

# **Masterarbeit**

vorgelegt an der Hochschule  
für angewandte Wissenschaften Fachhochschule Würzburg-Schweinfurt  
in der Fakultät Informatik und Wirtschaftsinformatik  
zum Abschluss eines Studiums im Studiengang Informationssysteme

Studienschwerpunkt: Mobile Computing

## **Skalierbares Echtzeit Volumen-Rendering**

Angefertigt in der Fakultät für Informatik und Wirtschaftsinformatik der Fachhochschule  
Würzburg-Schweinfurt

-  
-  
-  
Abgabetermin:

Eingereicht von: Christian Ofenberg aus -



Hiermit versichere ich, dass ich die vorgelegte Masterarbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Würzburg, den

(Unterschrift)

## Übersicht

In der Medizin, bei wissenschaftlichen Simulationen oder in der Filmindustrie gehört die Erfassung und Verarbeitung von Volumendaten zum Standardrepertoire. Mit den kontinuierlich weiter ansteigenden Leistungsressourcen handelsüblicher Grafikkarten wird der Einsatz von volumetrischen Daten auch für die Videospielindustrie attraktiv.

Die Einsatzgebiete und Quellen von Volumendaten sind vielfältig, gleiches gilt für deren Visualisierungsmöglichkeiten. Die Anforderungen an das Volumen-Rendering reichen von einer interaktiven bis hin zu einer qualitativ hochwertigen Darstellung ohne Echtzeitanforderungen. Im medizinischen Umfeld ist eine wahrheitsgetreue Visualisierung von entscheidender Bedeutung. In der Unterhaltungsbranche hingegen sind ästhetische Faktoren ausschlaggebend.

Diese Arbeit soll aufzeigen, dass die zahlreichen Anforderungen durch ein skalierbares System abgedeckt werden können. Der Volumen-Rendering Prozess wird analysiert und in einzelne Shaderfunktionen mit wohldefinierten Schnittstellen zerlegt. Ein Compositingsystem stellt anhand der jeweiligen Anforderungen dynamisch verschiedene Implementierungen der Shaderfunktionen zu einem auf einer Grafikkarte lauffähigen Shader zusammen. Dies fördert die Wiederverwendbarkeit einzelner Komponenten und ermöglicht eine feingranulare Konfiguration der Volumen-Visualisierung.

## Abstract

In medicine, scientific simulations or in the film industry, the collection and processing of volumetric data is a common task. With the continuously increasing resources of commercially available graphics cards, the use of volume data gets also attractive for the video game industry.

The applications and sources of volume data are manifold, the same applies to their visualization approaches. The requirements for volume-rendering range from interactive to a high-quality representation without real-time demands. In the medical environment, an exact visualization is important while the entertainment industry is driven by aesthetic factors.

This work shows that multiple demands can be covered by a scalable system. The volume-rendering process is analysed and decomposed into individual shader functions with well defined interfaces. Basing on the specific requirements, a compositing system dynamically combines different implementations of the shader functions to a shader which can be executed on a graphics card. This enhances the reusability of individual components and enables a fine-grained configuration of the volume visualization.

# Inhaltsverzeichnis

|  |           |
|--|-----------|
| <b>1 Einleitung</b>                              | <b>1</b>  |
| 1.1 Motivation . . . . .                         | 1         |
| 1.2 Zielsetzung . . . . .                        | 3         |
| 1.3 Gliederung der Arbeit . . . . .              | 3         |
| <b>2 Grundlagen des Volumen-Rendering</b>        | <b>5</b>  |
| 2.1 Datensatz . . . . .                          | 5         |
| 2.2 Volumen-Rendering . . . . .                  | 12        |
| <b>3 Integration in eine 3-D-Engine</b>          | <b>15</b> |
| 3.1 Datensatz im Speicher . . . . .              | 16        |
| 3.2 Szene . . . . .                              | 19        |
| 3.3 Szenen-Renderer . . . . .                    | 21        |
| 3.4 Strahlgenerierung . . . . .                  | 27        |
| <b>4 Volumen-Renderer Shaderfunktionen</b>       | <b>37</b> |
| 4.1 Strahlschnitt . . . . .                      | 37        |
| 4.2 Künstliches Rauschen . . . . .               | 41        |
| 4.3 Strahlverfolgung . . . . .                   | 44        |
| 4.4 Schnitt innerhalb des Volumens . . . . .     | 48        |
| 4.5 Rekonstruktion der Volumendaten . . . . .    | 51        |
| 4.6 Shading . . . . .                            | 54        |
| 4.7 Klassifikation . . . . .                     | 57        |
| 4.8 Gradientenschätzung . . . . .                | 63        |
| 4.9 Lokale Beleuchtung . . . . .                 | 67        |
| <b>5 Experimente</b>                             | <b>71</b> |
| 5.1 Versuchsumgebung und Datenmaterial . . . . . | 71        |

|  |            |
|--|------------|
| 5.2 Leistungsergebnisse . . . . .                                      | 74         |
| 5.3 Fehlerquellen . . . . .  | 83         |
| <b>6 Zusammenfassung</b>   | <b>85</b>  |
| <b>7 Ausblick</b>  | <b>89</b>  |
| <b>A Verwendete Datensätze</b>   | <b>91</b>  |
| <b>B Direct3D Shadermodell vs. OpenGL-Versionen vs. GLSL-Versionen</b> | <b>95</b>  |
| <b>Literaturverzeichnis</b>  | <b>97</b>  |
| <b>Bilderverzeichnis</b>   | <b>107</b> |
| <b>Tabellenverzeichnis</b>   | <b>109</b> |
| <b>Quellcodeverzeichnis</b>  | <b>111</b> |
| <b>Hilfsmittelverzeichnis</b>  | <b>113</b> |
| <b>Abkürzungsverzeichnis</b>   | <b>115</b> |

# Kapitel 1

## Einleitung

### 1.1 Motivation

Volumen-Rendering wurde ursprünglich durch medizinische und wissenschaftliche Problemstellungen motiviert. In der Filmindustrie gehören Volumen-basierende visuelle Effekte ebenfalls seit langem zum Standardrepertoire der Effektspezialisten. Beide Anwendungsgebiete haben gemeinsam, dass üblicherweise ausreichende finanzielle Mittel für Hochleistungshardware bis hin zu eigenen Rechenzentren zur Verfügung stehen. Im medizinischen Umfeld sind in der Regel interaktive Visualisierungen üblich, offline gerenderte, qualitativ hochwertige Volumen-Renderings sind allerdings ebenfalls gebräuchlich. In der Filmindustrie hingegen existieren keine Echtzeitanforderungen für das Endergebnis. Während der Produktion wird mit vereinfachten Daten- und Visualisierungsmethoden gearbeitet um eine schnelle Iteration bei der Verfeinerung der Abläufe zu gewähren. Das finale Rendering hingegen kann Stunden bis hin zu Tagen dauern.

Parallel zur stetigen Weiterentwicklung der Computerhardware wurden ebenfalls Scanner zum Erheben von Volumendaten kontinuierlich weiterentwickelt, dies führt zu größeren Datenmengen. In Abbildung 1.1 ist ein 876 MiB großer medizinischer Datensatz zu sehen. Gleichzeitig wachsen die Ansprüche an die Bildqualität der Visualisierung. Neben höheren Auflösungen gehören komplexe Beleuchtungsmodelle mittlerweile ebenfalls zum Standard, während je nach Anwendungsgebiet die Anforderung der Interaktivität gleich bleibt. Dieses ständige Wettrüsten zwischen leistungsfähigerer Hardware und wachsenden Ansprüchen brachte eine nahezu unüberschaubare Anzahl an Volumen-Rendering Algorithmen zum Vorschein.

In der vergangenen Dekade fand eine kleine Hardware Revolution statt. Ursprünglich primär für die Beschleunigung von polygonbasierenden Videospielen entwickelte Grafikkarten durchliefen eine rasche Entwicklung, von anfangs fest verdrahteten Funktionalitäten bis hin zum frei programmierbaren hochleistungsfähigen Consumer-Vektorrechner. Bereits zu den Anfängen



Bild 1.1: Volumen-Rendering eines 876 MiB großen medizinischen Datensatzes. Volumendatensatzes *Schwein*.

der freien Graphics Processing Unit (GPU) Programmierbarkeit fand die Grafikhardware bereits reges Interesse in der medizinischen Visualisierung. Trotz anfänglich stark begrenzter Grafikkartenspeicherressourcen entstanden zahlreiche GPU-basierende Volumen-Renderer. Parallel zur rasanten GPU Weiterentwicklung wurden kontinuierlich bereits bestehende Central Processing Unit (CPU)-Algorithmen auf die GPU übertragen oder neue Volumen-Rendering verfahren entwickelt, welche sich die hochparallelisierte GPU-Architektur zu nutze machen. Zum aktuellen Zeitpunkt sind mobile *Gamer-Notebooks* erhältlich die in der Lage sind, medizinische Datensätze in Echtzeit darzustellen. Die gleichen Datensätze bereiteten noch vor 10 Jahren stationärer Spezialhardware Probleme. Gleichzeitig gewannen mobile Endgeräte wie beispielsweise Smartphones oder Tablets an Popularität, die ersten Volumen-Renderer für die Hosentasche ließen ebenfalls nicht lange auf sich warten.

Dank der nun verfügbaren Hardweareressourcen und des breitgefächerten Arsenals an Volumen-Rendering Literatur aus dem medizinischen und wissenschaftlichen Umfeld entdeckt allmählich ebenfalls die Videospielindustrie die Nutzung von Volumendaten für sich. Neben offensichtlichen Einsatzgebieten wie Feuer oder Nebel, die über polygonbasierende Verfahren nur umständlich realisierbar sind, finden Volume-Rendering Technologien ebenfalls verstärkt als Hilfsmittel Verwendung. Beispiel hierfür ist eine Speicherung von Lichtinformationen in einem Volumen.

Das Feld des Volumen-Rendering reicht von interaktiven Visualisierungen bis hin zur qualitativ hochwertigen, jedoch nicht echtzeitfähigen Bildsynthese. Gleichzeitig wird hierbei unterschiedlich leistungsfähige Consumer-Hardware bis hin zu teuren Hochleistungsrechnern eingesetzt. Die Anforderungen variieren ebenfalls. Während in der Film- und Videospielindustrie vor allem ästhetische Faktoren eine Rolle spielen, ist im medizinischen Umfeld hingegen eine möglichst wahrheitsgetreue Visualisierung von entscheidender Bedeutung. Diese widersprüchlichen Anforderungen bedeuten jedoch nicht zwangsläufig, dass für jeden Anwendungsfall von Grund auf komplett eigenständige Softwaresysteme entwickelt werden müssen.

## 1.2 Zielsetzung

Die Zielsetzung dieser Arbeit besteht in der Realisierung eines skalierbaren Echtzeit Volumen-Rendering Systems. Der Volumen-Rendering Prozess soll anhand verfügbarer Literatur analysiert und in einzelne Shaderfunktionen mit wohldefinierten Schnittstellen zerlegt werden. Das skalierbare System hat zur Aufgabe, eine Vielzahl von Volumen-Rendering Algorithmen miteinander zu verbinden und jeweils mehrere Lösungsmöglichkeiten anzubieten. Hierdurch soll es ermöglicht werden den Volumen-Renderer feingranular zu Konfigurieren und somit verschiedene Anforderungen in einem einzigen gemeinsamen System umzusetzen.

Durch die Bereitstellung unterschiedlicher Umsetzungen der Shaderfunktionen soll es ermöglicht werden, Fehler oder Leistungsanomalien in Volumen-Rendering Teilschritten aufzudecken sowie die Wiederverwendbarkeit einzelner Komponenten zu ermöglichen. Weitere Implementierungen einzelner Shaderfunktionen sollen im Nachhinein hinzufügbar sein, ohne das hierzu sämtliche Shader angepasst werden müssen.

Um die Nutzung unterschiedlicher Hardware- und Betriebssysteme zu ermöglichen, soll die realisierte Lösung prinzipiell Plattform unabhängig und erweiterbar sein. Hierzu muss das zu erstellende System die Speicherung der Volumendaten in verschiedenen Texturtypen zulassen. Des Weiteren soll eine Unterstützung mehrerer Shadersprachen vorhanden sein. Das System soll es ermöglichen nachträglich Unterstützung für beispielsweise Open Graphics Library (OpenGL) for Embedded Systems (OpenGL ES) 2.0 hinzuzufügen.

## 1.3 Gliederung der Arbeit

Dank der Videospielbranche ist polygonbasierende Echtzeitgrafik außerhalb der medizinischen und wissenschaftlichen Welt bekannter als Volumen-basierende Echtzeitgrafik. Daher sollen im

folgenden 2. Kapitel die Grundlagen des Themengebietes des Volumen-Renderings vermittelt werden, ohne dabei bereits zu tief in die Materie einzusteigen.

Bei Volumen-Rendering handelt es sich um einen Begriff für eine breite Palette an Algorithmen. Gleichzeitig ist es heutzutage nicht mehr unüblich, dass eine erstellte Softwarelösung auf unterschiedlichen Hardwaretypen und Betriebssystemen lauffähig sein muss. Es ist daher sinnvoll, das umfangreiche Themengebiet der plattformübergreifenden Softwareentwicklung auszublenden indem als technisches Grundgerüst ein bereits existierendes Framework verwendet wird. Kapitel 3 bildet die Verbindungsschnittstelle zwischen der Theorie und der praktischen Realisierung eines konkreten Volumen-Renderers.

Das Herzstück dieser Arbeit sind die in Kapitel 4 vorgestellten Volumen-Renderer Shaderfunktionen. Durch die Integration in ein bestehendes Framework, kann der Schwerpunkt auf die frei kombinierbaren Bestandteile des zur Laufzeit dynamisch erzeugten GPU-Shaders eingegangen werden. Die breite Auswahl an Volumen-Renderer Literatur wird auf einzelne Shaderfunktionsbausteine verteilt und es werden Zusammenhänge zwischen den verschiedenen Algorithmen hergestellt.

Der resultierende skalierbare Volumen-Renderer bietet zahlreiche Konfigurationsmöglichkeiten. Die Bandbreite an Anforderungen von interaktiver Bildwiederholrate, dem Echtzeit-Rendering, bis hin zu qualitativ hochwertiger Bildsynthese werden mit dem gleichen System erfüllt. Aufgrund dessen ist eine hohe Anzahl an Permutationen der einzelnen Funktionsbausteine möglich. Die exakte Erfassung aller Kombinationsmöglichkeiten würde jedoch den Rahmen dieser Arbeit sprengen. Daher geht Kapitel 5 unter anderem auf hervorstechende Algorithmenkombinationen ein.

Der Zusammenfassung dieser Arbeit in Kapitel 6 folgt das abschließende Kapitel 7, welches einen Ausblick über Erweiterungsmöglichkeiten des vorgestellten skalierbaren Echtzeit Volumen-Renderers geben soll.

# Kapitel 2

## Grundlagen des Volumen-Rendering

Außerhalb der traditionellen Volumen-Rendering Einsatzgebiete Medizin und Wissenschaft, sowie für visuelle Effekte in der Filmindustrie, besitzen Volumen-basierende Verfahren einen geringen Bekanntheitsgrad. Die Zielsetzung dieses Kapitels besteht daher im Vermitteln von Grundlagen, die zum Verständnis der weiteren Kapitel benötigt werden. Der Schwerpunkt dieser Arbeit liegt auf der Visualisierung von Volumendaten, jedoch sollen Hintergründe über diese Daten nicht unerwähnt bleiben.

### 2.1 Datensatz

**Was ist ein Voxel?** Die Elemente einer 2-D-Rastergrafik sind allgemeinen unter dem Kunstwort *Pixel* bekannt. Dies ist eine verkürzte Schreibweise von *Picture Element*. Die Position eines Bildpunktes ist durch das Raster vorgegeben und wird nicht zusätzlich gespeichert. Sobald ein derart digitalisiertes Bild zum Zwecke der Texturierung in den Grafikkartenspeicher geladen wird, ist der Begriff *Texel*, ausgeschrieben *Texture Element*, gängig.

Das gerade beschriebene Prinzip lässt sich auf *Voxel* übertragen, die in einer 3-D-Rastergrafik gespeichert werden. Der Begriff *Voxel* ist hierbei, je nach Literaturquelle, eine verkürzte Schreibweise von *Volume Pixel* [Sha99] oder *Volume Element* [Had06, S. 17]. Die Elemente einer im Grafikkartenspeicher abgelegten 3-D-Textur werden weiterhin als *Texel* bezeichnet. Im Gegensatz zu einem *Pixel* besitzt ein *Voxel* oftmals keinen direkt darstellbaren Farbwert, sondern nur einen einzigen Skalarwert. Der Skalarwert kann für eine physikalische Eigenschaft wie Geschwindigkeit, Temperatur, Druck oder Dichte stehen [Ray99].

In der Literatur existieren zahlreiche Namen für diese 3-D-Rastergrafik. So zählt [Kau94] die Namen *Volume Buffer*, *Cubic Frame Buffer* und *3D Raster* auf, während [Wre10] die Namen *Voxel Grid*, *Voxel Volume* sowie *Voxel Buffer* verwendet. In Abhängigkeit davon, ob Skalarwer-

te oder Vektoren gespeichert werden, sind ebenfalls die Namen *Scalar Field/Buffer/Grid* beziehungsweise *Vector Field/Buffer/Grid* gebräuchlich. Der Einfachheit halber wird im Rahmen dieser Arbeit von *Volumen* gesprochen.

Die Anordnung der Daten in einem gleichförmigen Raster ist nicht die einzige Form, wie Volumendaten repräsentiert werden können. Wetter- und Atmosphärenforscher verwenden unter anderem Radardaten, um Wetterphänomene zu studieren. Diese Daten liegen in einem sphärischen Koordinatensystem vor und müssen für Volume-Rendering entsprechend umgerechnet werden [Ril06, Con11]. Neben der Datenordnung ist für die verwendeten Datenstrukturen ebenfalls die Datendichte relevant. Für dünn besetzte Volumendaten lässt sich beispielsweise die, ursprünglich von Sony Pictures Imageworks entwickelte, Open-Source Bibliothek *Field3D*<sup>1</sup> einsetzen. Unterschiedliche Datenanordnungen, sowie die Berücksichtigung von großen jedoch nur dünn besetzten Datensätzen, verkomplizieren eine technische Realisierung nicht unwesentlich. Aufgrund dessen sollen im weiteren Verlauf nur Datensätze mit einem gleichförmigen Raster betrachtet werden.

**Polygone vs. Voxel** Unter der Zuhilfenahme von polygonbasierenden Verfahren lassen sich Oberflächen von Objekten zufriedenstellend annähern. Dank zahlreicher Texturen mit weiterführenden Oberflächeninformationen wie beispielsweise *Diffuse Maps*, *Normal Maps* oder *Specular Maps* lassen sich selbst kleine Unebenheiten und komplexe Lichtinteraktionen umsetzen. Moderne Grafikkarten beherrschen ebenfalls native Tessellation. Über eine Textur mit Höheninformationen lässt sich somit Hardwarebeschleunigtes *Displacement Mapping* realisieren. Dieses Verfahren war längere Zeit den visuellen Effekten für Filme vorbehalten und ermöglicht eine besseren Annäherung von Oberflächenstrukturen oder Oberflächenkonturen, ohne hierfür zusätzliche Vertex- und Polygondaten im 3-D-Modell zu speichern.

