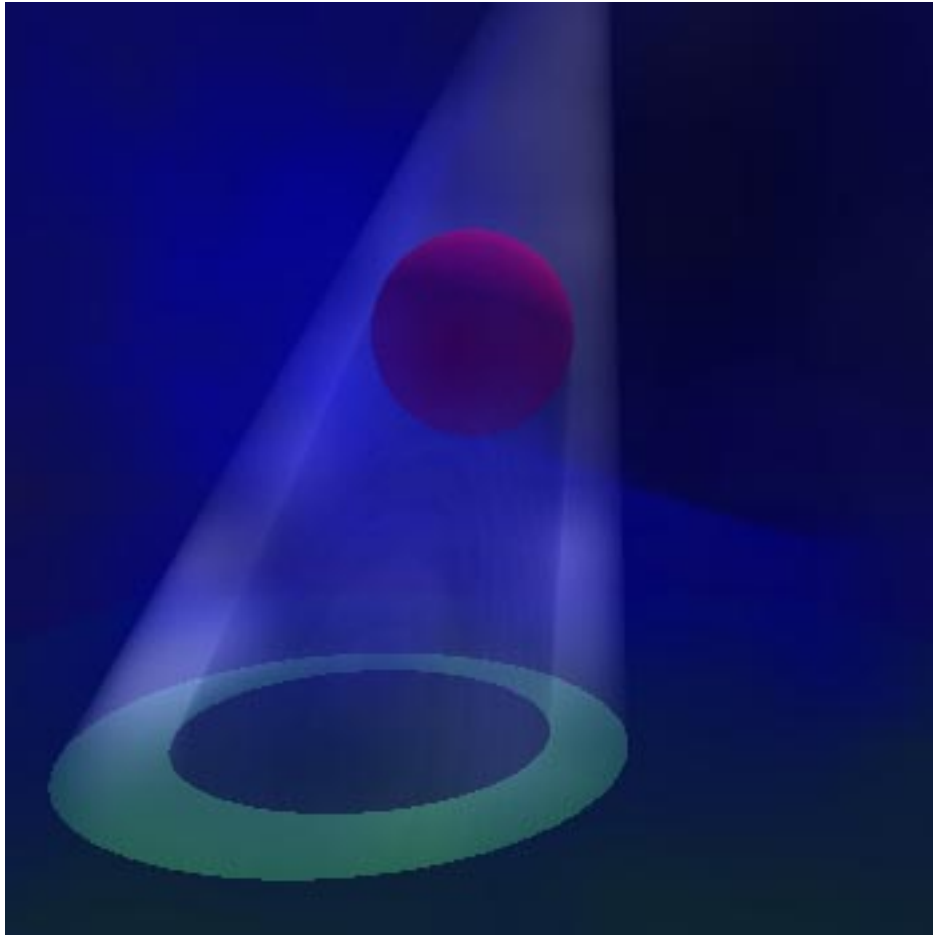


Abbildung 4.5: Eine Szene, umschlossen von einem mit Rauch gefüllten Volumen. Das Volumen wurde nach dem Modell von Blinn/Kajiya & von Herzen visualisiert.