Unabhängig davon wie viele zusätzliche Dreiecke automatisch von der Grafikkarte erzeugt werden, lässt sich jedoch nicht alles effizient über Polygone realisieren. Feuer, Nebel, Rauch und Wolken besitzen keine festen, wohldefinierten Oberflächen. Ein klassischer Lösungsansatz besteht in der Nutzung sogenannter *Billboards*. Hierbei handelt es sich um mit einer Textur versehene Polygone, die Blickwinkelabhängig ausgerichtet werden. Auch ohne genaues Hinsehen wird dieser Trick schnell ersichtlich. Fell hingegen besteht aus einzelnen Haaren mit jeweils über Polygone beschreibbaren Oberflächen. Allerdings ist deren Anzahl derart hoch, dass eine direkte Lösung aus einzeln modellierten Haaren praktisch nicht nutzbar ist. Annäherungen von Fell über das Verfahren *Shells and Fins* [Len01] sind möglich, allerdings wird bei einer Nahansicht ersichtlich, dass keine einzelnen Haare vorhanden sind. All diese Lösungsansätze haben

---

<sup>1</sup><http://field3d.googlecode.com>

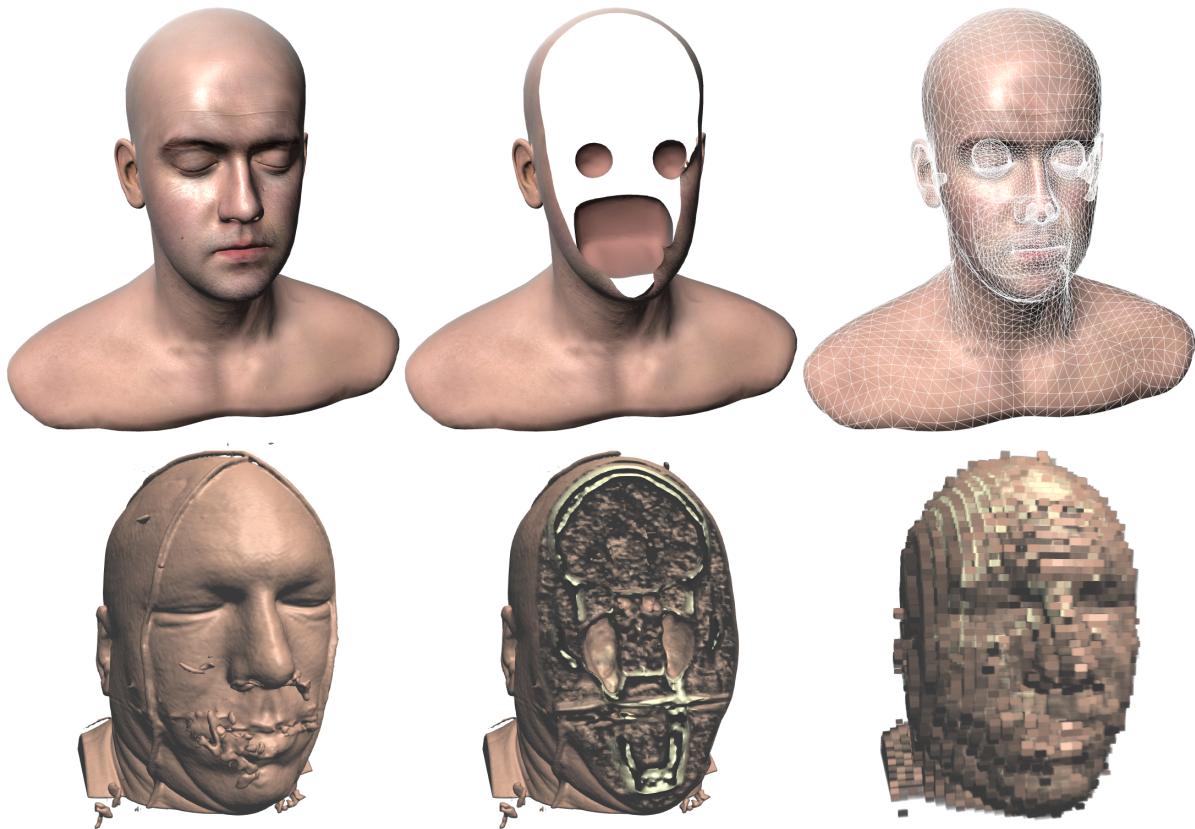


Bild 2.1: Polygonbasierender *Infinite Kopf*-Scan [IR-] oben, voxelbasierender Volumendatensatz *Head (Visible Male)* unten.

gemeinsam, dass für akzeptable Ergebnisse eine hohe Anzahl an übereinander geblendenen Polygone notwendig ist. Hierdurch wird der Grafikkarte sehr viel Füllratenleistung abverlangt.

Bei Wasser hingegen wird nur die Oberfläche über Polygone dargestellt. Sobald die Kamera diese durchdringt, werden weitere Effekte, wie beispielsweise Tiefennebel und eine Farbverfälschung aktiviert. Wird der Übergang zwischen über und unter Wasser genauer beobachtet, ist diese Umschaltung oftmals sichtbar. Fährt die Kamera in ein polygonales 3-D-Modell oder wird eine Schnittebene eingesetzt, so wird ersichtlich, dass es sich um einen Hohlkörper handelt. Abbildung 2.1 verdeutlicht dies anhand des polygonbasierenden *Infinite Kopf*-Scan [IR-] und des voxelbasierenden Volumendatensatzes *Head (Visible Male)*.

Volumen-basierende Verfahren sind für Objekte ohne eindeutig definierbare Oberfläche geeignet. Ebenfalls sind hiermit Anforderungen, wie die Visualisierung des Innenlebens für einen virtuellen Flug durch den Körper umsetzbar. Jedoch ist zu berücksichtigen, dass der Speicherbedarf von 3-D-Rastergrafiken deutlich größer ausfällt als bei polygonbasierenden Verfahren. Während für Polygone lediglich Vertices, Indices und 2-D-Texturen benötigt werden, müssen selbst bei spärlich besetzten Datensätzen umfangreiche Daten gespeichert werden. Abhilfe

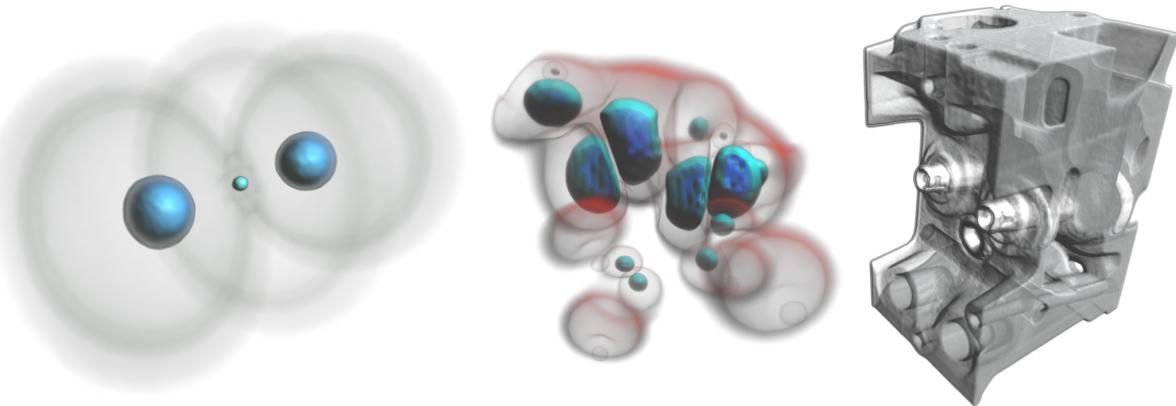


Bild 2.2: Die Volumen-Rendering Anwendungsfälle reichen von der Darstellung von Atomen bis hin zu ganzen Objekten. Volumendatensätze von links nach rechts: *Hydrogen*, *Neghip* und *Engine*.

schaffen Datenstrukturen wie beispielsweise *Sparse Voxel Octrees* [Lai10]. Weder Polygone noch Voxel sind eine Universallösung für alle denkbaren Anwendungsfälle.

**Datenquellen** Anhand der in 2.2 abgebildeten Auswahl an Anwendungsfällen für Volumen-Rendering ist zu erkennen, dass sehr unterschiedliche Quellen für Volumendaten existieren.

Zur Erfassung des *Visible Human* Datensatzes [Ack98] wurde ein destruktives Verfahren eingesetzt. Die Körper von zwei Spendern wurden tiefgefroren und anschließend in dünne Scheiben geschnitten. Die daraus resultierenden Bildebenden wurden einzeln abgelichtet und digitalisiert. Mit diesem Verfahren lassen sich hochauflösende Datensätze erstellen. Für den alltäglichen klinischen Einsatz am lebenden Patienten ist es allerdings nicht geeignet.

Scanner sind eine der wichtigsten Quellen für Volumendaten. Oftmals sind die bildgebenden Verfahren zweidimensional. Durch eine wiederholte Ausführung mit einer räumlichen Verschiebung des zu scannenden Objektes oder des Scanners, ergibt sich eine Bildsequenz für eine dreidimensionale Darstellung. Es existieren zahlreiche Gerätetypen wie beispielsweise Computed Tomography (CT), Magnetic Resonance Imaging (MRI), Positron Emission Tomography (PET) oder Digital Subtraction Angiography (DSA), um nur eine kleine Auswahl zu nennen. Je nach eingesetzter Gerätetechnik ergeben sich unterschiedlich zu interpretierende Daten, die in der Regel aus einem einzigen Skalarwert pro Voxel bestehen. Neben den eigentlichen Volumendaten muss ebenfalls bekannt sein, wie die Daten erfasst wurden. Die sogenannte *Modalität*. Anhand dieses Wissens lassen sich bei der späteren visuellen Rekonstruktion Farbwerte festlegen. Bei CT-Aufnahmen wird die Abschwächung von Röntgenstrahlung gemessen. Die Werte von Luft, Fettgewebe oder Knochen sind durch die Hounsfield-Skala bekannt. Die erfassten Datensätze werden im medizinischen Umfeld im offenen Standardformat Digital Ima-

ging and Communications in Medicine (DICOM) [Med] gespeichert. Neben den klassischen Einsatzgebieten Medizin und Molekularbiologie, werden die erwähnten Scanner ebenfalls in der Industrie eingesetzt. So lässt sich die Qualität von Materialien überprüfen, ohne diese beim Test zu zerstören [Mei00].

Auch mathematische Gebilde lassen sich als Quelle von Volumendaten verwenden. Nicht alle mathematischen Funktionen sind über zweidimensionale Diagramme anschaulich darstellbar. Eine Visualisierung über Volumen-Rendering stellt eine Alternative da.

Komplexe Sachverhalte werden in der Regel über Modelle angenähert, die Unmengen an Daten produzieren. Wetter-, Strömungs- und Flüssigkeitssimulationen dürften hierbei mitunter die bekanntesten Vertreter von Simulationen sein. Die so erzeugten Datenmengen lassen sich über Volumen-Rendering anschaulich darstellen und für Präsentationszwecke optisch ansprechend aufbereiten.

Die nach Ken Perlin benannte Rauschfunktion *Perlin-Noise* [Per02] bildet das Fundament für zahlreiche Algorithmen, die automatisiert Daten generieren. Hierüber lassen sich beispielsweise Volumendaten für pyroklastische Wolken generieren [Wre10]. Ein Hochsprachen-Shader für einen volumetrischen, über die Zeit hinweg automatisch animierbaren, prozeduralen Feuereffekt ist in [Gre05] zu finden.

Volumendaten können ebenfalls das Ergebnis einer Konvertierung sein. Punktwolken sowie polygonbasierende Geometrien lassen sich durch einen *Voxelization* genannten Prozess in Volumen umwandeln [Had06, S. 316]. Es existieren GPU-basierende Algorithmen, die selbst 3-D-Modelle mit mehreren Millionen Polygonen in weniger als einer Sekunde in eine Voxel-basierende Repräsentation umwandeln [Sch10]. Dieser Vorgang ist auf komplette Szenen übertragbar. Der stets wachsende Detailgrad der in Videospielen eingesetzten polygonalen Modelle, inklusive mehrerer hochauflösender Texturschichten, erfordert kontinuierlich mehr Grafikkartenspeicher. Die gesamte Grafikpipeline wird immer komplexer. Waren es vor einigen Jahren nur Vertexshader und Fragmentshader, so existieren mittlerweile fünf programmierbare Pipelinestufen. [Lai10] stellt daher die Frage in den Raum, ob eventuell nicht besser alles direkt als Voxel abgelegt wird. Dies könnte helfen, die Komplexität wieder zu senken.

Bei visuellen Effekten, wie diese in der Filmbranche gängig sind, müssen diese Daten selbst generiert werden. Dieser Vorgang ist unter dem Begriff *Volume-Modelling* bekannt und ist nach [Wre10, S. 11] ein nahezu endloses Themengebiet. Scanner Daten liefern eine Ausgangsbasis, weitere Rohdaten werden durch Konvertierung von polygonbasierenden Geometrien eingebbracht. Durch prozedurale Daten werden feingranulare Details eingefügt. Simulationen helfen unter anderem beim Animieren der umfangreichen Datenmengen. Da alle Möglichkeiten ausgeschöpft werden, existieren viele Methoden und Kombinationen um die Volumendaten zu erzeugen. Ein Überblick über Volumen-Modelling ist in [Had06, Kap.12] zu finden.

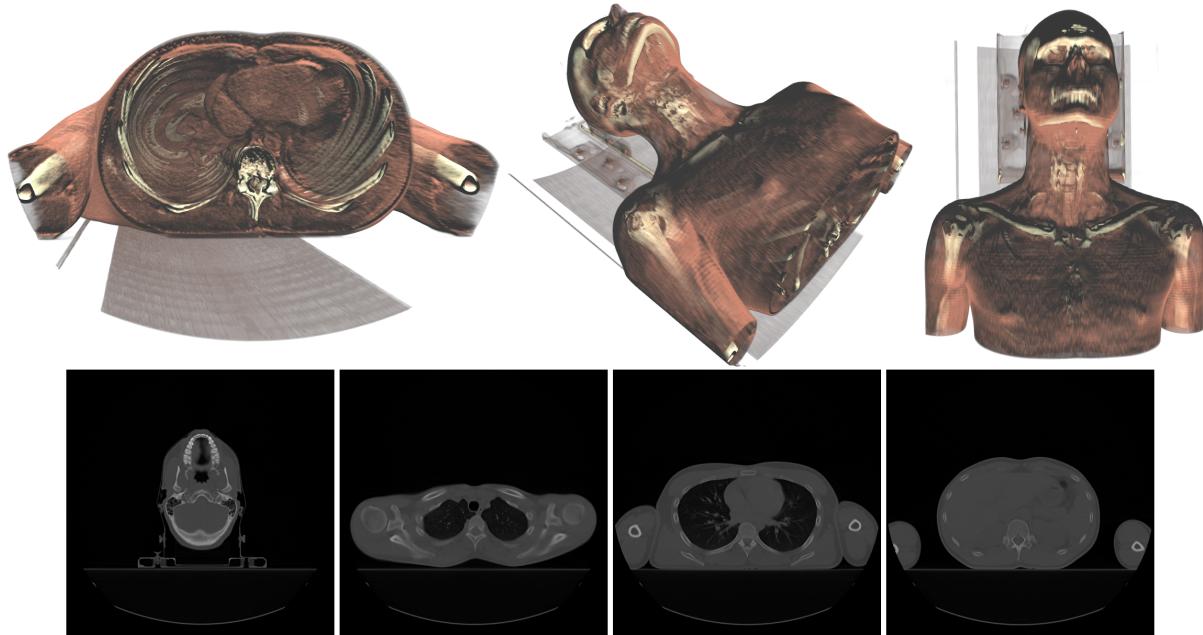


Bild 2.3: CT-Datensatz aus einer realen Patientenbehandlung. Oben linke Seite:  $512 \times 512$  Bildpunkte entlang der xy-Achse. Oben rechte Seite: 117 Schichten entlang der z-Achse von Kopf zu Fuß. Unten: CT-Ausgangsmaterial von 4 der 117 Schichten. Volumendatensatz *Thomas*.

**Größe der Volumendaten** 40 Jahre nach der Einführung von CT-Scannern im Jahre 1972 ist deren Einsatz im medizinischen Bereich für die Diagnostik und für präoperative Planungen heute eine Selbstverständlichkeit. Die ersten kommerziellen CT-Scanner boten eine Auflösung von  $64 \times 64 \times 64$  Voxel [Sme09]. Derzeit im Einsatz befindliche Geräte erzeugen Einzelbilder mit einer Bildauflösung von  $512 \times 512$  und einer Genauigkeit von 12 bit. Entlang der räumlichen z-Achse werden bis zu 50 Schichten mit jeweils einer Dicke von 1 mm bis 5 mm und einem Abstand von 1 mm bis 5 mm aufgezeichnet [Fan08]. Dieser Vorgang lässt sich mehrmals wiederholen. Abbildung 2.3 zeigt einen  $512 \times 512 \times 117$  großen CT-Datensatz aus einer realen Patientenbehandlung. Um die Strahlenbelastung für den Patienten so gering wie möglich zu halten, werden entlang der z-Achse, die in dieser Abbildung von Kopf zu Fuß verläuft, nur so viele Schichten aufgenommen wie benötigt. Aktuelle High-End CT-Scanner können Bilder in einer Auflösung von  $1024 \times 1024$  erzeugen und brauchen für die Aufnahme einer einzelnen Schicht weniger als 0,3 s [onla]. *SOMATOM Sensation Open* von Siemens kann einen gesamten Körper in 20 s einscannen [Sie09]. Nach [Sme09] ist es zu erwarten, dass die Auflösung von CT-Scannern in den nächsten Jahren bis auf  $2048 \times 2048$  Bildpunkte ansteigen wird.

Die mitunter größten verfügbaren medizinischen Volumendaten sind die Mitte der 90'er Jahre aufgenommenen Datensätze des *Visible Human* Projektes [Ack98]. Diese sind 15 GiB bis 39 GiB groß. Eine noch umfangreichere, neu digitalisierte Version ist verfügbar. [Koh03]

berichtet von Multi-Terabyte Datensätzen in der dreidimensionalen Mikroskopie, um beispielsweise Organe auf zellulärer Ebene zu untersuchen.

Für visuelle Effekte in der Filmindustrie sind nach [Wre10, S. 12] dicht besetzte Datensätze in einer Auflösung von  $1000 \times 1000 \times 1000^2$  noch handhabbar. Bereits  $2000 \times 2000 \times 2000^3$  große Volumen stellen eine Herausforderung dar und benötigen hohe Speicherkapazitäten. Dünn besetzte Datensätze sind, unter der Zuhilfenahme geeigneter Datenstrukturen, bei Größen von über  $4000 \times 4000 \times 4000^4$  noch nutzbar.

**Animation** Vergleichbar zu polygonbasierenden Verfahren sind auch mit Volumenbasierenden Verfahren prinzipiell Animationen möglich [Had06, Kap. 13]. Die umfangreichen Datenmengen bringen jedoch einige Herausforderungen mit sich.

Eine Diskussion darüber, wie sich über die Zeit verändernde Volumendaten effektiv von handelsüblicher Hardware gerendert werden können, findet sich in [Lum01]. Dieser Animationsstyp ist vergleichbar mit einem abgespielten Film und wird in der Literatur als 4-D-Volumen-Rendering bezeichnet.

Bei polygonbasierenden Verfahren ist es für Animationen üblich, ein abstraktes Skelett mit einer überschaubaren Anzahl an Kontrollpunkten zu erstellen. Vertices hängen über Gewichte an einem oder mehreren Kontrollpunkten, das Zuweisen der Gewichte ist als *Skinning* bekannt. Animationen finden im Folgenden nur noch auf dem abstrakten Skelett statt, die Vertices werden automatisch angepasst und müssen somit nicht einzeln per Hand animiert werden. Eine Möglichkeit, Animationsabläufe zu erzeugen besteht darin, Bewegungen realer Personen über *Motion Capturing* aufzuzeichnen. Dieses Verfahren ist bereits mit handelsüblicher und kostengünstiger Hardware wie der Kinect von Microsoft möglich [Ale11]. Wo nötig, können einzelne Kontrollpunkte weiterhin per Hand animiert oder über Programmlogik angesteuert werden. Ebenfalls ist es üblich, Animationen über eine Physiksimulation zu erstellen. Die Animationen werden somit dynamisch zur Laufzeit generiert. [Gag01] stellt ein Verfahren vor, um aus Volumendaten automatisiert ein abstraktes Skelett erstellen zu lassen und beschreibt die notwendige Rekonstruktion der Voxel. Hierdurch kann ein Volumen-basierendes Modell über die bereits erwähnten gängigen Methoden animiert werden. Dies wurde von [Gag01] anhand des *Visible Human* Datensatzes [Ack98] demonstriert, jedoch nur mit einem auf  $290 \times 169 \times 940$  Voxel herunter skalierten Datensatz.