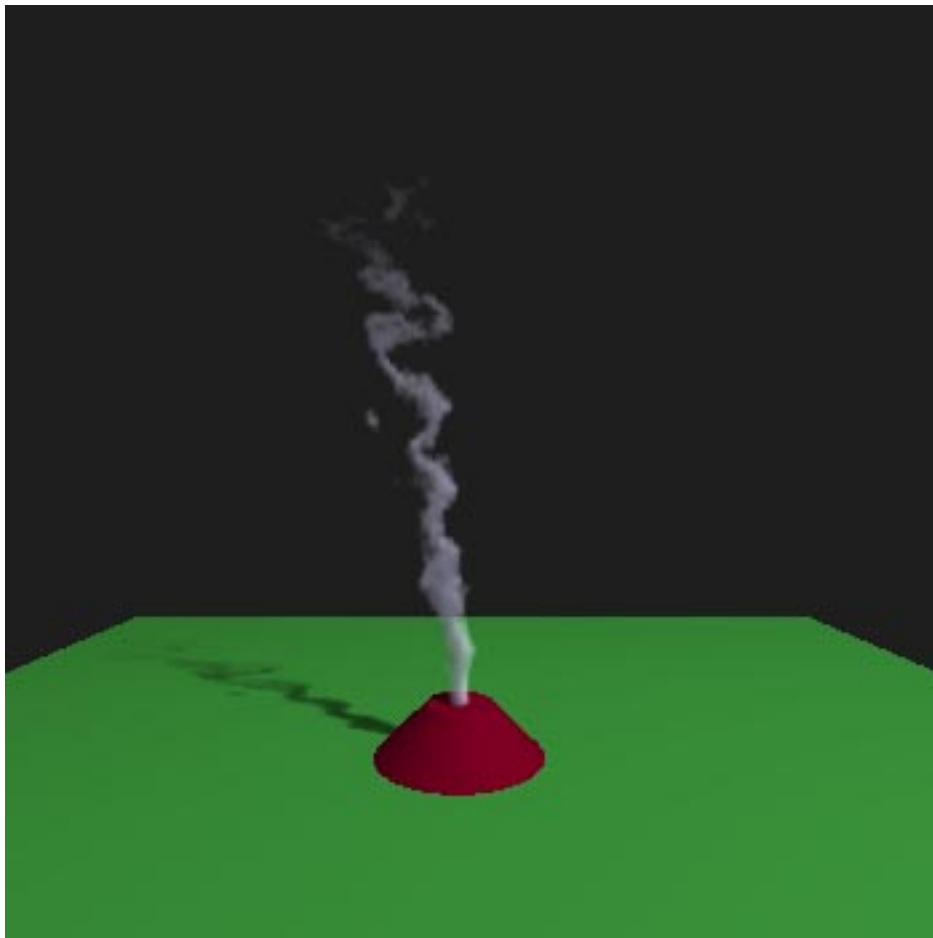


Abbildung 4.6: Aufsteigender Rauch. Prozedural definiert und visualisiert nach dem Modell von Blinn/Kajiya & von Herzen.

Kapitel 5

Ausblick

Mit dem im Rahmen dieser Arbeit vorgestellten und implementierten Entwurf wird u.a. eine Basis zur Realisierung weiterer Methoden zur Definition, Repräsentation und Visualisierung von Volumina etabliert. Die entscheidenden Schnittstellen zur Erweiterung der Funktionalität des MRT sind dabei durch die abstrakten Basisklassen `t_VolumeData` und `t_VolumeObject` gegeben, welche unter Ausnutzung des Ableitungsmechanismus eine einfache Integration neuer Typen von Volumendaten bzw. neuer Typen von Volumenobjekten erlauben.

Mit weiteren Typen von Volumendaten lassen sich zum Beispiel neue Gitterstrukturen oder verschiedenartige Kompressionstechniken für dreidimensionale Datenfelder integrieren. Auch können analog zu den Oberflächenobjekten beliebige, prozedural definierte, Objekte realisiert werden, mit denen Phänomene wie Wolken, Flammen oder auch fein strukturierte Materialien wie Wolle oder Haare adäquat darstellbar sind. Als diesbzgl. weiterführende Literatur sei hier auf das Buch von [Ebe94] verwiesen, in welchem auf die Modellierung solcher Phänomene bzw. Materialien ausführlich eingegangen wird.

Neben verschiedenen Typen von Volumendaten, bietet es sich auch an, das System um weitere Shading- und/oder Beleuchtungsmodelle zu erweitern, welche im Rahmen neuer Volumenobjekte realisiert werden können. Für die informationsorientierte Visualisierung wissenschaftlicher Daten wäre es zum Beispile sinnvoll, vereinfachte Beleuchtungsmodelle zu implementieren, mit denen eine effizientere, d.h. schnellere Darstellung der Daten möglich ist. Auch sind weitere Klassifizierungsverfahren denkbar, die auf die Visualisierung ganz bestimm-

ter Strukturen abgestimmt sind. Ist hingegen eine photorealistische Visualisierung gewünscht, so könnten Modelle implementiert werden, welche unter Anwendung von Monte-Carlo-Verfahren Mehrfachstreuungen in partizipierenden Medien berücksichtigen. Motiviert wird letzteres vor allen Dingen in Bezug auf eine realitätsnahe Darstellung atmosphärischer Phänomene.

Anhang A

Szenenbeschreibungen

Szenenbeschreibung für Abbildung 4.1

```
COLORRANGE 255
EYEP      (0,0,200)
LOOKP     (0,0,0)
UP        (0,1,0)
FOV       <35,35>
SCREEN    <300,300>
//-----
BACKGROUND (30,30,80)
AMB_LIGHT  (60,60,60)
POS_LIGHT  (255,255,255) (-50,0, 200)
POS_LIGHT  (255,255,255) (0,0,-100)
//-----
SURFACE 1 0.7; (255,255,255) 0.0; (255,255,255) 1.0, 15
SURFACE 2 0.7; (255,255,255) 0.7; (255,255,255) 0.0, 15
VOLDATA 1 REGULAR_GRID "voldata/Mrbrain" // bbox is inside unit cube
                                           // centered at (0,0,0)
//-----
ROT_Y(-40)
PARALLEL 1 (-80,-100,-90) (-80,-100, 90) (-80,100,-90)

SCALE(200,200,200)*ROT_X(180)*ROT_Y(-50)*TRANS(30,10,10)
ISOSURFACE 2 1 <0.17, 0.78> // SURFACE_NO, VOLDATA_NO, THRESHOLD, STEPSIZE
//-----
ENDFILE
```

Szenenbeschreibung für Abbildung 4.2

```

COLORRANGE 255
EYEP      (0,0,200)
LOOKP     (0,0,0)
UP        (0,1,0)
FOV       <35,35>
SCREEN    <300,300>
//-----
BACKGROUND (30,30,80)
AMB_LIGHT  (60,60,60)
POS_LIGHT  (255,255,255) (0,0,200)
//-----
SURFACE 1 0.7; (255,255,255) 0; (255,255,255) 1.0, 15
VOLDATA 1 REGULAR_GRID "voldata/Engine" // bbox is inside unit cube
// centered at (0,0,0)
//-----
ROT_X(90)*TRANS(0,-145,0)
PARALLEL 1 (-130,-80,-80) (-130,150,-80) (130,-80,-80)

SCALE(225,225,225)*ROT_X(180)*ROT_Z(90)*TRANS(5,20,-30)
LEVOY_VOLUME

VOLDATA_NO 1 // number of volume data object
STEP_SIZE 0.87 // sample step size
SAMPLE_EXACT 1 // "exact" sampling (trilinear interp.)
ATTENUATION 0 // no attenuation of shadow rays

AMBIENT 30/255 // ambient reflection coefficient
DIFFUSE 1.0 // diffuse reflection coefficient

// piecewise linear transfer function maps scalar value to opacity
SCALAR_OPACITY_RAMP ( 0/255, 0.000) ( 67/255, 0.000) ( 93/255, 0.157)
(180/255, 0.157) (200/255, 1.000) (255/255, 1.000)

// piecewise linear transfer function remaps gradient magnitude;
// needed for adjustment
GRADMAG_OPACITY_RAMP ( 0/221, 0.000) ( 60/221, 1.000) (221/221, 1.000)

// piecewise linear transfer function maps scalar value to diffuse
// color
SCALAR_COLOR_RAMP ( 0/255, ( 0, 0, 0)) ( 67/255, ( 0, 0, 0))
( 93/255, (200,200,255)) (180/255, (200,200,255))
(200/255, (240, 62, 22)) (255/255, (240, 62, 22))
//-----
ENDFILE