---

<sup>2</sup>  $1000 \times 1000 \times 1000$  entspricht 954 MiB bei einem Byte pro Voxel

<sup>3</sup>  $2000 \times 2000 \times 2000$  entspricht 7629 MiB bei einem Byte pro Voxel

<sup>4</sup>  $4000 \times 4000 \times 4000$  entspricht 61 035 MiB bei einem Byte pro Voxel beim Speichern in einem 3-D-Feld

## 2.2 Volumen-Rendering

Die Aufgabe von Volumen-Rendering besteht darin, Volumendatensätze zu visualisieren. Die dreidimensionalen Datensätze müssen hierzu üblicherweise auf eine zweidimensionale Bildebene projiziert werden. Die Bedeutung der einzelnen Datenwerte im Volumen ist abhängig von der Modalität. Direkt darstellbare Farbwerte sind beispielsweise im medizinischen Anwendungsbereich unüblich. Die Extraktion von für den Benutzer relevanten Daten ist ein wichtiger Bestandteil der Volumen-Visualisierung. Zum Verständnis räumlicher Zusammenhänge im Datensatz, wird der Blickwinkel häufig verändert. Dies benötigt eine interaktive Bildwiederholrate [Ray99]. Eine interaktive Manipulation der Visualisierung kann eine weitere zu erfüllende Anforderung sein.

Bei Volumen-Rendering handelt es sich um einen Überbegriff für eine Vielzahl an Algorithmen, die eine Visualisierung von Volumendaten zum Ziel haben.

**Indirect Volume Rendering (IVR) und Direct Volume Rendering (DVR)** Die Volumen-Rendering Ansätze lassen sich in IVR und DVR unterteilen.

Die Klasse IVR fasst Methoden zusammen, die Volumendaten für eine Weiterverarbeitung in eine andere Repräsentation überführen. So werden bei oberflächenextrahierenden Methoden anhand eines Schwellwertes aus den Volumendaten polygonbasierende 3-D-Modelle erzeugt. Diese Geometrien lassen sich im Anschluss mit einer traditionellen Grafiksschnittstelle wie OpenGL darstellen. Einer der bekanntesten Vertreter der oberflächenextrahierenden Methoden ist der *Marching Cubes*-Algorithmus [Lor87].

Das von [Dre88] und [Lev88] vorgestellte DVR arbeitet direkt auf den Volumendaten, ohne diese zur Weiterverarbeitung in eine andere Darstellung umzuwandeln. Vergleichbar zu IVR ist DVR als Überbegriff für eine Vielzahl von Volumen-Rendering Verfahren zu verstehen. Im weiteren soll nur noch DVR betrachtet werden.

**Objekt- und Bildbasierend** Die Klasse DVR lässt sich in Objekt- und bildbasierende Methoden untergliedern [Fan08].

Die objektbasierenden Verfahren projizieren einzelne Voxel auf die Bildebene. Bei Splatting [Wes90] hinterlässt jeder Voxel einen Fußabdruck auf dem Bildschirm. Weitere Verfahren verwenden 2-D-Texturmapping [RS00], 3-D-Textur Slicing [Wil94, Cab94] oder Shear-Warp [Lac94].

Bildbasierende Methoden nehmen den zu berechnenden Bildpunkt auf dem Bildschirm als Ausgangsbasis. Das bekannteste Verfahren hierfür ist Raycasting, das wie auf Abbildung 2.4 dargestellt, einen Strahl vom Auge durch den Pixel wirft um die Farbe dieses Bildpunk-

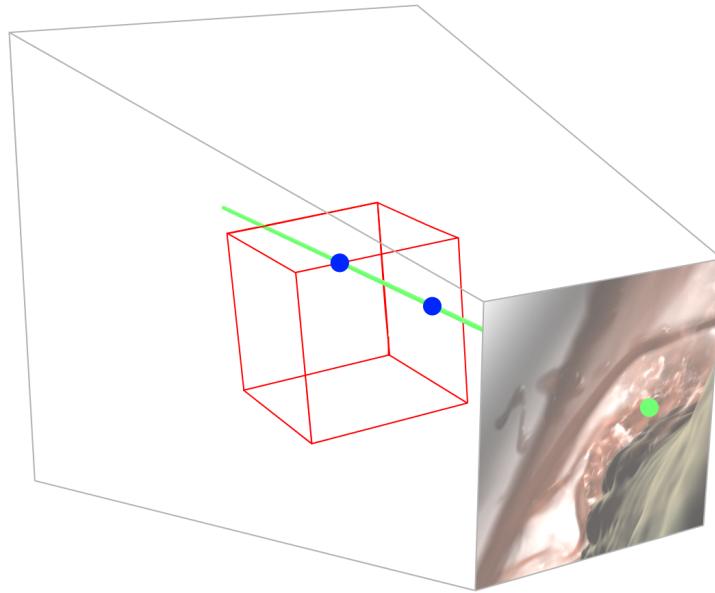


Bild 2.4: Volumen-Raycasting.

tes zu ermitteln [Sab88]. Raycasting sollte nicht mit dem deutlich aufwändigeren *Raytracing* [Pur02] verwechselt werden. Unter der Zuhilfenahme von Testpersonen kommt [Sme09] zu dem Schluss, dass die Ergebnisse von Raycasting die größte Akzeptanz bei den Benutzern findet.

**Raycasting auf der GPU** Bei einer sequentiellen CPU-basierenden Realisierung von Volumen Raycasting sind bei höheren Bildschirmauflösungen keine interaktiven Visualisierungen zu erwarten. Nicht jeder Benutzer ist Willens, für Volumen-Rendering Spezialhardware [Pfi99, Mei02] einzusetzen.

GPU's mit unterschiedlicher Leistungsfähigkeit sind hingegen in fast jedem handelsüblichen PC verbaut. [Krü03] und [Roe03] stellen eine GPU-basierte Volume Raycasting Lösung vor.

Eine detaillierte Übersicht über Volumen-Rendering ist in [Mei00] zu finden. Für weiterführende Informationen über das umfangreiche Themengebiet sei auf [Had06] und [Had09] verwiesen.



# Kapitel 3

## Integration in eine 3-D-Engine

Wie in Kapitel 2 ersichtlich wurde, ist das Feld des Volumen-Rendering umfangreich. Aufgrund dessen liegt es nahe, für das Gesamtsystem ein bereits fertiges Framework zu verwenden. [Kra06] verwendet *Studierstube* [Sch02] als Ausgangsbasis. Der im Rahmen dieser Arbeit angefertigte Volumen-Renderer hingegenbettet sich in die Open-Source 3-D-Engine *PixelLight* [Bus] ein. Über einen grafischen Editor besteht die Möglichkeit, beispielsweise Schnittebenen direkt in einem 3-D-Bildausschnitt frei im Raum auszurichten. Unterstützung für spezielle Eingabegeräte ist vorhanden, so kann für die räumliche Navigation unter anderem eine 3-D-Maus verwendet werden. Die Wahl viel auf diese Engine, da der Autor der vorliegenden Masterarbeit an der Technologie mitgewirkt hat. Dies ersparte eine Einarbeitungszeit.

Durch Nutzung der vorhandenen Infrastruktur konnte der Schwerpunkt auf Volumen-Rendering gesetzt werden, das prinzipiell Plattformunabhängig ist. Es wird lediglich auf durch die 3-D-Engine bereitgestellte Funktionalität zurückgegriffen. Aufgrund der Pluginarchitektur dieser Technologie konnte der Volumen-Renderer komplett als loses Plugin realisiert werden, das sich nahtlos in das Gesamtsystem einfügt. [Huc10] beschreibt eine vergleichbare Pluginarchitektur für die gleiche Aufgabenstellung. Die im Rahmen dieser Arbeit entstandene Realisierung wurde gemäß dem Model-View-Control (MVC)-Architekturmuster in drei Plugins zerlegt:

1. Das *Datenmodell* wurde in einem *PLVolume* genannten Plugin implementiert und fügt neue Knotentypen für den Szenengraphen hinzu. Ebenfalls befindet sich hier die Verwaltung der Volumendaten.
2. Die *Präsentation* übernimmt der im *PLVolumeRenderer*-Plugin realisierte Volumen-Renderer. Dies stellt den Kern der Masterarbeit da.
3. Für die optionale *Programmsteuerung* wurde das *PLVolumeGUI*-Plugin umgesetzt, das sich nahtlos in den *Qt*-basierenden [onlc] PixelLight-Betrachter einfügt. Unter anderem

wird es hierdurch ermöglicht, Transferfunktionen visuell zu bearbeiten. Um die Entwicklung einer speziellen Testanwendung in C++ zu vermeiden, wurde einfache Anwendungslogik über *Lua*-Skripte [onlb] umgesetzt.

Im Folgenden wird Grundwissen über die Programmierung von Echtzeitgrafik vorausgesetzt. [AM08] liefert einen umfassenden Einblick in die Grafikprogrammierung, während [Fer03] zum Verständnis der Shaderprogrammierung zu empfehlen ist.

### 3.1 Datensatz im Speicher

Der Speicherbedarf von Volumendaten wächst aufgrund der dritten Dimension mit der Auflösung des Datensatzes deutlich schneller an, als dies bei einer regulären 2-D-Textur der Fall ist. Neben dem Visualisieren der Volumendaten ist das Speichermanagement ein nicht zu vernachlässigendes Thema. Aufgrund des Anspruches der Plattformunabhängigkeit sollen mobile Endgeräte an dieser Stelle nicht unerwähnt bleiben.

**Textur** Zum Speichern der Volumendaten auf der Grafikkarte bietet sich auf einem Desktop-PC der Einsatz von 3-D-Texturen an. Diese sind seit der 1998 veröffentlichten OpenGL-Version 1.2 offiziell Bestandteil des Standards. Die Grafikkarte übernimmt beim Zugriff auf die 3-D-Texturdaten automatisch eine trilineare Filterung, sofern dies nicht deaktiviert wurde. Aus einer Shadersprache heraus ist ein problemloser und freier Texturzugriff möglich. Auf mobilen Endgeräten ist die notwendige OpenGL ES 2.0 Erweiterung *GL\_OES\_texture\_3D* nicht sehr verbreitet. Auf dem *HP TouchPad* ist diese Erweiterung jedoch aufgelistet.

Die Erweiterung *GL\_EXT\_texture\_array* ist im OpenGL Standard 3.0 enthalten, der im Jahre 2008 veröffentlicht wurde. Hierbei handelt es sich um eine Art von 3-D-Textur, zwischen den Schichten entlang der z-Achse findet jedoch keine automatische Filterung statt. Während die Grafikkarte für jede Schicht bilineare Filterung anbietet, muss eine trilineare Filterung, sofern benötigt, in einem Fragmentshader realisiert werden. Auf modernen Grafikkarten können 2-D-Texturfelder als gegeben angesehen werden. Da jedoch ebenfalls 3-D-Texturen vorhanden sind, bieten die 2-D-Texturfelder zumindest für die Speicherung von Volumendaten, keine Vorteile. Auf mobilen Endgeräten hingegen macht es Sinn, diese Texturart zu berücksichtigen. Während das *HP TouchPad* Unterstützung für 3-D-Texturen besitzt, sind 2-D-Texturfelder nicht verfügbar. Auf dem *Tegra 2* Smartphone *LG P990* hingegen verhält es sich entgegen gesetzt. Eine Unterstützung von 3-D-Texturen ist nicht vorhanden, jedoch konnten in einem experimentellen Volumen-Renderer 2-D-Texturfelder erfolgreich eingesetzt werden.

2-D-Texturen sind seit der ersten 1992 veröffentlichten OpenGL-Version verfügbar. Auf aktuellen mobilen Endgeräten gehören diese ebenfalls zum Standardrepertoire und können aufgrund dessen immer als vorhanden vorausgesetzt werden. Besteht der Anspruch auf eine robuste Volumen-Renderer Realisierung sowie eine optimale plattformübergreifende Kompatibilität, ist die Unterstützung für 2-D-Texturen in einem Volumen-Renderern weiterhin als sinnvoll zu erachten. Die *flachen* 3-D-Texturen werden in einem 2-D-Texturatlas abgelegt, eine gegebene 3-D-Texelposition wird auf eine 2-D-Position übertragen. Für diese Umrechnung verwendet [Har03] im Fragmentshader die z-Positionskomponente der gegebenen Texelposition als Index in eine 1-D-Textur. Die 1-D-Textur wiederum enthält den xy-Offset für die angefragte Schicht, dieser wird anschließend auf die xy-Texelposition addiert. Vor vier Jahren nutzte [Ras08] 2-D-Texturen für die Speicherung von Volumendaten mit dem Argument einer erhöhten Leistungsfähigkeit im Gegensatz zu 3-D-Texturen. Die maximale Größe von Texturen ist begrenzt. Im Gegensatz zu einer 3-D-Textur oder einem 2-D-Texturfeld, ist aufgrund dessen die Größe der nutzbaren Datensätze bei 2-D-Texturen stärker eingeschränkt. Die Implementierung ist durch die Erstellung eines Texturatlases mit anschließender Koordinatenumrechnung umständlich. Beim Speichern von Offsets in einer 1-D-Textur, wird pro Volumendatenzugriff ein weiterer Texturzugriff benötigt. Dies bedeutet für die GPU ein zusätzliches Speichermanagement, das sich potentiell negativ auf die zu erwartende Leistungsfähigkeit auswirkt.

In der Echtzeitcomputergrafik ist es üblich, für jede Textur ebenfalls Mipmaps zu speichern. Dies sind schrittweise verkleinerte Versionen der ursprünglichen Textur. Bei 2-D-Texturierung lässt sich hierdurch die Bildqualität bei der Texturfilterung erhöhen, sowie die Bildwiederholrate. Diese Aussage ist jedoch nicht direkt auf Volumen-Rendering über Raycasting übertragbar, da es sich hierbei um ein bildbasiertes Verfahren handelt. Die Grafikkarte kann nicht automatisch die zu verwendende Mipmap bestimmen. In einem Fragmentshader existiert freier Zugriff auf bestimmte Mipmaps einer Textur, Algorithmen können diese zusätzlichen Daten für Optimierungen nutzen. Anhand Abbildung 3.1 ist zu erkennen, dass für die Visualisierung der Volumendaten gezielt eine bestimmte Texturauflösung nutzbar ist. Dies kann beispielsweise dazu verwendet werden, während einer Interaktion dynamisch die Texturauflösung zu reduzieren, um eine höhere Bildwiederholrate zu gewährleisten. Der zusätzliche Speicherbedarf für Mipmaps hält sich in Grenzen, es wird lediglich 1/7 mehr an Grafikkartenspeicher benötigt [Had06, S. 190]. Eine 217 MiB große 3-D-Textur benötigt mit Mipmaps 248 MiB.

**Grafikkartenspeicher** Sollte der Grafikkartenspeicher nicht ausreichen um die Volumendaten zu speichern, ist ein Volumen-Rendering dieses Datensatzes nicht ohne weiteres möglich. Der einfachste Lösungsweg besteht darin, den Umfang der Volumendaten durch Skalierung zu reduzieren. Für qualitativ hochwertige Visualisierungen ist dies jedoch keine Option. Verfahren

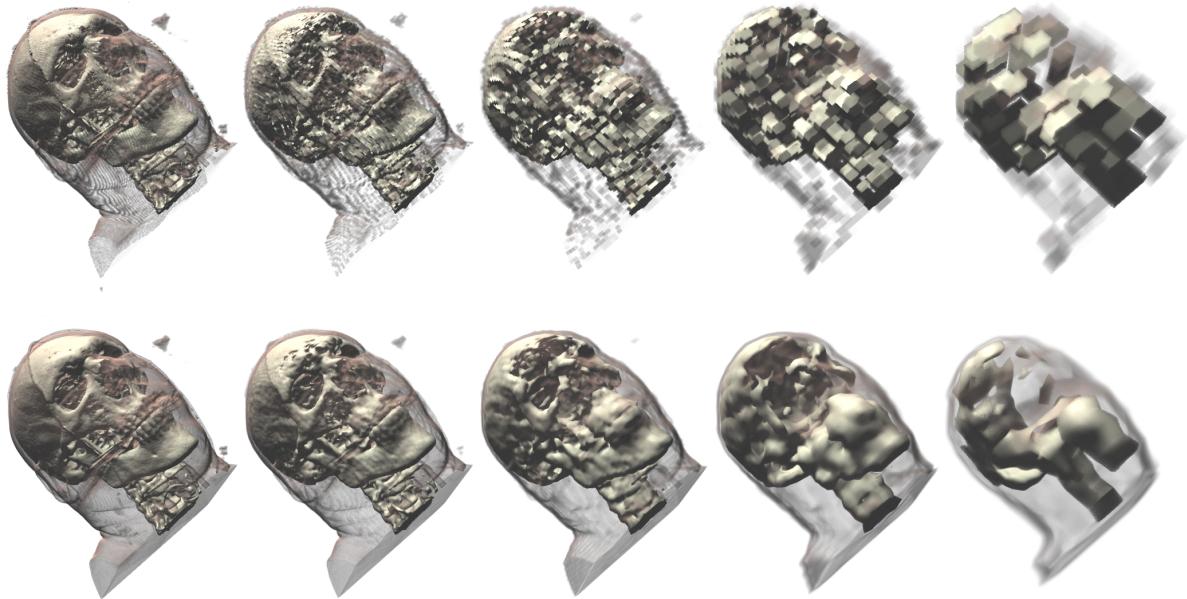


Bild 3.1: Mipmapping Stufe von links nach rechts: 0, 1, 2, 3 und 4. Unten mit trilinearer Filterung. Volumendatensatz *Head (Visible Male)*.

wie *Bricking* [Had06, S. 446] können Abhilfe schaffen. [Sch05] konnte einen 580 MiB großen Datensatz mit nur 190 MiB speichern. Hierzu wurde das Volumen in gleichgroße Blöcke unterteilt, gespeichert wurden nur nicht leere Blöcke. Ob ein Block leer ist oder nicht, hängt dabei von der jeweiligen Transferfunktion ab. Zum Rekonstruieren werden in einer niedriger auflösenden 3-D-Textur Indexkoordinaten gespeichert. Um korrekte trilineare Filterung zu ermöglichen, muss jeder Block ebenfalls einen Rand besitzen.

Für interaktives Volumen-Rendering ist die Block-basierende Lösung nur geeignet, solange sich zur Laufzeit weder die Volumendaten noch die Transferfunktion ändern. Für qualitativ hochwertigere Rekonstruktionsfilter muss pro Block ein größerer Rand gespeichert werden. Dies treibt den für Bricking zusätzlich benötigten Speicher weiter in die Höhe. Des Weiteren bietet diese Lösung nur einen Mehrwert, wenn tatsächlich ausreichend viele Blöcke als leer eingestuft werden können.

Im Jahre 2005 war der verfügbare Grafikkartenspeicher eine der größten Einschränkungen von GPU-basierten Volumen-Raycasting [Sch05]. 256 MiB waren Standard für neue Grafikkarten, dies war gerade ausreichend um einen  $512 \times 512 \times 512$  Voxel großen Datensatz in 16 bit zu speichern. Aktuell sind Notebooks<sup>1</sup> mit 2 GiB Grafikkartenspeicher erhältlich. Für den professionellen Einsatz existieren Workstation-Grafikkarten, wie beispielsweise *Quadro 6000*<sup>2</sup>,

<sup>1</sup>Beispiel: *Samsung Serie 7 Gamer 700G7A*, erschienen Ende 2011

<sup>2</sup><http://www.nvidia.com/object/product-quadro-6000-us.html>

mit 6 GiB Grafikkartenspeicher. Ein  $1024 \times 1024 \times 1024$  Voxel großer Datensatz in 16 bit ist heute ohne zusätzlichen Aufwand nutzbar. Größere Volumen, wie beispielsweise die Mitte der 90’er Jahre aufgenommenen 15 GiB bis 39 GiB großen Datensätze des *Visible Human* Projektes [Ack98], stellen weiterhin eine Herausforderung beim Volumen-Rendering da [Rod99, Gut02]. Dies gilt insbesondere für GPU-basierende Verfahren.

**Texturkompression** Der Speicherbedarf der Volumendaten lässt sich über Kompression reduzieren. Aufgrund des verringerten Speichertransfers wird in der Regel die Bildwiederholrate verbessert. Grafikkarten unterstützen verlustbehaftete Kompression, im medizinischen Einsatzbereich ist dies kritisch zu betrachten da eine Bildverfälschung eintritt. Bis vor wenigen Jahren existierten lediglich GPU-Kompressionsformate mit 8 bit pro Pixelkomponente. 16 bit Datensätze müssen folglich auf 8 bit heruntergerechnet werden um weitere Optimierungsmöglichkeiten zu ermöglichen. Für Volumendaten mit 8 bit pro Voxel bietet sich das qualitativ hochwertige Kompressionsformat *LATC1*<sup>3</sup> an. Im Gegensatz zu älteren Kompressionsformaten sind bei einer klassischen Texturierung mit dem bloßen Auge üblicherweise keine Kompressionsartefakte erkennbar.