```

Szenenbeschreibung für Abbildung 4.3

```

COLORRANGE 255
EYEP      (0,0,200)
LOOKP     (0,0,0)
UP        (0,1,0)
FOV       <35,35>
SCREEN    <300,300>
//-----
BACKGROUND (30,30,80)
AMB_LIGHT  (60,60,60)
POS_LIGHT  (255,255,255) (0,0,200)
//-----
SURFACE 1 0.7; (255,255,255) 0; (255,255,255) 1.0, 15
VOLDATA 1 REGULAR_GRID "voldata/Cthead" // bbox is inside unit cube
// centered at (0,0,0)
//-----
ROT_X(90)*TRANS(0,-130,0)
PARALLEL 1 (-130,-80,-80) (-130,150,-80) (130,-80,-80)

SCALE(155,155,155)*ROT_X(90)*ROT_Y(110)*TRANS(0,40,0)
LEVOY_VOLUME

VOLDATA_NO 1 // number of volume data object
STEP_SIZE 0.6 // sample step size
SAMPLE_EXACT 1 // "exact" sampling (trilinear interp.)
ATTENUATION 0 // no attenuation of shadow rays

AMBIENT 30/255 // ambient reflection coefficient
DIFFUSE 1.0 // diffuse reflection coefficient

// piecewise linear transfer function maps scalar value to opacity
SCALAR_OPACITY_RAMP ( 0/255, 0.000) ( 80/255, 0.380) (135/255, 1.000)
(255/255, 1.000)

// piecewise linear transfer function remaps gradient magnitude;
// needed for adjustment
GRADMAG_OPACITY_RAMP ( 0/221, 0.000) ( 70/221, 1.000) (221/221, 1.000)

// piecewise linear transfer function maps scalar value to diffuse
// color
SCALAR_COLOR_RAMP ( 0/255, (100,200,255)) ( 90/255, (100,200,255))
(130/255, (187,187,111)) (255/255, (187,187,111))
//-----
ENDFILE

```

Szenenbeschreibung für Abbildung 4.4

```

COLORRANGE 255
EYEP      (0,0,200)
LOOKP     (0,0,0)
UP        (0,1,0)
FOV       <35,35>
SCREEN    <300,300>
//-----
BACKGROUND (30,30,80)
AMB_LIGHT  (60,60,60)
POS_LIGHT  (255,255,255) (0,0,200)
ATT_POS_LIGHT (255,255,255) (-5000,20000,-15000)
//-----
SURFACE 1 0.7: (100,200,255) 0.7: (255,255,255) 0.0, 15
VOLDATA 1 REGULAR_GRID "voldata/Cthead" // bbox is inside unit cube
// centered at (0,0,0)
//-----
ROT_X(90)*TRANS(0,-130,0)
PARALLEL 1 (-130,-80,-80) (-130,150,-80) (130,-80,-80)

SCALE(155,155,155)*ROT_X(90)*ROT_Y(90)*TRANS(-30,40,0)
LEVOY_VOLUME

VOLDATA_NO 1 // number of volume data object
STEP_SIZE 0.6 // sample step size
SAMPLE_EXACT 1 // "exact" sampling (trilinear interp.)
ATTENUATION 1 // attenuation of shadow rays

CUTREGION BOX (-0.5,0,0) <0.5,0.5,0.5>
// cut box out of unit cube

AMBIENT 30/255 // ambient reflection coefficient
DIFFUSE 1.0 // diffuse reflection coefficient

// piecewise linear transfer function maps scalar value to opacity
SCALAR_OPACITY_RAMP ( 0/255, 0.000) ( 80/255, 0.380) (135/255, 1.000)
(255/255, 1.000)

// piecewise linear transfer function remaps gradient magnitude;
// needed for adjustment
GRADMAG_OPACITY_RAMP ( 0/221, 0.000) ( 70/221, 1.000) (221/221, 1.000)

// piecewise linear transfer function maps scalar value to diffuse
// color
SCALAR_COLOR_RAMP ( 0/255, (100,200,255)) ( 90/255, (100,200,255))
(130/255, (187,187,111)) (255/255, (187,187,111))
//-----
ENDFILE

```


Szenenbeschreibung für Abbildung 4.6

```

COLORRANGE 255
EYEP      (0,0,100)
LOOKP     (0,0,0)
UP        (0,1,0)
FOV       <35,35>
SCREEN    <255,255>
//-----
BACKGROUND (30,30,30)
ATT_POS_LIGHT (255,255,255) (100,200,100)
AMB_LIGHT   ( 70, 70, 70)
//-----
SURFACE 1 0.7; (100,255,100) 0.4; (255,255,255) 0.0, 1
SURFACE 2 0.4; (255, 0, 70) 0.4; (180,150, 70) 0.0, 1
VOLDATA 1 REGULAR_GRID 120 120 120
          SMOKESTREAM TURBULENCE_SCALE 12 HEIGHT_BOTTOM -0.7
          // regular grid (120x120x120) initialized by "smoke-data"
          // defined by scale factor for turbulence function
          // and bottom of smoke column along y-axis
          // ... and a lot of other parameters listed in the source code ...
//-----
GEN_CYLINDER 2 (-2,-45,-2) (-2,-35,-2) <15, 5>
BOX          1 (0,-45,0) <100,2,100>

SCALE(100,60,100)
BLINN_VOLUME

          VOLDATA_NO          1
          STEPSIZE             1.0
          SAMPLE_EXACT         1
          OPTICAL_DEPTH        0.5
          ATTENUATION          1
          PHASE_FUNCTION       2
          AMBIENT              0.0
          DIFFUSE              1.0
          SCALAR_COLOR_RAMP    (0.0, (220,220,255)) (1.0, (220,220,255))
//-----
ENDFILE

```

Anhang B

Das Dateiformat von HP

Wie in Abschnitt 3.3.1 angesprochen, können Volumendaten-Objekte vom Typ `t_VDRegularGrid` auch mit einem auf einem Festspeicher befindlichen Volumen-Datensatz initialisiert werden. Das Dateiformat, in welchem die Volumendaten abgelegt werden müssen entspricht dem, welches die Firma Hewlett Packard (HP) im Rahmen des Voxelator-Projekts [Hew97] benutzt. Aufgrund des Prototypenstadiums, in dem sich sowohl das Projekt als auch das Dateiformat befindet, existiert nur eine unzureichende Dokumentation darüber, sodaß das Dateiformat im folgenden genauer beschrieben werden soll.

Ein Datensatz ist im Binärformat abgelegt und setzt sich aus einem Header sowie den eigentlichen Volumendaten zusammen. Der Header ist von variabler Länge und beinhaltet die in Tabelle B.1 aufgeführten Elemente. Es gilt dabei zu beachten, daß die einzelnen Bytes eines Elements nach aufsteigender Wertigkeit abgelegt sind. Für ein besseres Verständnis sei weiterhin angemerkt, daß in der anschließenden Beschreibung ein Volumen-Datensatz als eine Folge von Bildern verstanden wird.

- `magic_number`
Eine Versionsnummer, welche zur Identifikation der Datei dient. Gültige Versionsnummern sind 192837465 sowie 192837466. Erstere Nummer besagt, daß der Header keine Informationen bzgl. Skalierung und Rotation beinhaltet.
- `header_length`
Die Länge des Headers in Bytes.