Ein  $512 \times 512 \times 1743$  Voxel großes Volumen mit 16 bit benötigt 871 MiB Speicher. Für eine Grafikkarte mit 512 MiB Speicher ist dieses Volumen nicht mehr ohne zusätzliche Algorithmen nutzbar. Bei einer Reduktion der Daten auf 8 bit pro Voxel, werden 435 MiB im Grafikkartenspeicher abgelegt. Durch den Einsatz von LATC1-Kompression reduziert sich der Speicherbedarf auf 217 MiB, 1/4 der ursprünglichen 871 MiB. Werden die Volumendaten auf der Festplatte direkt in diesem Format abgelegt, so reduziert sich ebenfalls die Ladezeit des Volumen.

Mit DirectX 11 wurden zwei weitere, als *BC6* und *BC7* bekannte Texturkompressionsformate eingeführt. Diese bieten qualitativ hochwertige Kompression für Fließkommazahlenbasierende Formate an. In OpenGL sind diese neuen Formate ebenfalls nutzbar. Aufgrund der Komplexität der neuen Formate und den hohen Hardwareanforderungen soll nicht weiter auf diese neuen Formate eingegangen werden.

## 3.2 Szene

Die verwendete 3-D-Engine organisiert Szenen über einen hierarchischen Szenengraphen. Grundlegende Elemente sind Knoten und Container. Ein Knoten besitzt neben den räumlichen Informationen Position, Rotation und Skalierung ebenfalls weiterführende abstrakte Informationen wie die räumliche Ausdehnung. Ob es sich bei einem Knoten um ein 3-D-Modell,

---

<sup>3</sup>Früher bekannt als *3DC+* und *ATIIN*, in DirectX 10 läuft diese Texturkompression unter dem Namen *BC4*

Lichtquelle oder eine Kamera handelt, ist für viele Algorithmen nicht von weiterem Interesse. Bei einem Container handelt es sich um einen Knoten, der wiederum mehrere Knoten beinhalten kann welche relativ zum Container ausgerichtet sind. Über Container lassen sich beliebig komplexe hierarchische Strukturen bilden. Aufgrund dessen existiert prinzipbedingt kein globales Weltkoordinatensystem. Koordinaten sind stets relativ zu einem definierten Bezugssystem. Dies stellt jedoch keine Einschränkung dar, sondern ermöglicht beliebig große Szenen, die sich aus kleinen Teilszenen zusammensetzen lassen. Ein dynamisches konstruieren von Szenen ist ebenfalls möglich. Um beispielsweise eine Abstandsmessung zwischen zwei Punkten im Raum vorzunehmen, lassen sich Koordinaten in ein gemeinsames Koordinatensystem umrechnen.

Der vorhandene Szenengraph besitzt ein Modifizierer-Konzept. Zur Steuerung einer Kamera bietet es sich an, zwei Modifizierer an den Kameraknoten anzuhängen. Der erste kümmert sich um die Steuerung der Blickrichtung, während der zweite für die Bewegung des Kameraknoten verantwortlich ist. Die einzelnen Modifizierer sind wiederum universell gehalten und lassen sich folglich an beliebige Knoten anhängen. Eine um ein Objekt rotierende Kamera ist hierüber ebenfalls realisierbar, entsprechende Modifizierer stehen zur Verfügung.

Für die Volumen-Renderer Integration wurde dieses vorhandene System um lose, zur Laufzeit eingeladene Plugins erweitert. Neue Knotentypen wurden vom Basisknoten des Szenengraphen abgeleitet. Ein Volumenknoten besitzt weiterführende Informationen, wie beispielsweise die zu verwendenden Volumendaten, sowie Konfigurationsmöglichkeiten um die Visualisierung der Daten zu beeinflussen. Die Eigenschaften der Szenenknoten sind, dank des Runtime Type Information (RTTI)-Systems der Engine, direkt über Skriptsprachen oder einen grafischen Editor ansteuerbar. Auf der anderen Seite wurden weitere Knoten für Schnittelemente realisiert. Eine Schnittebene ist somit lediglich ein im Raum frei platzierbarer Szenenknoten. Zur Ausrichtung der Schnittebene lassen sich vorhandene grafische Modifizierer anhängen, um direkt visuell in der 3-D-Ansicht zu arbeiten. Mehrere Instanzen der neuen Knotentypen in verschiedenen räumlichen oder hierarchischen Anordnungen sind möglich.

Der Szenengraph entspricht im MVC-Architekturmuster dem Datenmodell. Sich räumlich überschneidende Volumen oder Schnittelemente sind auf dieser Architekturebene nicht weiter von Interesse.

Um eine mehrmalige Wiederverwendung in der Szene zu ermöglichen, sind die Volumendaten nicht fest mit den realisierten Szenenknoten verbunden. Die Knoten sind lediglich eine Verbindungsschnittstelle zwischen Szenengraph und den eigentlichen Daten. Einmal eingeladene Volumendaten lassen sich somit mehrmals gleichzeitig in der Szene platzieren, während beispielsweise Visualisierungseigenschaften, wie in 3.2 abgebildet, variiert werden können. Dies funktioniert ebenfalls mit verschiedenen Datensätzen gleichzeitig.

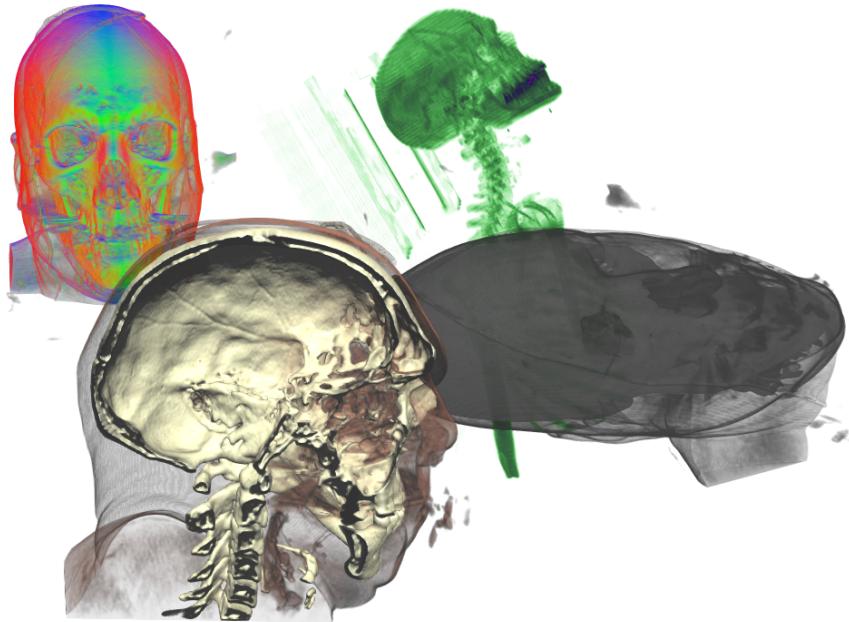


Bild 3.2: Mehrere Volumen gleichzeitig in einer Szene. Volumendatensatz *Head (Visible Male)* und Volumendatensatz *Thomas* rechts hinten.

### 3.3 Szenen-Renderer

Der Szenengraph selbst ist nicht zuständig für das Zeichnen eines gewählten Bildausschnittes. Gemäß dem MVC-Architekturmuster übernimmt dies eine eigenständige Präsentationskomponente: Der Szenen-Renderer. Dieser wiederum lässt sich in die zwei zentralen Bestandteile Sichtbarkeitsbestimmung und Darstellen der als sichtbar eingestuften Szenenknoten zerlegen.

Das Sichtbarkeitssystem der verwendeten 3-D-Engine erzeugt aus dem Szenengraphen einen neuen Graphen, der nur sichtbare Szenenknoten beinhaltet. Die als sichtbar eingestuften Szenenknoten werden automatisch anhand der Entfernung zur Kamera sortiert. Hierdurch ist es in den nachfolgenden Renderschritten möglich, diesen Graphen für nicht transparente Objekte von vorne nach hinten zu durchlaufen. Durch weniger Überzeichnen wird Füllrate eingespart. Für transparente Objekte kann der Graph von hinten nach vorne zur Bildebene hin durchlaufen werden. Hierdurch wird beim Übereinanderblenden eine korrekte Farbmischung ermöglicht. Dies hat zur Folge, dass auch mehrere Volumen in einer Szene korrekt gezeichnet werden können, sofern die Volumen räumlich komplett voneinander getrennt sind.

Räumlich ineinandergreifende transparente Objekte sind in der Echtzeitgrafik ein Problem, da kein einfacher Tiefenpuffer-Test verwendet werden kann um die Verdeckung von Fragmenten zu bestimmen. Polygonbasierende Geometrien setzen sich aus zahlreichen Dreiecken zusammen. Ein Sortieren einzelner Dreiecke ist aufgrund des verfügbaren Zeitrahmens

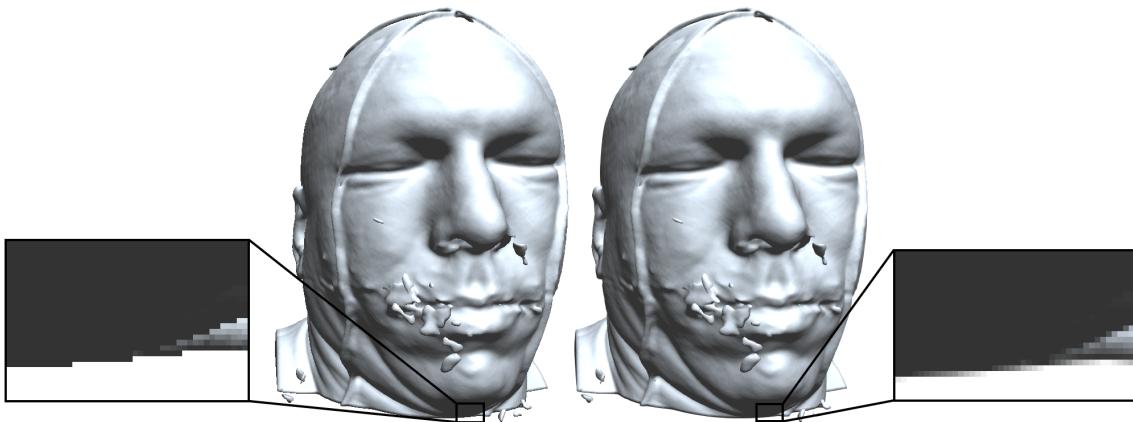


Bild 3.3: Links ohne, rechts mit Kantenglättung. Volumendatensatz *Head (Visible Male)*.

pro Bild nicht möglich. DVR verwendet keine Dreiecke die sortiert werden könnten. Eine Lösungsmöglichkeit für überlappende Volumen besteht darin, die Volumen für den Render schritt als ein gemeinsames Volumen zu betrachten und bei der Strahlverfolgung gleichzeitig durch alle beteiligten Volumen zu traversieren [Wre10, S. 189, S. 253]. Aufgrund der Komplexität der Thematik soll diese Lösung im Rahmen dieser Arbeit nicht weiterverfolgt werden.

**Szenen Renderschritt** Rendern ist in der verwendeten 3-D-Engine in einzelnen Teilschritten realisiert. So existiert ein Renderschritt am Anfang der Renderpipeline, dessen Aufgabe darin besteht, den Bildschirminhalt zu löschen. In diesem Renderschritt kann ebenfalls das Zeichnen in eine Textur, anstatt direkt in den Bildpuffer, konfiguriert werden. Dies ermöglicht unter anderem eine dynamische Veränderung der Bildpufferauflösung [Bin11]. Weitere Renderschritte kümmern sich anschließend um das Befüllen des Bildpuffers.

Bevor der Bildpufferinhalt dem Benutzer präsentiert wird, sind bildverbessernde Nachbearbeitunsschritte möglich. Volumen-Rendering über Raycasting ist ein bildbasiertes Verfahren, dies hat zur Folge, dass die von der GPU unterstützte polygonbasierende Kantenglättung Multisample Anti-Aliasing (MSAA) nicht verwendet werden kann. Alternativ lässt sich, wie in Abbildung 3.3 zu sehen, beispielsweise Fast Approximate Anti-Aliasing (FXAA) [Lot09] von NVIDIA Corporation (NVIDIA) für die Kantenglättung einsetzen. Hierbei handelt es sich um einen einfachen Nachbearbeitungsschritt, der den aktuellen Bildpufferinhalt als Eingabe erhält und ausschließlich anhand der Farbwerte eine Kantenglättung vornimmt. Die Bildwiederholrate wird hierdurch nicht deutlich verschlechtert, das resultierende Bild wirkt allerdings ruhiger. Als unerwünschter Seiteneffekt kann jedoch eine leichte Unschärfe im Bild hinzukommen. Bevor das aktuelle Bild präsentiert wird, lässt sich wie in [Ngu07, Kap. 24] und [Bre09] beschrieben eine Gammakorrektur anwenden. Unterstützung für High Dynamic Range Rendering (HDR)

ist vorhanden, für die Umrechnung in den von handelsüblichen Bildschirmen darstellbaren Farbraum ist eine Reinhard Tone-Mapping Operator [Rei02] Implementierung nutzbar.

Aufgrund der bereits verfügbaren Infrastruktur und bildverbessernder Funktionalitäten kann im folgenden der Schwerpunkt auf das eigentliche Volumen-Rendering gelegt werden. Für dessen Realisierung wurde ein weiterer Teilschritt in die Renderpipeline eingefügt. Dieser Teilschritt ist in der Lage, die Volumenknoten in einer Szene zu verarbeiten.

**Shader** Herzstück des Volumen-Renderers ist ein Fragmentschader, der von der Grafikkarte für jeden Bildpunkt abgearbeitet wird.

Neben Shadersprachen lässt sich eine moderne GPU ebenfalls mit einem General-Purpose Computing on Graphics Processing Units (GPGPU)-Ansatz über Compute Unified Device Architecture (CUDA) oder Open Computing Language (OpenCL) programmieren. Die Ansätze sind hierbei nicht allzu unterschiedlich. So lässt sich ein Fragmentschader im Allgemeinen einfach zu einem OpenCL-Kernel portieren. Für den Einsatz von GPGPU-Hochsprachen wird eine zeitgemäße Grafikkarte benötigt, andernfalls ist die verfügbare Funktionalität und hierdurch der allgemeine GPGPU-Mehrwert eingeschränkt. Shader hingegen sind mittlerweile Standard, selbst OpenGL 3.3 ist auf einer breiteren Hardwarepalette verfügbar. Dank des weit verbreiteten OpenGL ES 2.0 werden Shader ebenfalls für mobile Endgeräte verwendet. Volumen-Rendering über Web Graphics Library (WebGL) wurde bereits erfolgreich von [Con11] umgesetzt. Bei WebGL handelt es sich um eine JavaScript-Schnittstelle für OpenGL ES 2.0.

In einem klassischen Volumen-Renderer über Raycasting werden die Strahlen parallel und unabhängig voneinander verarbeitet. Es wird weder eine Synchronisierung zwischen den einzelnen Strahlen noch der Einsatz komplexer Datenstrukturen benötigt. Unter diesen Umständen bietet eine GPGPU-Hochsprache keine deutlichen Vorteile. Gleichzeitig schränkt OpenCL und vor allem das nur auf NVIDIA-Hardware lauffähige CUDA die Anzahl der nutzbaren Plattformen ein. Eine Realisierung eines High-End Volumen-Renderers kann hingegen die Verwendung von GPGPU-Hochsprachen notwendig machen. Hierdurch lassen sich Datenstrukturen realisieren, die nicht durch Texturvorgaben eingeschränkt werden. Diese Klasse von Volumen-Renderer Realisierungen benötigt leistungsfähige und aktuelle Hardware, aufgrund dessen ist die Lauffähigkeit auf einer möglichst hohen Anzahl an Hardwarekonfigurationen dort kein ausschlaggebender Faktor.

Das in dieser Arbeit vorgestellte Szenen-Renderer Plugin für Volumen-Rendering verwendet Shader um eine breite Palette an Hardwarekonfigurationen abzudecken. Das erstellte System ist prinzipiell Shadersprachen unabhängig. Konkrete Shaderbausteine für OpenGL Shading Language (GLSL) 3.3 und C for Graphics (Cg) wurden realisiert, ein Hinzufügen von Unterstützung für weitere Shadersprachen wie GLSL für mobile Endgeräte ist möglich. Dies

ist ebenfalls als Sicherungsmaßnahme zu verstehen. Es existieren zahlreiche Hardware- und Treibervariationen, daher muss damit gerechnet werden, dass ein Grafikkartentreiber einen GLSL-Shader nicht verarbeiten kann. Bereits geringfügige Abweichungen, oder unterschiedliche Interpretationen der GLSL-Spezifikation, führen zu Problemen. Im Notfall kann auf einen Cg-Shader ausgewichen werden. Während das von NVIDIA stammende Cg auf NVIDIA-Grafikkarten in der Regel selbst mit modernen Shaderfunktionalitäten problemlos läuft, ist die Unterstützung moderner Shader auf nicht NVIDIA-Hardware nicht zufriedenstellend. Daher muss ein skalierbares Volumen-Rendering System sowohl GLSL als auch Cg unterstützen. Die verwendete 3-D-Engine bietet hierfür abstrakte Shaderschnittstellen an. Während die eigentlichen Shader-Quellcodes für jede Shadersprache vorliegen müssen, ist deren Ansteuerung im C++ Quellcode unabhängig von der gewählten Shadersprache.

**Shadergenerierung** In den Anfängen der Shaderprogrammierung wurden Shader in einer Assemblersprache erstellt. Hauptherausforderung bestand zu diesem Zeitpunkt darin, Problemstellungen mit den wenigen verfügenden Funktionen und der stark eingeschränkten Anzahl an maximal nutzbaren Instruktionen zu bewältigen. Während auf mobilen Endgeräten die Instruktionslimitierung weiterhin eine zu bewältigende Hürde darstellt, sind auf dem Desktop-PC ab Shadermodell 3 65,536 Instruktionen innerhalb eines Shaders nutzbar [Kil05]. Ab Shadermodell 4 existiert kein Instruktionslimit mehr [One07]. Hauptherausforderung stellt heutzutage die Erstellung und Verwaltung der umfangreichen Shader und Shadervariationen da.

Im einfachsten Fall wird ein Shader in einer Hochsprache von Hand geschrieben. Ein Fragmentshader für einen grundlegenden Volumen-Renderer für die Strahlverfolgung über Maximum Intensity Projection (MIP) ist mit wenigen Codezeilen realisierbar. Eine Klassifikation zur Umrechnung von Skalarwerte in Farbwerte lässt sich durch eine zusätzliche Codezeile hinzufügen. Weitere Funktionalitäten lassen sich ebenfalls ohne Schwierigkeiten in einen bereits bestehenden Shader einfügen. Für das Rendern von Isolänen kann ein alternativer Shader entwickelt werden. Denkbar ist ebenfalls eine Fallunterscheidung innerhalb eines Shaders. Wird diese Vorgehensweise der Shadererstellung weiterverfolgt, so ergibt sich ein umfangreicher Shader. Ein Rendern über einfaches MIP wird ineffektiv, viele nicht verwendete Codefragmente müssen weiterhin von der Grafikkarte verarbeitet werden.

Das Problem der effektiven Shadererzeugung lässt sich über sogenannte *Übershader* lösen [Mit07]. Codefragmente werden durch `#ifdef` Precompiler-Anweisungen umschlossen. Je nach eingefügten Precompiler-Definitionen entsteht ein Shader mit mehr oder weniger Instruktionen. Abhängig vom Übershader-Umfang ergibt sich eine überschaubare Anzahl an Shaderpermutationen bis hin zu einer Anzahl, die ein einmaliges Übersetzen aller Kombinationsmöglichkeiten vor dem Programmstart nicht mehr sinnvoll möglich macht. Umfangreiche Übershader benötigen

gen ein System für das Konfigurationsmanagement. Gemäß den aktuellen Anforderungen, werden zur Laufzeit dynamisch Shader erzeugt, die nur den benötigten Funktionsumfang besitzen. Das Übershader-Konzept ermöglicht somit eine hohe Anzahl an effizienten Shadervariationen, der eigentliche Übershader muss hingegen lediglich einmalig geschrieben und gepflegt werden.