Element	Typ	Bytes
magic_number	integer	4
header_length	integer	4
width	integer	4
height	integer	4
images	integer	4
bits_per_voxel	integer	4
index_bits	integer	4
scaleX	float	4
scaleY	float	4
scaleZ	float	4
rotX	float	4
rotY	float	4
rotZ	float	4
manufacturer	char*	1/Zeichen
original_name	char*	1/Zeichen

Tabelle B.1: Elemente des Headers

bits_per_voxel	index_bits	intensity_bits
8	0	8
16	0	16
	4	12
	8	8
32	16	16

Tabelle B.2: Zugelassene Kombinationen von Index- und Intensitätsbits

- `width`
Die Breite eines Bildes in Pixeln.
- `height`
Die Höhe eines Bildes in Pixeln.
- `images`
Die Anzahl der Bilder des Volumens.
- `bits_per_voxel`
Die Anzahl der Bits eines jeden Datenwertes im Volumen. Dieser setzt sich aus einem Index- und einem Intensitäts-Wert zusammen. Ersterer wird momentan bei der Visualisierung ignoriert. Nach der Definition des Formats sind nur die in Tabelle B.2 aufgeführten Kombinationen von Index- und Intensitäts-Bits zugelassen.
- `index_bits`
Anzahl der Index-Bits pro Datenwert.
- `scaleX, scaleY, scaleZ`
Definieren die Skalierung in X-, Y- bzw. Z-Richtung und geben somit das Verhältnis zwischen Höhe, Breite und Tiefe des Volumens an.
- `RotX, RotY, RotZ`
Definieren die Rotation um X-, Y- bzw. Z-Achse. Diese Werte werden ignoriert, da die Rotation (und Translation) der Daten extern festgelegt wird.
- `manufacturer`
Eine ASCII-Zeichenkette, welche den Hersteller des Datensatzes angibt.
- `original_name`
Eine ASCII-Zeichenkette, welche den ursprünglichen Namen des Datensatzes angibt.

Zur besseren Handhabung der Datensätze, insbesondere im Rahmen der Klasse `t_VDRegularGrid`, wurde die Klasse `t_VDHPInputStream` eingeführt, welche die für den Zugriff auf die Daten notwendigen Funktionen nach außen hin

vollständig kapselt. Wird ein Objekt vom Typ `t_VDHPInputStream` instanziiert, so werden zunächst automatisch die Header-Informationen eingelesen. Der Zugriff auf die Daten kann dann mittels des Eingabeoperators erfolgen.

Literaturverzeichnis

- [Ben97] Heinzgerd Bendels. Eine topologische Datenstruktur und ihre Anwendungen im 3D-Graphiksystem MRT. Master's thesis, University of Bonn, Bonn, Germany, March 1997.
- [Bli82] James F. Blinn. Light Reflection Functions for Simulation of Clouds and Dusty Surfaces. *Computer Graphics*, 16(3):21–29, July 1982.
- [BS84] R.E. Barnhill and S.E. Stead. Multistage Trivariate Surfaces. *Rocky Mountain Journal of Mathematics*, 14(1):103–118, 1984.
- [CW93] M.F. Cohen and J.R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, Boston, 1993.
- [Due88] M.J. Duerst. Letters: Additional Reference to Marching Cubes. *Computer Graphics*, 22(2), 1988.
- [Ebe94] D. Ebert et al. *Texturing and Modeling - A Procedural Approach*. Academic Press, Boston, 1994.
- [Fel92] Dieter W. Fellner. *Computergrafik*, volume 58 of *Reihe Informatik*. BI-Wissenschaftsverlag, Mannheim, 2 edition, 1992.
- [Fel94] Dieter W. Fellner. MRT++ Design Issues and Brief Reference. University of Bonn, Department of Computer Science, Bonn, Germany, May 1994.
- [Fis95] Martin Fischer. Reference Pointers for Class Hierarchies. Technical Report IAI-TR-95-12, University of Bonn, Department of Computer Science, Bonn, Germany, August 1995.

- [FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics - Principles and Practice*. The Systems Programming Series. Addison–Wesley, 2 edition, 1990.
- [Gar90] Michael P. Garrity. Raytracing Irregular Volume Data. *Computer Graphics (Proc. San Diego Workshop on Volume Visualization)*, 24(5):35–40, November 1990.
- [Gla95] Andrew. S Glassner. *Principals of Digital Image Synthesis*. Morgan Kaufmann, San Francisco, 1995.
- [Gro96] Alwin Groene. *RayViS: An Object Oriented Visualization System based on Ray Tracing*. PhD thesis, University of Tuebingen, Tuebingen, November 1996.
- [Hew97] Hewlett Packard. *The Voxelator Project*, 1997. Internet Address: <http://hpcc997.external.hp.com/wsg/vox.html>.
- [Kaj86] James T. Kajiya. The Rendering Equation. *Computer Graphics (Proc. SIGGRAPH'86)*, 20:143–150, 1986.
- [KvH84] James T. Kajiya and Brian P. von Herzen. Ray Tracing Volume Densities. *Computer Graphics (Proc. SIGGRAPH'84)*, 18(3):165–178, July 1984.
- [Lac95] Philippe Lacroute. *Fast Volume Rendering Using a Shear–Warp Factorization of the Viewing Transformation*. PhD thesis, Stanford University, Stanford, CA, September 1995.
- [LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proc. SIGGRAPH'87)*, 21(4):163–169, July 1987.
- [Lev88] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.
- [Max95] Nelson Max. Optical Models for Volume Rendering. In M. Goebel, H. Mueller, and B. Urban, editors, *Visualization in Scientific Computing*, pages 35–40, Springer, 1995.

- [MS93] Heinrich Mueller and Michael Stark. Adaptive Generation of Surfaces in Volume Data. *The Visual Computer*, 9:182–199, 1993.
- [Mue97] Gordon Mueller. Beschleunigung strahlbasierter Rendering-Algorithmen. Master's thesis, University of Bonn, Bonn, Germany, 1997.
- [RT87] Holly E. Rushmeier and Ken E. Torrance. The Zonal Method for Calculating Light Intensities in the Presence of a Participating Medium. *Computer Graphics (Proc. SIGGRAPH '87)*, 21:293–302, 1987.
- [Sab88] Paolo Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *Computer Graphics*, 22(4):51–58, August 1988.
- [SH92] Robert Siegel and John R. Howell. *Thermal Radiation Heat Transfer*. Hemisphere Publishing, New York, 3 edition, 1992.
- [Sil93] Silicon Graphics, Inc., Mountain View, California. *The OpenGL Reference Manual*, 1993.
- [SK90] Don Speray and Steve Kennon. Volume Probes: Data Exploration on Arbitrary Grids. *Computer Graphics (Proc. San Diego Workshop on Volume Visualization)*, 24(5):5–12, November 1990.
- [SP94] Francois X. Sillion and Claude Puech. *Radiosity and Global Illumination*. Morgan Kaufmann, San Francisco, CA, 1994.
- [Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 2 edition, 1991.
- [Sun93] Sun Microsystems, Inc., Mountain View, California. *The Solaris XGL Graphics Library*, 1993.
- [UK88] Craig Upson and Michael Keeler. V-Buffer: Visible Volume Rendering. *Computer Graphics*, 22(4):59–64, August 1988.
- [WC90] Jane Wilhelms and Judy Challinger. Direct Volume Rendering of Curvilinear Volumes. *Computer Graphics (Proc. San Diego Workshop on Volume Visualization)*, 24(5):41–47, November 1990.

- [WvG90] Jane Wilhelms and Allen van Gelder. Topological Considerations in Isosurface Generation. *Computer Graphics*, 24(5):79–86, 1990.
- [WW92] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques - Theory and Practice*. ACM Press, New York, 1 edition, 1992.