Wegen den unterschiedlichen Anforderungen an die resultierende Volumen-Visualisierung benötigt ein skalierbarer Volumen-Renderer eine hochgradig konfigurierbare Shaderarchitektur. Über den Übershader-Ansatz resultiert dies in einem großen Shader. Ein nachträgliches Einfügen weiterer Funktionalitäten oder eine Fehlerkorrektur im Shader ist somit nur unter enormen Zeitaufwand möglich. Bei nachträglichen Änderungen muss ebenfalls mit unerwünschten Seiteneffekten auf andere Bestandteile des umfangreichen Shaders gerechnet werden.

Aufgrund dessen wurde im Rahmen dieser Arbeit ein anderer, zu [Roe08] vergleichbarer Ansatz gewählt. Volumen-Rendering über Raycasting wurde hierzu im Detail analysiert, in sich geschlossene Teilaufgaben wurden identifiziert. Das Ergebnis ist eine überschaubare Anzahl an Shaderfunktionen mit wohldefiniertem Aufgabenbereich und fest vorgegebener Funktionsignatur. Ein Beispiel für eine Shaderfunktion stellt die Strahlverfolgung da, für die in der Literatur neben MIP zahlreiche weitere Algorithmen existieren. Für eine Beleuchtung hingegen werden Normalenvektoren benötigt, die beim Volumen-Rendering in der Regel über eine Gradientenschätzung ermittelt werden. Für die Gradientenschätzung existiert ebenfalls umfangreiche Literatur, während für den Shader-Quellcode der Beleuchtung Details der Gradientenschätzung nicht relevant sind.

Im Abschnitt 3.4 wird auf die Strahlgenerierung eingegangen, von hier aus werden die ersten Shaderfunktionen aufgerufen. Die einzelnen erarbeiteten Shaderfunktionen und das Zusammenspiel zwischen den Shaderfunktionen wird in Kapitel 4 ausführlich vorgestellt. Auf der C++ Seite existiert für jede Shaderfunktion eine abstrakte Basisklasse, die Schnittstelle beinhaltet unter anderem eine Methode zum Erfragen des Shader-Quellcodes in einer gewünschten Shadersprache. Hiervon abgeleitete Klassen implementieren eine konkrete Realisierung für die Shaderfunktion. Die jeweilige Umsetzung kennt den dazugehörenden Shader-Quellcode in den unterstützten Shadersprachen sowie die benötigten Implementierungsabhängigen Shaderparameter, deren Zuweisung durch eine weitere Methode von außen angestoßen wird.

**Shader Compositor** Die Aufgabe des Shader Compositor besteht darin, von jeder der möglichen Shaderfunktionen jeweils eine konkrete Umsetzung auszuwählen. Wie diese Auswahl getroffen wird, ist hierbei abhängig von der gewählten Shader Compositor Implementierung. Je nach Anforderungen sind verschiedene Realisierungen denkbar, die jeweils andere Strategien bei der Zusammenstellung des Volumen-Renderer Shaders verfolgen.

In der Regel wird für die Auswahl die Szene analysiert. Existiert beispielsweise eine räumliche Überschneidung zwischen einem Volumenknoten und einem Knoten, der eine Schnittebene repräsentiert, so wird dies berücksichtigt. Weitere Auswahlkriterien sind die Visualisierungsoptionen des jeweiligen Volumenknoten sowie globale Szenen-Renderer Einstellungen. Sollte eine Shaderfunktion nicht benötigt werden, wird eine Null-Funktion verwendet, deren Implementierung keine Auswirkung auf den resultierenden Shader besitzt. Wird beispielsweise keine Beleuchtung benötigt, so liefert die für die Beleuchtung zuständige Null-Funktion die Farbe Schwarz zurück. Eine Addition von Null hat keine Auswirkung auf das Ergebnis. Shadercompiler sind in der Regel in der Lage, Operationen ohne Auswirkung zu erkennen. Dies resultiert darin, dass Null-Funktionen beim übersetzten Shader komplett wegfallen und somit keine Auswirkung auf die Bildwiederholrate haben. Dieses Konzept vereinfacht die Realisierung von Shaderfunktionen, da nicht berücksichtigt werden muss, ob eine Shaderfunktion eine reale Implementierung repräsentiert oder nicht. Der Fokus kann aufgrund dessen auf die Realisierung des gewünschten Ablaufes in einer Implementierung einer Shaderfunktion gelegt werden.

Für Shaderfunktionen, wie beispielsweise der Strahlverfolgung, wird in einem generierten Shader nur jeweils eine Umsetzung ausgewählt und verwendet. Hierdurch lassen sich allerdings nicht alle Anforderungen erfüllen. Wird ein Volumenknoten von mehreren Schnittebenen gleichzeitig überlappt, müssen diese Schnittebenen auf das Ergebnis eine entsprechende Auswirkung besitzen. Diese Aufgabe wurde über ein Template-Konzept gelöst. Aus Sicht des Aufrufers der Template-Shaderfunktion wird weiterhin nur ein einfacher Funktionsaufruf ausgeführt. Die Template Implementierung einer Shaderfunktion besteht wiederum aus einer Menge an Funktionsaufrufen. Die Implementierung des Templates wird auf C++ Seite durch einfache Textersetzung automatisiert aufgestellt und als Shader-Quellcode dem Shader Compositor zurückgeliefert.

Der Shader Compositor führt am Ende eine einfache Stringkonkatenation durch und über gibt den so generierten Shader an die Grafikschnittstelle. Das Ergebnis ist eine Instanz des Volumen-Renderer Shaders der die aktuellen Anforderungen erfüllt. Innerhalb dieser Instanz wird intern eine Liste der gewählten Implementierungen gespeichert, hierüber werden von dem erstellten System die Implementierungsabhängigen Shaderparameter gesetzt. Jede Shaderzusammenstellung lässt sich durch die gewählte Konfiguration eindeutig identifizieren. Ein zur Laufzeit dynamisch zusammengestellter Shader wird nur einmalig erzeugt und kann anschließend beliebig oft verwendet werden.

Die Shaderfunktionen sind auf C++ Seite in das RTTI-System der verwendeten Engine eingehängt. Prinzipiell lassen sich somit neue Implementierungen für Shaderfunktionen nachträglich dynamisch über Plugins hinzufügen. Hierdurch lassen sich beispielsweise neue Verfahren zur Gradientenschätzung von außen als loses Plugin hinzufügen, ohne das hierfür

der gesamte Volumen-Renderer Shader umgeschrieben werden müsste. Aus zeitlichen Gründen wurde dies jedoch beim erstellten konkreten Shader Compositor nicht berücksichtigt. Eine alternative Shader Compositor Implementierung ist aufgrund der Softwarearchitektur jedoch nachträglich einfügbar.

## 3.4 Strahlgenerierung

Während der Rasterisierung erzeugt die Grafikkarte Fragmente. Ein Fragment entspricht im folgenden einem Strahl durch das Volumen. Pro Fragment wird ein Fragmentshader abgearbeitet. Dieser kann als C-Programm aufgefasst werden welches von außen Parameter bekommt und vordefinierte Ausgaben liefert. Die Fragmentshader Ausgabe ist in der Regel ein Farbwert mit vier Komponenten. Ebenfalls können Tiefenwerte ausgegeben werden. Traditionell erhalten Fragmentshader die Eingaben vom Vertexshader. Weitere im Laufe der vergangenen Jahre hinzugekommenen Shader Zwischenstufen werden im Rahmen dieser Arbeit nicht behandelt. In modernen Shadersprachen liegen zusätzliche automatische Eingaben pro Fragment vor, beispielsweise die Bildschirmposition.

**Problemstellung** Bei der Strahlgenerierung muss pro Fragment der Eintrittspunkt  $\mathbf{r}_s \in \mathbb{R}^3$  und Austrittspunkt  $\mathbf{r}_e \in \mathbb{R}^3$  des Strahls im Volumen ermittelt werden. Aus diesen zwei Punkten lässt sich die Strahlänge

$$l = \sqrt{\mathbf{r}_e - \mathbf{r}_s}, l \in [0, 1] \quad (3.1)$$

sowie die normalisierte Strahlrichtung

$$\mathbf{r}_d = \frac{\mathbf{r}_e - \mathbf{r}_s}{l}, \mathbf{r}_d \in \mathbb{R}^3 \quad (3.2)$$

errechnen. Die zu ermittelnden Parameter des Strahls

$$\mathbf{r}(t) = \mathbf{r}_s + t\mathbf{r}_d, t \in [0, 1] \quad (3.3)$$

wurden in Abbildung 3.4 zusammengefasst. Per Definition ist ein Strahl in beide Richtungen unendlich lang. Da allerdings der Begriff *Raycasting* gebräuchlich ist und es umständlich wäre, ebenfalls den Begriff der Geraden zu verwenden, wird im Rahmen dieser Arbeit von z. B. Strahlstartposition statt Geradenstartposition gesprochen um Unklarheiten zu vermeiden.

**Funktionssignatur und Ablauf** Bei der Strahlgenerierung handelt es sich gleichzeitig um den in Quellcode 3.1 abgebildeten Einstiegspunkt des Fragmentshaders. Eine eigene Funktions-

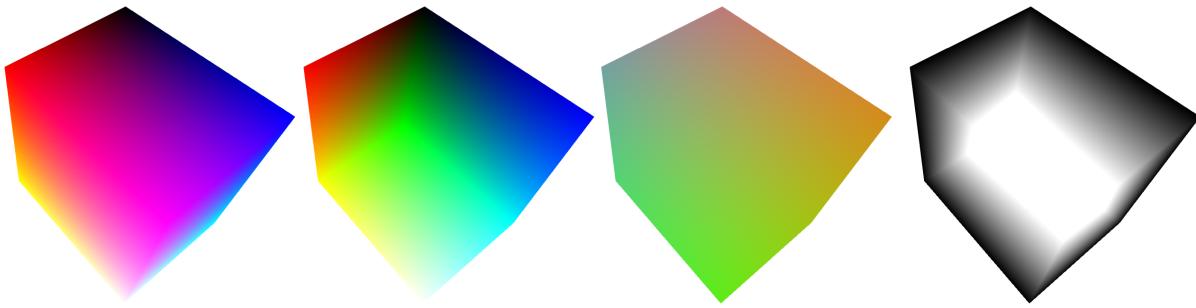


Bild 3.4: Von links nach rechts: Strahlstartposition  $r_s \in \mathbb{R}^3$ , Strahlendposition  $r_e \in \mathbb{R}^3$ , normalisierte Strahlrichtung  $r_d \in \mathbb{R}^3$  und Strahllänge  $l \in [0, 1]$ .

```
1 void main();
```

Quellcode 3.1: Einstiegspunkt des Volumen-Rendering Fragmentshaders.

signatur ist aufgrund dessen unnötig. Implementierungen müssen folgenden Ablauf umsetzen:

1. Parameter des Strahls ermitteln
2. Aufruf der Shaderfunktion 4.1 für den Strahlschnitt
3. Aufruf der Shaderfunktion 4.2 für künstliches Rauschen
4. Aufruf der Shaderfunktion 4.3 für die Strahlverfolgung
5. Berechneten Farbwert als Fragmentshader Ergebnis zurückgeben

Wird die Reihenfolge von Strahlschnitt und künstlichem Rauschen vertauscht, so fällt beim resultierenden Bild unangenehm auf, dass an Stellen an denen ein Schnitt stattfand, kein künstliches Rauschen vorhanden ist. Zur Ermittlung der Ein- und Austrittspunkte des Strahls im Volumen existieren mehrere Realisierungsmöglichkeiten, die im folgenden erläutert werden sollen.

**Ein- und Austrittspunkt des Strahls über Rasterisierung** [Krü03] präsentiert Beschleunigungstechniken für GPU-basiertes Shadermodell 2 Volumen-Rendering über Raycasting. Das Verfahren beruht darauf von der GPU einen Würfel rasterisieren zu lassen um Fragmente zu erzeugen, deren Farbwert im Anschluss über einen Raycasting Fragmentshader ermittelt wird. Jeder Vertex der Würfelgeometrie besitzt eine normalisierte Koordinate innerhalb des Volumens, diese wird während der Rasterisierung von der Grafikkarte interpoliert. Der von [Krü03] vorgestellte Algorithmus entstand zu einer Zeit, in der die Anzahl der Fragmentshader Instruktionen noch stark begrenzt war. Daher wurde das Rendern in mehrere Renderschritte unterteilt:

1. Der Eintrittspunkt in das Volumen wird ermittelt, indem die Vorderseite des Würfels in eine Textur gezeichnet wird. Diese Textur muss hierbei die gleiche Größe wie der aktuelle Bildpuffer, beispielsweise der des Ausgabefensters, besitzen. Es werden keine Farbwerte gespeichert sondern Positionen im Volumen, daher sollte in eine Gleitkommazahlen-Textur gezeichnet werden [Roe03].
2. In einem weiteren Renderschritt wird die Rückseite des Würfels in eine weitere Textur gezeichnet, um die Austrittsposition aus dem Volumen zu ermitteln. Dieser Renderschritt nimmt als weitere Eingabe die im vorherigen Renderschritt erzeugte Textur mit dem Eintrittspunkt pro Fragment. Der verwendete Fragments shader errechnet die weiteren Strahlparameter und schreibt die Ergebnisse in die Textur, in die gerade gezeichnet wird. Die Strahlrichtung wird im RGB-Kanal abgelegt, die Strahllänge hingegen im Alpha-Kanal.
3. Die folgenden Renderschritte nutzen die beiden Texturen mit den Strahlparametern pro Fragment und führen das eigentliche Volumen-Rendering durch. Die Anzahl der pro Renderschritt ausgeführten Schritte entlang des Strahls hängt hierbei von der Instruktionsbegrenzung der verwendeten GPU ab. Zwischen den Renderschritten sind weitere Optimierungsmöglichkeiten wie z. B. frühzeitiger Abbruch der Strahlverfolgung möglich [Krü03].

Dank der GPU Weiterentwicklung lässt sich der beschriebene Algorithmus vereinfachen [Sch05]. Derzeit ist Shadermodell 5 aktuell, jedoch ist bereits Shadermodell 3 dank Unterstützung echter Schleifen ausreichend für die folgende Vereinfachung des Algorithmus. Renderschritt 1 bleibt unverändert. Alle folgenden Renderschritte lassen sich hingegen zu einem einzigen Renderschritt verschmelzen. Nachdem wie in Renderschritt 2 beschrieben alle benötigten Strahlparameter ermittelt wurden, folgt direkt im gleichen Fragments shader die eigentliche Strahlverfolgung. Aufgrund der Unterstützung dynamischer Verzweigungen sind Optimierungen wie der frühzeitige Abbruch der Strahlverfolgung weiterhin möglich. Selbst über die beschriebene Vereinfachung sind immer noch zwei Renderschritte nötig, dies führt zu mehr Kommunikation mit der Grafiksschnittstelle. Gerade Zeichenaufrufe sind zeitaufwändige Funktionsaufrufe, die soweit möglich vermieden werden sollten. In der Regel werden nicht viele Volumen gleichzeitig dargestellt, daher ist dieser Nachteil je nach Anwendungsfall vernachlässigbar.

Weder der Vertexshader in Quellcode 3.2, noch der Fragments shader Quellcode 3.3 zur Berechnung des Ein- und Austrittspunktes des Strahls mit anschließender Strahlverfolgung sind sonderlich komplex. Alle weiteren in Kapitel 4 vorgestellten Teilschritte werden von diesem Fragments shader angesteuert.

Vorteilhaft an diesem Algorithmus ist, dass sich prinzipiell andere Hilfsgeometrien zum Rendern verwenden lassen. Um die Form des visualisierten Volumens zu verändern, könnte statt

```

1 in vec3 VertexPosition;
2 in vec3 VertexTexCoord;
3 out vec3 re; // Strahlendposition
4 uniform mat4 mvp; // Object-Space zu Clip-Space Matrix
5 void main()
6 {
7     gl_Position = mvp*vec4(VertexPosition, 1.0);
8     re = VertexTexCoord;
9 }
```

Quellcode 3.2: Vertexshader zur Berechnung des Austrittspunktes des Strahls.

```

1 in vec3 re; // Strahlendposition
2 layout(location = 0) out vec4 fc; // Fragmentfarbe
3 uniform sampler2D ft; // Textur mit Strahlstartpositionen
4 uniform float StepSize; // Schrittweite
5 void main()
6 {
7     vec3 rs = texelFetch(ft, ivec2(gl_FragCoord.xy), 0).rgb;
8     vec3 rd = re - rs;
9     float l = length(rd);
10    vec3 rdn = rd/l;
11    vec3 stepDelta = rdn*StepSize;
12    ClipRay(rs, rdn, l);
13    rs += stepDelta*JitterPosition(rs);
14    fc = RayTraversal(rs, int(l/StepSize), stepDelta, l);
15 }
```

Quellcode 3.3: Fragmentsader zur Berechnung des Ein- und Austrittspunkt des Strahls über Rasterisierung mit anschließender Strahlverfolgung.

eines Würfels beispielsweise eine Kugel oder ein Zylinder rasterisiert werden. Eine Veränderung des Fragmentsaders wäre hierfür nicht erforderlich. Die Hilfsgeometrie wird über Polygone gezeichnet. Bei aktiviertem Tiefentest verwirft die GPU daher weiterhin automatisch Fragmente gemäß der gewählten Tiefenpuffer-Einstellung. Farbwerte im Bildpuffer werden in diesem Fall nicht aktualisiert. In der traditionellen OpenGL Grafikpipeline findet der Tiefentest erst statt, nachdem der Fragmentsader ausgeführt wurde. Die Strahlverfolgung wird somit ebenfalls für Fragmente ausgeführt, die nach Aussage des Tiefenpuffer verdeckt werden und somit keinen sichtbaren Einfluss auf das resultierende Bild besitzen. Um Füllrate einzusparen, führten Grafikkartenhersteller eine als *EarlyZ* bekannte Optimierung ein. Hierdurch wird automatisch ein Tiefentest durchgeführt noch bevor ein rechenintensiver Fragmentsader gestartet

wird. Die Voraussetzungen für das automatische *EarlyZ* werden in [NVI], [Adva] und [Advb] beschrieben.

**Ein- und Austrittspunkt des Strahls über Strahl/Würfel-Schnittpunktberechnung** Unter Zuhilfenahme einer Strahl/Würfel-Schnittpunktberechnung lässt sich das in [Krü03] vorgestellte Verfahren zum GPU-basierten Volumen-Rendering auf einen Renderschritt reduzieren. Texturzugriffe werden durch Berechnungen ausgetauscht. Speicherzugriffe sind in der Regel ein Flaschenhals, während eine GPU Berechnungen sehr effizient ausführen kann. Die Kommunikation mit der Grafiksschnittstelle wird hierdurch ebenfalls reduziert. Das im folgenden vorgestellte Verfahren bietet keine deutlichen Leistungsvorteile, da die Strahlverfolgung den Großteil der benötigten Rechenzeit ausmacht. Vorteilhaft ist jedoch, dass sich dieses Verfahren recht einfach auf GPGPU-Sprachen wie beispielsweise CUDA oder OpenCL übertragen lässt.

Ein Assembler-Shader einer Strahl/Würfel-Schnittpunktberechnung ist in [Roe03] zu finden. [Gre05] stellt eine Strahl/Würfel-Schnittpunktberechnung innerhalb des Fragmentsshaders mit Hilfe der *Slab*-Methode von Kay und Kayjia [Kay86] vor und verwendet hierfür eine lesbare Shader-Hochsprache. Weiterhin müssen Fragmente erzeugt werden, im einfachsten Fall reichen hierfür zwei Dreiecke für ein Rechteck, welches den gesamten Bildschirm ausfüllt. Um Füllrate einzusparen, macht es jedoch Sinn, die Rückseiten einer Würfelgeometrie zu rasterisieren. Die Vertices der verwendeten Hilfsgeometrie benötigen lediglich eine Koordinate im Raum, die normalisierten Volumenkoordinaten werden im Fragmentsader errechnet.

Da das Volumenkoordinatensystem innerhalb des Fragmentsshaders normalisiert ist, handelt es sich bei dem Würfel für die Schnittpunktberechnung um einen Einheitswürfel. Als Ursprung des Strahls kann die Kameraposition im Volumenkoordinatensystem verwendet werden. Die Strahlrichtung lässt sich ermitteln, indem zwei Punkte im Fensterkoordinatensystem in das Volumenkoordinatensystem rückprojiziert werden [Had06, S. 287]. Anschließend wird anhand dieser Punkte der Richtungsvektor berechnet. Der erste Punkt befindet sich an der Fragmentposition und Tiefenwert 0, während sich der zweite Punkt an Fragmentposition und Tiefenwert 1 befindet.

Eine einfachere Lösung zur Berechnung der Strahlrichtung besteht darin, der Würfelgeometrie wie in [Krü03] beschrieben weiterhin Volumenkoordinaten zuzuweisen. Somit ist der Austrittspunkt des Strahls bekannt. Als Ursprung des Strahls wird die Kameraposition im Volumenkoordinatensystem verwendet. Anhand der zwei Punkte auf dem Strahl lässt sich nun die Strahlrichtung bestimmen.

**Abtastrate** Gemäß dem *Nyquist-Shannon-Abtasttheorem* [Sha49, Wyn98, Uns00] muss beim umwandeln von analogen in digitale Signale, die Abtastfrequenz mehr als doppelt so groß sein

als die höchste im Signal vorkommende Frequenz. Andernfalls lässt sich das Signal nicht korrekt rekonstruieren. Beim Volumen-Rendering liegen die Daten in der Regel bereits digital vor, es gilt nun die z. B. von einem Scanner digitalisierten Daten bestmöglich zu rekonstruieren. Durch die Digitalisierung verlorengegangene Details im Signal sind jedoch nicht vollständig wiederherstellbar. Die mit qualitativ minderwertigen Volumendaten erzielbare Bildqualität ist hierdurch eingeschränkt. Um das ursprüngliche kontinuierliche Signal angemessen anhand der diskreten Daten wiederherzustellen, müssen mindestens zwei Stichproben pro kleinste Voxel zu Voxel Entfernung genommen werden. Nach [Roe03] muss mindestens eine vierfache Überabtastung stattfinden, um das Strahl-Integral mit einer ausreichenden Genauigkeit zu rekonstruieren. Dies gilt ebenfalls beim Einsatz von Pre-Integration. Eine derart hohe Abtastrate beeinträchtigt das Laufzeitverhalten beachtlich. Eine Optimierungsmöglichkeit besteht darin, die Abtastrate während der Strahlverfolgung wie in [Wre10, S. 256] und [Had06, S. 218] beschrieben dynamisch zu verändern. Um eine stabile Bildwiederholrate zu erzielen, lässt sich unter anderem die globale Abtastrate dynamisch zur Laufzeit regulieren. Bei nicht ausgeschöpften Leistungsressourcen kann durch eine automatisch erhöhte Abtastrate eine bessere Bildqualität erzielt werden [Cor11].

Nach [RS06] sind MRI-Datensätze schwer zu visualisieren, da verschiedene Gewebetypen von den gleichen Skalarwerten repräsentiert werden. Gleches gilt für CT-Datensätze [Mei00]. Bei einer Visualisierung werden dadurch interessante Strukturen durch weniger interessante Strukturen verdeckt. Aufgrund der blickwinkelabhängigen Verdeckung lässt sich das Problem durch ein verändern des Transferfunktion-Designs nicht so einfach lösen. [RS06] schlägt eine Lösung des Problems durch eine Veränderung der Strahlverfolgung vor, während [Ras08] die Abtastrate individueller Strahlen über eine Textur verändert.

**Überspringen leerer Zwischenräume im Volumen** Im Kontext des Volumen-Rendering ist ein Voxel per Definition leer, wenn seine Opazität null ist [Lev90]. Das von [Krü03] präsentierte Verfahren zur Berechnung des Ein- und Austrittspunktes des Strahls über Rasterisierung ermöglicht eine einfache Realisierung des Überspringens leerer Zwischenräume im Volumen<sup>4</sup>, da jeder Vertex der Hilfsgeometrie ebenfalls eine Position innerhalb des Volumenkoordinatensystems besitzt. [Krü03] und [Sch05] schlagen vor, das Volumen in gleichgroße Blöcke zu unterteilen. Zum damaligen Zeitpunkt konnte damit etwas Arbeit auf die beim Volumen-Rendering ansonsten unerforderten Vertexshader-Einheiten ausgelagert werden. Heutige GPU-Architekturen besitzen eine vereinheitlichte Shader-Architektur [Advb, S. 2]. Es wird dynamisch entschieden ob eine Recheneinheit für Vertexshader oder Fragmentshader verwendet wird. Das Argument des Einsparens von Füllrate hingegen ist weiterhin gültig.

---

<sup>4</sup>Engl. *Empty-Space-Skipping*

Das Überspringen leerer Zwischenräume erfordert im Allgemeinen zusätzliche Datenstrukturen, die jeweils aktualisiert werden müssen, sobald sich etwas an der Klassifikation oder den Volumendaten selbst ändert. Für 4-D-Volumendaten ist diese Optimierung deshalb ungeeignet. Sich über die Zeit hinweg nicht verändernde Volumendaten hingegen können hiervon unter Umständen profitieren. Ein in einer Szene mehrmals gleichzeitig dargestelltes Volumen über unterschiedliche Klassifikationen kann es nötig machen, das pro Volumen mehrere solcher zusätzlichen Datenstrukturen verwaltet werden müssen. Bei dicht besetzten Volumendaten, wie diese im medizinischen Umfeld in der Regel vorhanden sind, hält sich der Leistungsgewinn durch so geartete Optimierungen in Grenzen. Im schlimmsten Fall kann aufgrund der zusätzlichen Arbeit die Leistung durch die hier vorgestellte Optimierung verschlechtert werden. Aufgrund dessen soll an dieser Stelle nicht weiter auf das Überspringen leerer Zwischenräume im Volumen eingegangen werden.

**Kamera im Volumen** Bei polygonbasierender Computergrafik werden lediglich Oberflächen dargestellt. Sobald die Kamera in ein so visualisiertes Objekt eintaucht, wird ersichtlich, dass dieses keinen Inhalt besitzt. Bei Volumen-basierender Computergrafik hingegen ist es möglich, mit der Kamera in das so dargestellte Objekt zu fahren. Weitere im Objekt liegende und ursprünglich verborgene Details werden dabei sichtbar. Im medizinischen Umfeld ist die Möglichkeit, mit der Kamera das Innere eines Volumen zu erkunden, besonders wichtig für die virtuelle Endoskopie [Sch06]. Bei dieser ist die perspektivische Projektion ein ausschlaggebender Faktor. Vor etwas über 10 Jahren stellte Volumen-Rendering mit perspektivischer Projektion durchaus noch ein Problem da. So beschreibt [Pfi99] das bei der Volumen-Rendering Spezialhardware *VolumePro* lediglich orthografische Projektion realisiert werden konnte, da perspektivische Projektion zu komplex gewesen sei. Heutzutage ist beim GPU-basierenden Volumen-Rendering die perspektivische Projektion nicht weiter erwähnenswert, da es sich im Vertexshader nur um eine einfache Matrix-Vektor Multiplikation handelt. Der Inhalt der Matrize ist für eine handelsübliche Grafikkarte nicht weiter von Interesse.

Wird der Ein- und Austrittspunkt des Strahls im Volumen über die vorgestellte Strahl/Würfel-Schnittpunktberechnung ermittelt und die benötigten Fragmente beispielsweise über das Zeichnen der Rückseite einer Würfelgeometrie erzeugt, so kann die Kamera automatisch in das Volumen eindringen, ohne das dies einen zu berücksichtigender Sonderfall darstellt. Aufgrund der Rasterisierung von Polygonen findet durch die Grafikpipeline weiterhin ein Schnitt durch die Ebene nahe der Kamera statt. Bei ungünstigen Konstellationen kann dies beim Austritt aus dem Volumen sichtbare Probleme verursachen, wie in Abbildung 3.5 zu erkennen. Eine Abschwächung der Problematik lässt sich über die OpenGL-Erweiterung *GL\_ARB\_depth\_clamp* erzielen. Hierdurch kann ein Schnitt von rasterisierten Polygonen ver-

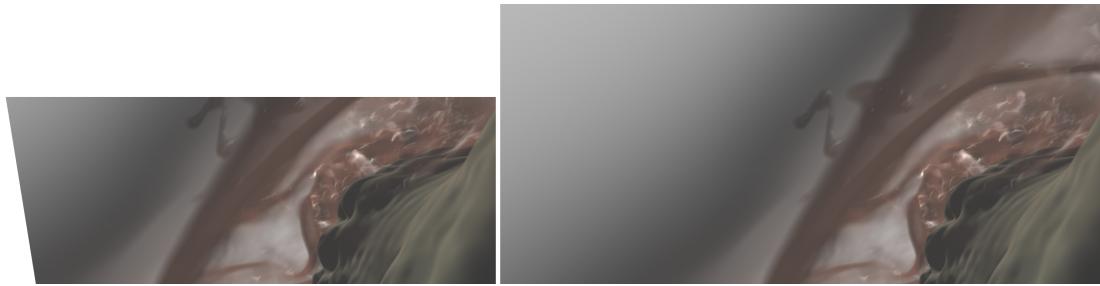


Bild 3.5: Unerwünschter Schnitt durch die Ebene nahe der Bildebene. Links ohne *GL\_ARB\_depth\_clamp*, rechts mit. Volumendatensatz *Head (Visible Male)*.

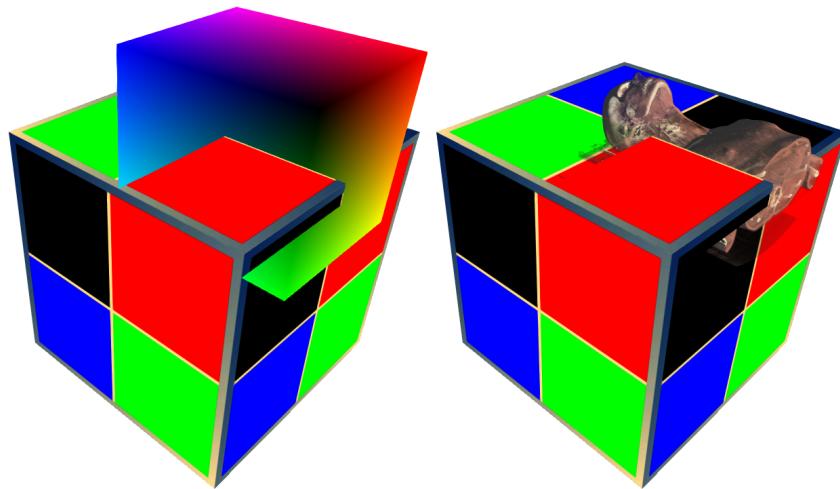


Bild 3.6: Tiefenpuffer Probleme bei der Strahlgenerierung. Links Hilfsgeometrie, rechts DVR. Volumendatensatz *Thomas*.

hindert werden und Fragmente werden nicht mehr verworfen. Werden jedoch Polygone von der Grafikpipeline komplett ignoriert, so verschwindet das Volumen beim Austritt weiterhin frühzeitig. In Abhängigkeit der Skalierung des Volumens und der Bewegungsgeschwindigkeit der Kamera, sollte die Schnittebene nahe der Kamera nicht zu weit vom Nullpunkt entfernt sein.

Bei der vorgestellten Lösung treten im Zusammenspiel mit einem Tiefenpuffer Test, wie in Abbildung 3.6 erkennbar, Probleme auf. Um die Blockierung der Strahlgenerierung zu verhindern, muss hierzu der Tiefentest in der Grafikpipeline deaktiviert und statt dessen in einem Fragmenshader ausgeführt werden. Durch einen Test zwischen der Schnittebene nahe der Kamera und der Hilfsgeometrie könnte überprüft werden, ob die Kamera potentiell in das Volumen blickt. Solange kein Schnitt erfolgt, könnten die Vorderseiten mit effektivem Tiefentest gezeichnet werden, bei einem Schnitt hingegen könnte auf die hier vorgestellte Lösung umgeschaltet werden.

Das Erzeugen von Fragmenten durch das Rasterisieren der Vorderseiten der Hilfsgeometrie hingegen ist weniger Vorteilhaft. Dies lässt sich jedoch nicht vermeiden wenn der Ein- und Austrittspunkt des Strahls über Rasterisierung [Krü03] ermittelt wird um ein Überspringen leerer Zwischenräume im Volumen anhand der in [Krü03] und [Sch05] vorgestellten Verfahren zu realisieren. Sobald die Kamera Schnittebene nahe der Bildebene die Vorderseite der Hilfsgeometrie schneidet, werden an den entsprechenden Stellen von der Grafikkarte keine Fragmente mehr erzeugt. Es entstehen Löcher. Anstatt in das Volumen zu sehen, ist an diesen Stellen kein Volumen mehr sichtbar. Um die so entstehenden Lücken der Vorderseiten zu füllen, schlägt [Sch05] den Einsatz des Tiefenpuffers vor. Eine weitere Lösung besteht darin, zusätzliche Polygone zu erzeugen um die Lücke zu füllen. Hierzu ist eine Schnittpunktberechnung zwischen der Hilfsgeometrie und der Schnittebene notwendig. Diese Polygone lassen sich von der CPU berechnen. [RS05] hingegen schlägt hierfür den Einsatz eines Vertexshaders vor.



# Kapitel 4

## Volumen-Renderer Shaderfunktionen

In Kapitel 3 wurde die Volumen-Renderer Integration in eine bestehende 3-D-Engine erörtert. Ebenfalls wurde auf die Strahlgenerierung über eine klassische polygonbasierende Grafikschnittstelle eingegangen.

Das Ziel dieses Kapitels besteht darin, die Farbe eines Bildpunktes zu ermitteln. Die Grafikkarte führt diese Berechnungen parallel für zahlreiche Bildpunkte aus. Eine Synchronisation zwischen den einzelnen Bildpunkten ist nicht notwendig. Aufgrund der wohldefinierten Aufgabenstellung im Fragmentshader soll für weiterführende Hintergründe über Grafikprogrammierung auf [AM08] verwiesen werden.

Die Gliederung dieses Kapitels entspricht der im Rahmen dieser Arbeit erstellten Volumen-Renderer Realisierung. Um die Zusammenhänge zwischen den einzelnen Teilabschnitten herzustellen, wird jeweils eine Definition der Funktionssignatur und des zu realisierenden Ablaufes präsentiert, erst anschließend werden Details erörtert.

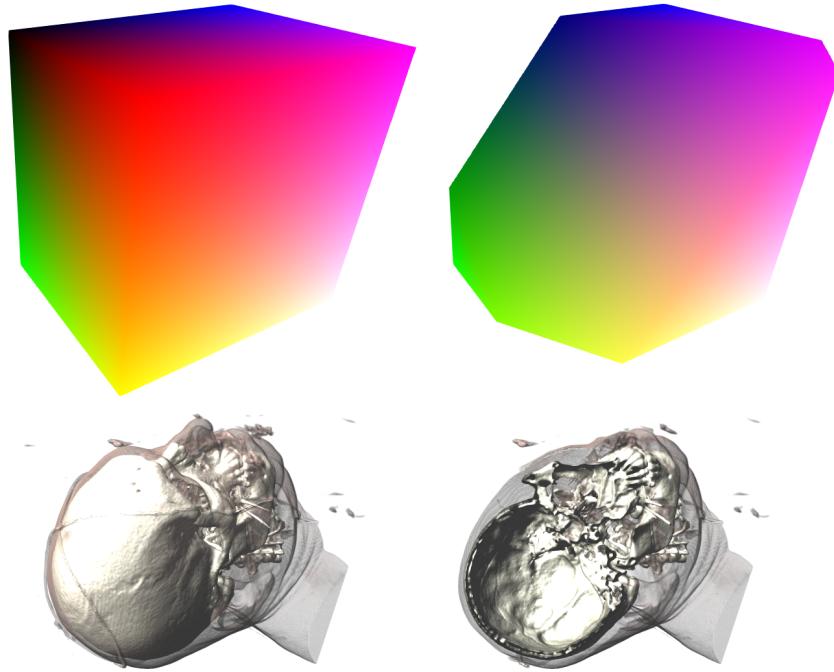
### 4.1 Strahlschnitt

Die Aufgabe des Strahlschnittes besteht darin, die Strahlstartposition und die Strahllänge zu modifizieren. Bei dieser Operation verändert sich lediglich die Strahllänge, es findet keine Richtungsänderung oder Segmentierung statt. Das realisierte Shadersystem sieht pro Strahl mehrere Aufrufe der im folgenden beschriebenen Shaderfunktion vor, hierdurch sind mehrere Schnittebenen gleichzeitig möglich.

**Funktionssignatur und Ablauf** Für den Strahlschnitt wird die in Quellcode 4.1 abgebildete Shaderfunktion aufgerufen. Als ersten Parameter wird die Strahlstartposition  $\left[ \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T, \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T \right]$  im Volumen übergeben. Dieser Parameter dient gleichzeitig als

```
1 void ClipRay(inout vec3 RayOrigin, vec3 RayDirection, inout
  float MaximumTravelLength);
```

Quellcode 4.1: Funktionssignatur für den Strahlschnitt.

Bild 4.1: Links ohne, rechts mit Schnittebene. Oben die Hilfsgeometrie, unten Volumen-Rendering. Volumendatensatz *Head (Visible Male)*.

Ausgabe der modifizierten Strahlstartposition. Die normalisierte Strahlrichtung innerhalb des Volumens wird als zweiter Parameter übergeben. Als letztes Argument erhält die Funktion die maximale Strahllänge innerhalb des Volumens, dies dient ebenfalls als Ausgabeparameter. Im folgenden werden zwei Implementierungen dieser Shaderfunktion vorgestellt.

**Schnittebene** Schnittebenen sind ein Standardwerkzeug in den meisten Volumen-Rendering Systemen [Had06, S. 382]. Wie in Abbildung 4.1 zu sehen, lassen sich über eine Schnittebene Teile des Volumens komplett ausblenden.

Die sich im Volumenkoordinatensystem befindende Schnittebene wird dem Fragmentshader als Hessesche Normalform [Bro01, S. 222]

$$\mathbf{r} \cdot \mathbf{n}_0 - p = 0 \quad (4.1)$$

mit Ortsvektor  $\mathbf{r}$ , senkrecht zur Ebene stehendem Normaleneinheitsvektor  $\mathbf{n}_0 \in \mathbb{R}^3$  und Abstand zum Koordinatenursprung  $p \in \mathbb{R}$  übergeben. Die Schnittebene kann folglich frei im Raum

ausgerichtet werden. In der erstellten Implementierung lassen sich hierfür visuelle Hilfsmittel nutzen, über die Position und Rotation direkt im 3-D-Fenster verändert werden können. Der Strahlschnitt lässt sich in folgende Arbeitsschritte unterteilen:

1. Strahlendposition

$$\mathbf{r}_e = \mathbf{r}_s + l \cdot \mathbf{r}_d \quad (4.2)$$

anhand der Strahlstartposition  $\mathbf{r}_s$ , Strahllänge  $l$  und der normalisierten Strahlrichtung  $\mathbf{r}_d$  ermitteln.

2. Als Ortsvektor  $\mathbf{r} \in \mathbb{R}^3$  der Hessesche Normalform die Strahlstartposition  $\mathbf{r}_s$  einsetzen um den Abstand  $d_s \in \mathbb{R}$  zur Schnittebene zu erhalten. Diesen Schritt für Strahlendposition  $\mathbf{r}_e$  wiederholen um den Abstand  $d_e \in \mathbb{R}$  zur Schnittebene zu erhalten.
3. Falls die Bedingung  $d_s \cdot d_e > 0$  erfüllt ist, liegen beide auf dem Strahl liegenden Positionen auf der gleichen Seite der Schnittebene. Der Strahl wurde vollständig geschnitten falls die Bedingung  $d_s < 0$  erfüllt ist, in diesem Fall wird der *MaximumTravelLength*-Parameter auf  $-1$  gesetzt und die Shaderfunktion wird beendet. Sollte  $d_s \geq 0$  eintreten, wird der Strahl nicht verändert und die Shaderfunktion wird beendet.
4. Falls die Bedingung  $d_s \cdot d_e > 0$  nicht erfüllt ist, findet eine Veränderung des Strahls statt. Der Punkt auf der negativen Seite der Schnittebene wird auf die Schnittebene gesetzt. Im abschließenden Schritt wird die neue Strahllänge ermittelt und über den *MaximumTravelLength*-Parameter als Ergebnis zurückgeliefert.

Der Quellcode 4.2 zeigt die resultierende Implementierung der Shaderfunktion.

**Tiefenpuffer** Für die Realisierung von Schnittgeometrien schlägt [Sch05] den Einsatz des Tiefenpuffers vor, um eine Erzeugung von Fragmenten zu unterbinden. Abseits des Ausblendens bestimmter Volumenteile unter Zuhilfenahme des Tiefenpuffers lässt sich dieser Puffer ebenfalls für weitere Funktionalitäten nutzen. Strahlen, die durch im Vordergrund liegende Objekte verdeckt werden, können komplett ignoriert werden. Dies führt zu einer Leistungsverbesserung. Die Strahlstartposition wird aufgrund dessen in der Implementierung der Shaderfunktion nicht verändert. Nicht transparente Objekte, die sich hinter dem gezeichneten Volumen befinden und räumlich komplett vom Volumen getrennt sind, lassen sich automatisch ohne Schwierigkeiten darstellen. Sobald eine räumliche Überschneidung von Volumen und nicht transparenten Objekten auftritt, kommt es hingegen zu sichtbaren Bildfehlern, die sich störend auf die Tiefenwahrnehmung auswirken.

```

1 uniform vec4 ClipPlane; // xyz=N0, w=p
2 void ClipRay(inout vec3 rs, vec3 rd, inout float maxLength)
3 {
4     vec3 re = rs + rd*maxLength;
5     float ds = dot(ClipPlane.xyz, rs) + ClipPlane.w;
6     float de = dot(ClipPlane.xyz, re) + ClipPlane.w;
7     if (ds*de > 0.0) {
8         if (ds < 0.0)
9             maxLength = -1.0;
10    } else {
11        if (ds < 0.0) {
12            rs -= rd*(ds/dot(ClipPlane.xyz, rd));
13        } else {
14            re -= rd*(de/dot(ClipPlane.xyz, rd));
15        }
16        float l = length(re - rs);
17        if (maxLength > l)
18            maxLength = l;
19    }
20 }
```

Quellcode 4.2: Fragmentshader für den Schnitt eines Strahls mit einer Ebene. Parameternamen in der Funktionssignatur aus Platzgründen gekürzt.

Für die beschriebene Situation existieren zahlreiche Anwendungsfälle. Bei Echtzeitgrafik, wie diese beispielsweise bei Videospielen üblich ist, erwartet der Benutzer eine korrekte Darstellung von sich überlappenden Objekten. Bei über Volumen-Rendering dargestellten Nebel müssen sich Objekte korrekt in diesem frei bewegen können. Im medizinischen Anwendungsfall muss die räumliche Lage von Instrumenten oder den zu Visualisierungszwecken aus Polygone modellierten Organen ersichtlich sein.

Für nicht transparente Objekte lässt sich zur Lösung dieses Problems der Tiefenpuffer verwenden. Moderne Grafikpipelines zeichnen in der Regel nicht direkt in den Bildpuffer des Fensters, sondern in eine Textur. Gleicher gilt für den Tiefenpuffer, der in einer Tiefentextur abgelegt wird und in einem Shader ausgelesen werden kann. Anhand der aktuellen Fragmentposition und des Tiefenwertes dieses Fragmentes aus der Tiefentextur, lässt sich über Rückprojektion ein Punkt im Volumenkoordinatensystem ermitteln. Dieser so errechnete Punkt ist der Schnittpunkt mit einem Objekt, das in den Tiefenpuffer schrieb, oder stellt einen Punkt weit im Hintergrund dar, falls der Tiefepuffer an der Stelle nicht beschrieben wurde. Es folgt ein Ermitteln des Abstandes zwischen der errechneten Position und der Strahlstartposition. Ist die aktuelle Strahllänge größer als dieser ermittelte Abstand, so wird die Strahllänge entsprechend

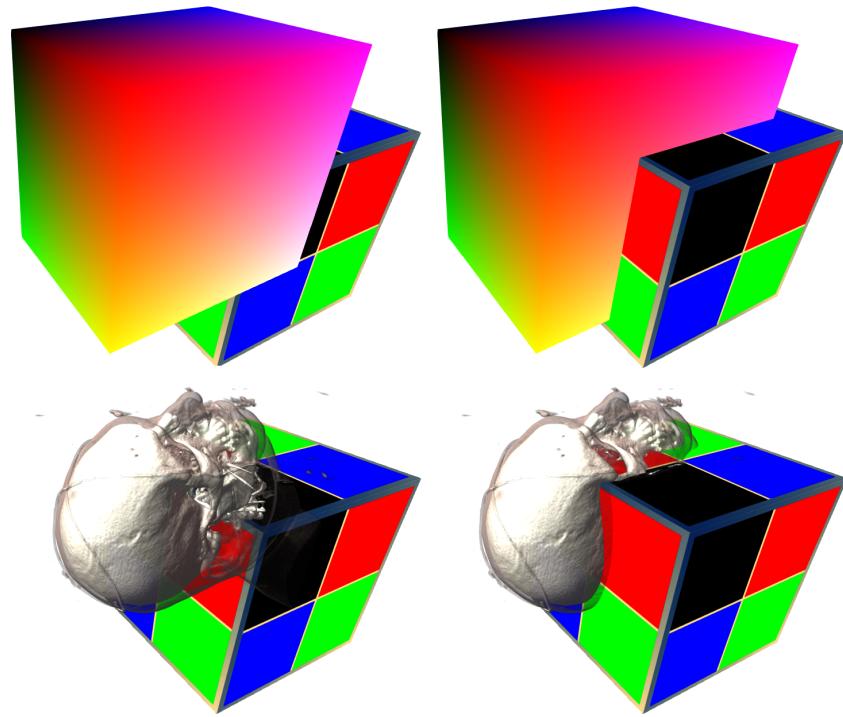


Bild 4.2: Links ohne, rechts mit Tiefentextur. Oben die Hilfsgeometrie, unten Volumen-Rendering. Volumendatensatz *Head (Visible Male)*.

verkürzt. Die Strahlverfolgung wird wie in Abbildung 4.2 zu erkennen, frühzeitig abgebrochen. Für weitere Hintergrundinformationen sowie Implementierungsdetails sei an dieser Stelle an [Kra06] und [Had06, S. 285] verwiesen.

## 4.2 Künstliches Rauschen

Bei einer niedrigen Abtastrate, die somit das Nyquist-Shannon-Abtasttheorem verletzt, werden kreisförmige Holz-Korn-Artefakte<sup>1</sup> deutlich sichtbar. Gerade in einem bewegten Bild wirkt dieser Effekt störend. Zur Beseitigung dieser Artefakte muss die Abtastrate erhöht werden [Eng04].

Als Alternative zur kostspieligen Beseitigung der Holz-Korn-Artefakte durch eine Erhöhung der Abtastrate schlägt [Had06, S. 220] einen nichtdeterministischen Monte-Carlo-Algorithmus vor. Die Artefakte werden über ein zufälliges künstliches Rauschen versteckt. Ein leichtes Rauschen im Bild ist weniger störend als deutlich erkennbare kreisförmige und wandernde Muster. Die technische Umsetzung ist nicht Aufwändig, es gilt lediglich die Startposition des Strahls etwas entlang der Strahlrichtung zu verschieben.

---

<sup>1</sup> Engl. *wood-grain artifacts*, auch bekannt als Moiré-Effekt



Bild 4.3: Von links nach rechts: Abtastrate 100 %, 50 %, 20 %, 10 % und 1 %, unten mit künstlichem Rauschen. Volumendatensatz *Head (Visible Male)*.

In Abbildung 4.3 wurde das Volumen über verschiedene Abtastraten gezeichnet, einmal mit und einmal ohne künstliches Rauschen. In der Praxis ist eine Abtastrate von 10 % oder gar 1 % des laut Abtasttheorems benötigten nicht sinnvoll. Bei einer derart geringen Abtastrate sind die Holz-Korn-Artefakte und das künstliche Rauschen deutlich zu erkennen.

**Funktionssignatur und Ablauf** Die in Quellcode 4.3 zu sehende Shaderfunktion erhält als Eingabe eine Position  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  im Volumen und liefert als Rückgabe einen Wert im Intervall  $[0, 1]$ . Diese Rückgabe wird im Nachfolgenden vom Aufrufer mit der Strahl-

```
float JitterPosition(vec3 Position);
```

Quellcode 4.3: Funktionssignatur für künstliches Rauschen.

schrittweite multipliziert und auf die Startposition des Strahls im Volumen addiert. Dies resultiert in einem künstlichen Rauschen, wodurch die Holz-Korn-Artefakte verdeckt werden.

**Realisierungsmöglichkeiten** [Had06] stellt eine Realisierung des künstlichen Rauschens durch Zuhilfenahme einer Textur vor. Eine kleine Textur, wie beispielsweise  $32 \times 32$  mit einer einzigen Komponente pro Texel, wird hierfür mit zufälligen Werten gefüllt. Der Texturzugriff wird auf eine sich wiederholende Textur eingestellt. Der Fragmentshader greift anhand der

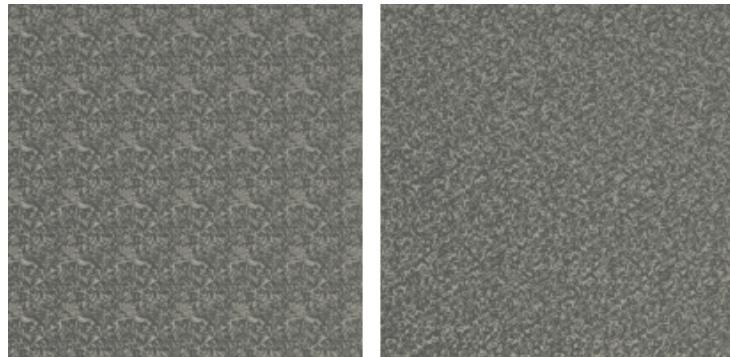


Bild 4.4: Realisierungsmöglichkeiten für künstliches Rauschen, links Textur mit Zufallszahlen, rechts trigonometrische Funktionen.

Fragmentposition auf diese Zufallstextur zu und erhält einen normalisierten Wert. Dieser wird als Ergebnis der Shaderfunktion zurückgeliefert.

Eine Realisierung über trigonometrische Funktionen, wie in Quellcode 4.4 abgebildet, ist einfacher. Es werden keine weiteren Shaderparameter benötigt und somit muss keine Textur mit Zufallszahlen generiert werden. Texturzugriffe sind aufgrund der nötigen Speicherverwal-

```

1 float JitterPosition(vec3 Position)
2 {
3     return fract(cos(gl_FragCoord.x*11.55 + gl_FragCoord.y
4 *42.123)*35684.525);
}
```

Quellcode 4.4: Realisierung des künstlichen Rauschens über trigonometrische Funktionen.

tung im Allgemeinen Zeitaufwändiger als mathematische Funktionen. Der Flaschenhals beim Volumen-Rendering ist allerdings bei der Strahlverfolgung zu finden. Ein einziger Texturzugriff für künstliches Rauschen fällt somit nicht ins Gewicht. Trotz dessen ist es ratsam, die Speicherverwaltung zu entlasten wenn dies möglich ist. In Abbildung 4.4 ist eine Nahansicht eines gezeichneten Volumens zu sehen. Es ist deutlich zu erkennen, dass es bei der Lösung über eine sich wiederholende Textur mit Zufallszahlen zu einer neuen Musterbildung kommt, während bei der Lösung über trigonometrische Funktionen keine unmittelbar sichtbaren Muster existieren.

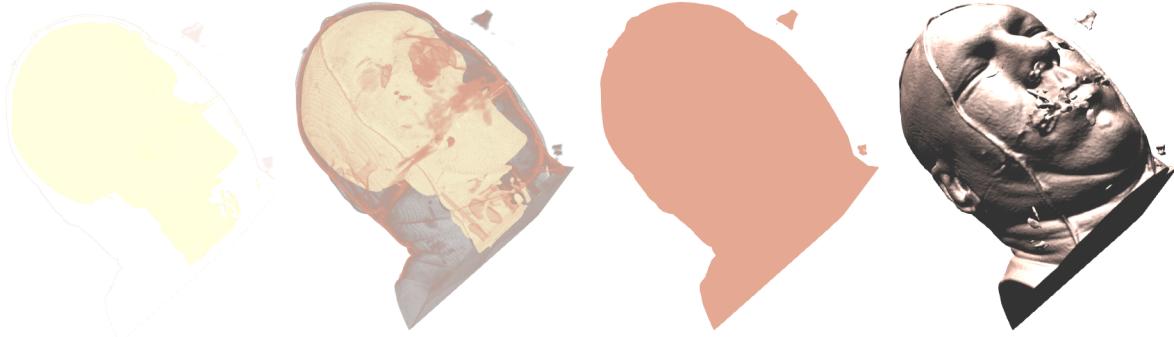


Bild 4.5: Von links nach rechts: MIP, Front-To-Back, Isofläche und Isofläche mit Beleuchtung. Jeweils gleiche Transferfunktion. Volumendatensatz *Head (Visible Male)*.

### 4.3 Strahlverfolgung

Bei der Strahlverfolgung existieren verschiedene Herangehensweisen, je nach gewünschtem Ergebnis. Einige der bekannteren Vorgehensweisen bei der Strahlverfolgung wurden in Abbildung 4.5 zusammengefasst.

**Funktionssignatur und Ablauf** Die Shaderfunktion in Quellcode 4.5 erhält als Eingabe die Startposition  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  des Strahls im Volumen, die Anzahl der Schritte  $[0, n]$  bei der Strahlverfolgung, die Schrittgröße  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  sowie die maximale Reiselänge  $[0, 1]$  des Strahls innerhalb des Volumens. Das Ergebnis der Strahlverfolgung ist der

```
vec4 RayTraversal(vec3 StartPosition, int NumberOfSteps, vec3 StepPositionDelta, float MaximumTravelLength);
```

Quellcode 4.5: Funktionssignatur für die Strahlverfolgung.

aus vier Komponenten bestehende Farbwert des Bildpunktes. Im Rahmen der Strahlverfolgung werden von der gewählten Implementierung in der Regel folgende Shaderfunktion aufgerufen:

1. Aufruf der Shaderfunktion 4.4 für Schnitt innerhalb des Volumens
2. Aufruf der Shaderfunktion 4.5 für die Rekonstruktion der Volumendaten
3. Aufruf der Shaderfunktion 4.6 für das Shading

In den folgenden Abschnitten wird eine Auswahl an unterschiedlichen Strategien bei der Strahlverfolgung vorgestellt.

**Isofläche** Bei polygonbasierender Computergrafik werden die Oberflächen von Objekten dargestellt. Für Volumen-Rendering existieren oberflächenextrahierende Methoden, beispielsweise der *Marching Cubes*-Algorithmus [Lor87], um diese Aufgabe weiterhin zu erfüllen. Bei DVR wird im einfachsten Fall der Strahl von der Bildfläche zum Volumen hin solange verfolgt, bis ein Skalarwert auftritt, der größer als ein gegebener Schwellwert ist. Die Strahlverfolgung wird abgebrochen. Es folgt ein Shading-Schritt mit anschließender Rückgabe des ermittelten Farbwertes. Bei Bedarf kann der Fragmentshader den Tiefenpuffer entsprechend dem ermittelten Tiefenwert im Volumen aktualisieren. Die so erzeugten Oberflächen sind im Allgemeinen unter dem Namen *Isofläche* bekannt. Fortschrittlichere Algorithmen zum Darstellen von Isoflächen führen nach der erfolgten ersten Schnittpunktschätzung eine iterative Verfeinerung des Schnittpunktes durch [Sch05, Had09].

**MIP** Beim von [Wal89] vorgestellten MIP handelt es sich um einen der einfachsten Algorithmen für die Strahlverfolgung und der anschließenden Farbwertermittlung. MIP wird zumeist in medizinischen Anwendungen im Zusammenspiel mit MRI-Daten verwendet, um unter Zuhilfenahme von Kontrastmittel aufgenommene Arterien zu visualisieren [Mei00]. Die Farbe wird über die maximale, während der Strahlverfolgung auftretende Intensität

$$I = \max_{k=0 \dots N} (s_k) \quad (4.3)$$

mit  $s_k$  als original Skalarwert ermittelt [Had06, S. 57].

Die Richtung der Strahlverfolgung hat keine Auswirkung auf das Ergebnis. Dies ist hilfreich für eine Röntgenbild-Darstellung [Gho05]. Einer der größten Vorteile von MIP besteht darin, dass keine aufwändigen Transferfunktionen spezifiziert werden müssen um gute Visualisierungsergebnisse zu erzielen [Bru09]. Auf Transferfunktionen wird später in diesem Kapitel noch näher eingegangen. Durch den Einsatz des Maximum-Operators geht allerdings der räumliche Zusammenhang verloren. Um diesen Nachteil auszugleichen, schlägt [Mor] den Einsatz von Stereo-Rendering vor. Beleuchtung, die in der Regel die Tiefenwirkung verbessert, ist bei MIP nicht sinnvoll einsetzbar. Trotz dessen lässt das in dieser Arbeit vorgestellte Volumen-Rendering System Beleuchtung zu, da diesem System das nötige Wissen fehlt um selbstständig feststellen zu können, welche Kombination von Algorithmen üblicherweise nicht verwendet werden. Dies obliegt dem Aufgabenbereich des Anwendungsprogrammierers oder Benutzers. Eine Implementierung der Formel 4.3 ist in Quellcode 4.6 abgebildet.

Von MIP existieren in der Literatur zahlreiche Variationen. [Mor] stellt Gradient Maximum Intensity Projection (GMIP) vor, das Gradientengewichte anstatt Skalarwerte als Maximum verwendet. Um die räumliche Tiefenwahrnehmung zu verbessern und Gradienten-

```

1 uniform float Opacity;
2 vec4 RayTraversal(vec3 rs, int steps, vec3 stepDelta, float l)
3 {
4     vec3 position = rs;
5     float maxScalar = 0.0;
6     vec3 maxPosition = vec3(0.0, 0.0, 0.0);
7     for (int step=0; step<steps; step++) {
8         if (!ClipPosition(position)) {
9             float scalar = Reconstruction(position);
10            if (maxScalar < scalar) {
11                maxScalar = scalar;
12                maxPosition = position;
13            }
14        }
15        position += stepDelta;
16    }
17    vec4 color = Shading(maxScalar, maxPosition, stepDelta);
18    color.a *= Opacity;
19    return color;
20 }
```

Quellcode 4.6: MIP-Fragmentshader. Parameternamen in der Funktionssignatur aus Platzgründen gekürzt.

basierte Beleuchtung zu ermöglichen, führt [Bru09] Maximum Intensity Differences Accumulation (MIDA) ein. Hierbei handelt es sich um einen Hybrid-Algorithmus, der als eine Zwischenstufe zwischen MIP und dem im folgenden Abschnitt vorgestellten Compositing angesehen werden kann. Ein weiches Überblenden von MIP zu Compositing ist hiermit ebenfalls möglich. Während beim Compositing bei jedem Schritt entlang des Strahls Shading und anschließende Mischung der Ergebnisse ausgeführt wird, findet hingegen bei MIDA das Shading und Mischen nur statt, wenn ein neuer maximaler Skalarwert während der Strahlverfolgung gefunden wurde. Damit werden ebenfalls Strukturen, die hinter dichten, nicht transparenten Regionen liegen, besser sichtbar. Ein ähnliches Ergebnis lässt sich ebenfalls über komplexe Transferfunktionen erreichen, deren Spezifikation allerdings aufwändiger und zeitintensiver ist. Anhand Abbildung 4.6 wird ersichtlich, dass anhand der verschiedenen MIP-Variationen jeweils andere Merkmale im Volumendatensatz sichtbar werden. Dies verdeutlicht die Wichtigkeit, eine Vielzahl an Strahlverfolgungsalgorithmen zur Auswahl zu haben.

**Compositing** Während MIP lediglich den größten auftretenden Skalarwert bei der Strahlverfolgung weiterverarbeitet, lässt sich eine Gruppe von Strahlverfolgungsalgorithmen beim

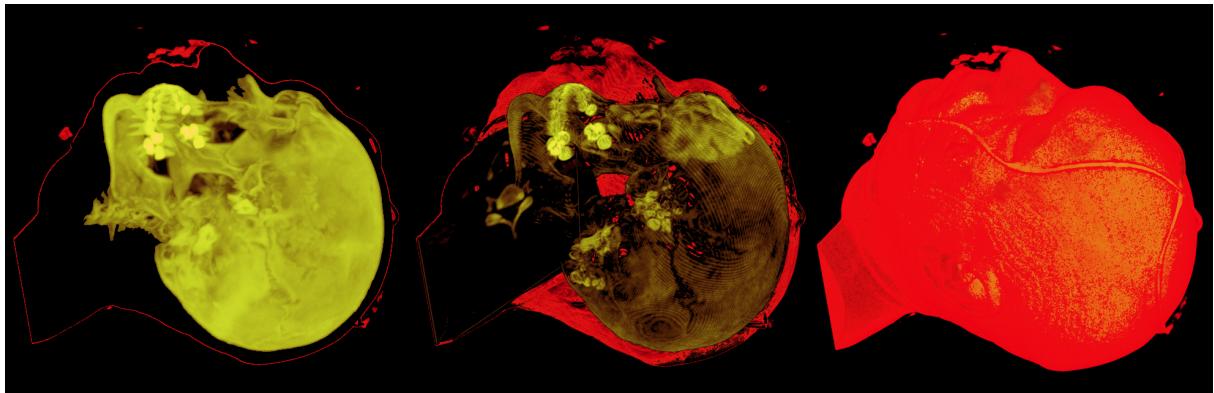


Bild 4.6: Von links nach rechts: MIP, GMIP, MIDA mit jeweils gleicher Transferfunktion. Volumendatensatz *Head (Visible Male)*.

Volumen-Rendering unter dem Überbegriff *Compositing* zusammenfassen. Das Ziel des Compositing besteht in einer iterativen Berechnung des diskretisierten Volumen-Rendering Integrals [Had06, S. 2]. Das Ergebnis der Strahlverfolgung setzt sich folglich aus einer Mischung zahlreicher Stichproben entlang des Strahls zusammen.

Eine erste Grobunterteilung der Compositing Algorithmen lässt sich anhand der Richtung der Strahlverfolgung bewerkstelligen. So beschreibt [Lev90], dass die Rendering-Algorithmen in [Dre88] und [Lev88] die Daten von hinten nach vorne verarbeiten, während die Algorithmen in [Sab88] und [Ups88] von vorne nach hinten abarbeiten.

Die Strahlverfolgung von hinten nach vorne zur Bildebene hin

$$C_{dst} = (1 - \alpha_{src})C_{dst} + \alpha_{src}C_{src} \quad (4.4)$$

wird *Back-To-Front-Compositing* genannt [Had06, S. 17]. Das vorläufige Ergebnis der Strahlverfolgung  $C_{dst}$  wird mit Null initialisiert und während der Traversierung durch das Volumen mit der aktuellen Farbwert-Stichprobe  $C_{src}$  und der aktuellen Opazität-Stichprobe  $\alpha_{src}$  iterativ aktualisiert. Es muss kein zusätzlicher Opazitätswert  $\alpha_{dst}$  bei der Strahlverfolgung gespeichert werden [Mei00, RS06]. Als es in Fragmentshadern noch keine Schleifen gab und die Anzahl der Instruktionen stark begrenzt war, stellte dies bei Rendertechniken, die in mehrere einzelne Renderschritte zerlegt wurden [Krü03] einen Vorteil da. Heutzutage lässt sich GPU-basierendes Volumen-Rendering in einem einzigen Renderschritt ausführen. Der Vorteil ist nicht mehr vorhanden da innerhalb des Fragmentshaders problemlos mit Vektoren gearbeitet werden kann, die vier Komponenten verwenden. Aus heutiger Sicht ist dieses Compositing-Schema wenig Vorteilhaft.

Die Strahlverfolgung von vorne nach hinten zum Strahlaustrittspunkt aus dem Volumen

$$\begin{aligned} C_{dst} &= C_{dst} + (1 - \alpha_{dst})\alpha_{src}C_{src} \\ \alpha_{dst} &= \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \end{aligned} \quad (4.5)$$

hingegen wird *Front-To-Back*-Compositing genannt [Had06, S. 16].

Back-To-Front- sowie Front-To-Back-Compositing benötigen eine Opazitätskorrektur. Wird dies ignoriert, so wird bei einer Änderung der Abtastrate das Ergebnis der Strahlverfolgung heller oder dunkler [Fer04, Kap. 39]. Die gespeicherte Opazität  $a_0$  für die Referenzabtastrate  $s_0$  gemäß Nyquist-Shannon-Abtasttheorem [Sha49, Wyn98, Uns00] lässt sich über

$$a = 1 - (1 - a_0)^{\frac{s_0}{s}} \quad (4.6)$$

für die aktuelle Abtastrate  $s$  korrigieren.

Beim *Front-To-Back*-Compositing stabilisiert sich die Farbe, sobald eine gewisse Opazität überschritten wird. Die Strahlverfolgung kann frühzeitig abgebrochen<sup>2</sup> werden, noch bevor der Strahl das Volumen verlassen hat [Lev90]. Bei medizinischen Datensätzen stellt ein Schwellwert von 0,95 üblicherweise einen praktisch sinnvollen Wert da. Ein geringerer Schwellwert führt zu einer Geschwindigkeitssteigerung beim Rendern, während mit einem höheren Wert hingegen die Bildfehler verringert werden können. Bei einer Opazität größer als eins wird die Strahlverfolgung üblicherweise immer abgebrochen.

## 4.4 Schnitt innerhalb des Volumens

Mit den bereits vorgestellten Schnittebenen lassen sich verschiedene Schnittgeometrien durch Kombination von mehreren Schnittebenen annähern. Viele hilfreiche und wichtige Geometrien sind durch diese Lösung jedoch nicht realisierbar [Wei03]. Ein bekanntes Beispiel ist der Volumenschnitt anhand einer Würfel- oder Kugelgeometrie. Diese einfache und effektive Technik ist in annähernd jedem kommerziellen System integriert und bietet grundlegende Interaktionsfunktionalität [Ras08]. Im Gegensatz zur bereits im Abschnitt 4.1 beschriebenen Veränderung der Strahlstartposition und Strahllänge, die als positiven Seiteneffekt im Allgemeinen die Leistungsfähigkeit verbessert, kommt es bei der in diesem Abschnitt vorgestellten Art des Schnittes potentiell zu einer Segmentierung des Strahls. Aufgrund dessen sind die einzelnen Stichproben entlang des Strahls individuell zu prüfen. Dies hat zur Folge, dass der hier vorgestellte Schnitt die Leistungsfähigkeit im Allgemeinen verschlechtert. Prinzipiell kann die im folgenden be-

---

<sup>2</sup>Engl.: *Early-Ray-Termination*

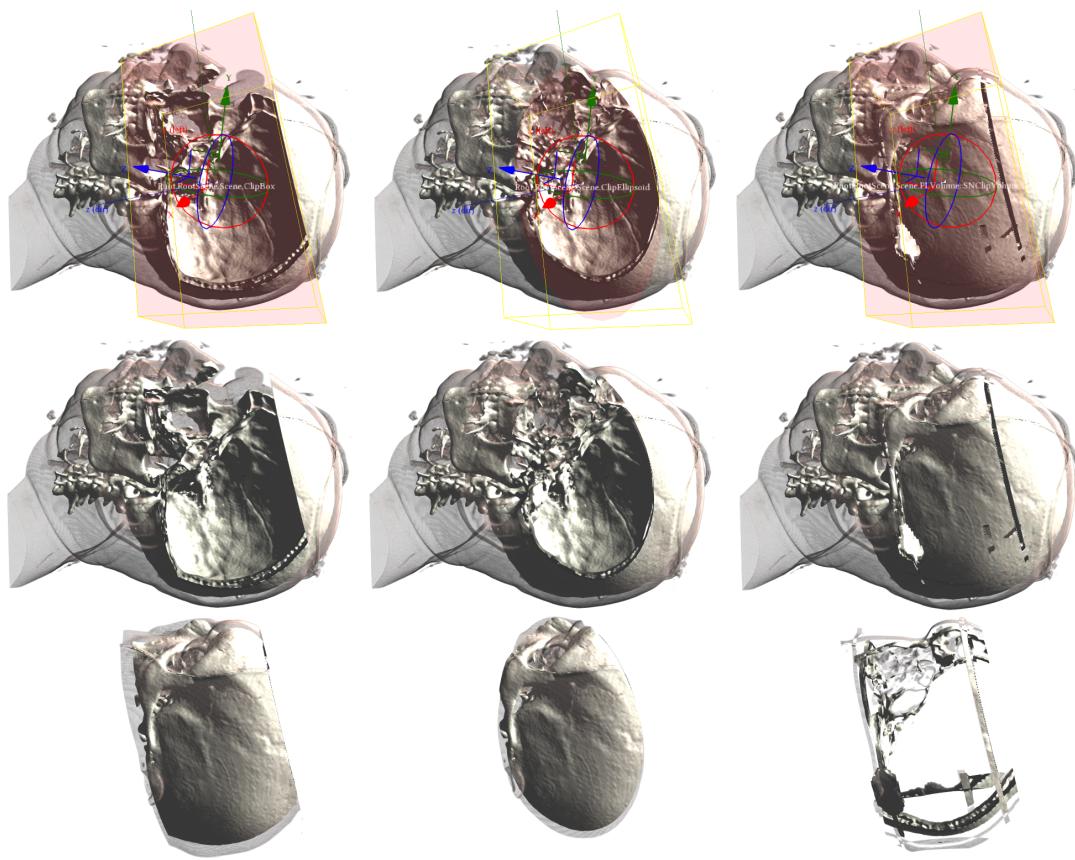


Bild 4.7: Von links nach rechts: Schnittwürfel, Schnittellipsoid und Schnittvolumen. Untere Reihe mit Invertierung des booleschen Schnittergebnisses. Volumendatensatz *Head (Visible Male)* und Volumendatensatz *Crossed Rods* als Schnittvolumen.

schriebene Shaderfunktion pro Stichprobe entlang des Strahls mehrmals aufgerufen werden um somit eine Kombination mehrerer Schnittgeometrien zu ermöglichen.

**Funktionssignatur und Ablauf** Die in Quellcode 4.7 abgebildete Shaderfunktion erhält als Eingabe eine Position  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  im Volumen. Der boolesche Rückgabewert

```
1 bool ClipPosition(vec3 Position);
```

Quellcode 4.7: Funktionssignatur für einen Schnitt innerhalb des Volumens.

ist wahr falls die gegebene Position im Volumen geschnitten wurde.

Im folgenden sollen die drei in Abbildung 4.7 vorgestellten Schnittgeometrien beleuchtet werden. Zahlreiche weitere Möglichkeiten, wie weitere Geometrietypen, beispielsweise Zylinder, bis hin zu komplexen mathematischen Funktionen sind denkbar.

**Schnittwürfel** Die Schnittgeometrie, hier ein Würfel, kann frei im Raum positioniert, rotiert und skaliert werden. Ein Strecken der Schnittgeometrie ist möglich, indem Koordinatenachsen der Schnittgeometrie unterschiedlich skaliert werden. Das lokale Koordinatensystem der Schnittgeometrie hat eine Ausdehnung von eins. Der Koordinatenursprung befindet sich im Mittelpunkt dieses Einheitskoordinatensystems. Dies ermöglicht eine einfachere Ausrichtung beim rotieren. Durch diese Definition ist eine  $4 \times 4$  Transformationsmatrix ausreichend, weitere Parameter wie beispielsweise die Höhe des Würfels werden nicht benötigt da diese Informationen bereits in der Skalierung enthalten sind. Dieses Prinzip gilt für alle weiteren in diesem Abschnitt beschriebenen Schnittgeometrien.

Um festzustellen, ob sich die aktuelle Position entlang des Strahls im Volumen innerhalb des so beschriebenen Schnittwürfels befindet, reicht eine Transformation dieser Position vom Volumenkoordinatensystem in das Einheitskoordinatensystem der Schnittgeometrie. Es folgt ein größer/kleiner Vergleich der einzelnen Positionskomponenten. Befindet sich die so überprüfte Position innerhalb des Einheitswürfels, so liefert die Shaderfunktion ein wahr zurück und die aktuelle Position entlang des Strahls im Volumen wird vom Aufrufer nicht weiterverarbeitet. Der Würfel schneidet ein Loch in das Volumen. Durch eine Invertierung des booleschen Schnittergebnisses werden alle Bestandteile des Volumen ausgeblendet, die sich nicht innerhalb des Würfels befinden. Hierüber könnte ein Wegschweben des herausgeschnittenen Volumenstückes realisiert werden. Dazu müsste bei einer entsprechenden Visualisierung das Volumen doppelt gezeichnet werden, einmal ohne und einmal mit invertiertem Volumenschnitt.

**Schnittellipsoid** Eine Kugel ist ein Sonderfall eines Ellipsoiden, bei der alle Koordinatenachsen die gleiche Skalierung besitzen. Genauso wie beim Schnittwürfel wird die zu testende Position entlang des Strahls im Volumen in das Einheitskoordinatensystem der Schnittgeometrie umgerechnet. Die Schnittoperation ist folglich eine einfache Mittelpunktabstands-Rechnung, unabhängig davon ob die Kugel für den Benutzer durch eine unterschiedliche Achsenkalierung gestreckt aussieht. Eine Umsetzung ist in Quellcode 4.8 zu finden. Um den Umfang der Implementierung im Rahmen zu halten, wird die Invertierung des booleschen Schnittergebnisses in diesem Beispiel über einen Shaderparameter gesteuert. Eine effizientere Realisierung bestünde in der Bereitstellung einer alternativen Implementierung der Shaderfunktion mit vertauschtem Vergleichsoperator.

**Schnittvolumen** Der von [Wei03] für Slicing-basierendes Volumen-Rendering vorgestellte Schnitt anhand individueller Voxel lässt sich auf Volumen-Raycasting übertragen. Das Schnittvolumen wird auf der Grafikkarte in einer zusätzlichen Textur gespeichert. Prinzipiell reicht hierfür ein boolescher Wert pro Voxel. Bei einer praktischen Realisierung muss mangels ei-

```

1 uniform mat4 VolumeToEllipsoid;
2 uniform bool InvertClipping;
3 bool ClipPosition(vec3 Position)
4 {
5     vec3 position = (VolumeToEllipsoid*vec4(Position, 1.0)).xyz;
6     bool clipPosition = (length(position) < 0.5);
7     return InvertClipping ? !clipPosition : clipPosition;
8 }
```

Quellcode 4.8: Schnittellipsoid Fragments shader.

nes booleschen Texturformates allerdings auf ein 8 bit Texturformat ausgewichen werden. Als Ausgangsbasis für die zu realisierende Umsetzung der Shaderfunktion kann die Implementierung des Schnittwürfels verwendet werden. Da Volumenkoordinaten den Ursprung im Nullpunkt haben, muss nach der Umrechnung der übergebenen Position ein entsprechender Offset hinzugefügt werden. Für den eigentlichen Schnitt wird lediglich ein Texturzugriff sowie ein Schwellwertvergleich benötigt.

Prinzipbedingt ermöglicht es diese Schnittvariante beliebige Schnittformen zu verwenden. Die Auflösung des Schnittvolumens ist unabhängig von der Auflösung des Volumendatensatzes und der Inhalt dieser zusätzlichen Textur kann dynamisch zur Laufzeit verändert werden. Hierdurch kann dem Benutzer ein wegradieren bestimmter Volumenbestandteile ermöglicht werden. Dieser Vorgang ist als *Punching* bekannt und lässt sich im wesentlichen in den Aufgabenbereich der grafischen Benutzeroberfläche einordnen. Aufgrund dessen soll an dieser Stelle nicht weiter auf diese Thematik eingegangen werden.

## 4.5 Rekonstruktion der Volumendaten

Bis zum jetzigen Zeitpunkt wurde primär der Strahl durch das Volumen betrachtet, die eigentlichen Volumendaten fanden bisher noch keine tiefergehende Beachtung.

**Funktionssignatur und Ablauf** Aufgrund der unterschiedlichen zu erfüllenden Anforderungen wurde diese Shaderfunktion in die drei in Quellcode 4.9 abgebildeten unterschiedlichen Funktionen zerlegt. Für das Volumen-Sampling existieren zwei Shaderfunktionen, für Rekonstruktion eine.

Während der Strahlverfolgung wird in der Regel die *Reconstruction*-Shaderfunktion aufgerufen um den Skalarwert  $[0, 1]$  an der Position  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  im Volumen zu ermitteln.

```

1 float Reconstruction(vec3 Position);
2 float FetchScalar(vec3 Position);
3 float FetchScalarOffset(vec3 Position, ivec3 Offset);

```

Quellcode 4.9: Funktionssignaturen für die Rekonstruktion der Volumendaten.

Die einfachste Implementierung dieser Funktion ruft direkt die *FetchScalar*-Funktion auf um den Skalarwert [0, 1] an der Position  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  im Volumen von der GPU zu erfragen.

Von dieser Shaderfunktion existiert ebenfalls die Variante *FetchScalarOffset* um den Skalarwert [0, 1] an der Position  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  sowie einem zusätzlichen Offset  $[(0 \ 0 \ 0)^T, (b - 1 \ h - 1 \ t - 1)^T]$ <sup>3</sup> im Volumen von der GPU zu erfragen. Um Daten in der Nachbarschaft eines Voxel zu ermitteln, wie es beispielsweise bei der Gradientenschätzung benötigt wird, würde die *FetchScalar*-Shaderfunktion ausreichen. Moderne Shadersprachen bieten eine Vielzahl an Möglichkeiten um Texturdaten von der GPU zu erfragen. Um Grafikkartentreibern Spielraum für Optimierungen zu geben, sollten entsprechende Funktionalitäten soweit sinnvoll genutzt werden.

**Rekonstruktionsfilter** Während der Strahlverfolgung werden an beliebigen Stellen die in einer Textur gespeicherten diskretisierten Volumendaten abgetastet. Die Abtastpunkte liegen hierbei selten auf den tatsächlichen Rasterpositionen und benötigen daher eine auf der Nachbarschaft basierende Interpolation [Mei00]. Die Art der Interpolation und die Gewichtung der Nachbartexel wird über sogenannte Rekonstruktionsfilter beschrieben.

Grafikkarten unterstützen nächster-Nachbar-Filterung<sup>4</sup>, lineare Filterung für 1-D-Texturen, bilineare Filterung für 2-D-Texturen sowie trilineare Filterung für 3-D-Texturen. Die nächster-Nachbar-Filterung kann im Grunde nicht als Filterung bezeichnet werden, es handelt sich vielmehr um die Auswahl des Texels der dem angefragten Punkt am nächsten liegt [Mei00]. Aufgrund der sichtbaren Blöcke ist die Bildqualität entsprechend niedrig. In der Regel wird trilineare Filterung für ein akzeptables Verhältnis zwischen Bildqualität und Bildwiederholrate verwendet. Diese hardwareseitige Unterstützung für Texturfilter stellt einen der Pluspunkte eines GPU-basierenden Volumen-Renderers dar. Ausführlichere Hintergründe über die erwähnten Rekonstruktionsfilter sind [Had06, S. 21] zu entnehmen.

Eine Implementierung der *FetchScalar*-Shaderfunktion für 3-D-Texturen ist in Quellcode 4.10 abgebildet. Hierbei handelt es sich um die einfachste Umsetzung dieser Shaderfunktion,

---

<sup>3</sup>b=Breite der Volumentextur, h=Höhe der Volumentextur und t=Tiefe der Volumentextur

<sup>4</sup>Engl. *nearest neighbor filter*, im Englischen ebenfalls unter *point sampling* bekannt

```

1 uniform sampler3D VolumeTexture;
2 float FetchScalar(vec3 Position)
3 {
4     return textureLod(VolumeTexture, Position, 0.0).r;
5 }
```

Quellcode 4.10: Fragmentsshader Implementierung für die *FetchScalar*-Funktion.

die Grafikkarte übernimmt die Hauptarbeit. Beim Einsatz eines 2-D-Texturfeldes<sup>5</sup> ist zu berücksichtigen, dass die GPU lediglich eine bilineare Filterung ausführt. Eine Filterung zwischen den Texturebenen muss im Fragments shader realisiert werden. Gleiches gilt beim Einsatz von 2-D-Texturen zum Speichern der Volumendaten, zusätzlich muss die gegebene 3-D-Position auf eine 2-D-Position innerhalb der 2-D-Textur übertragen werden.

**Qualitativ hochwertiger Rekonstruktionsfilter** Während trilineare Filterung dank direkter Hardwareunterstützung für eine interaktive Darstellung geeignet ist, kann eine bessere Qualität durch die Verwendung von Interpolationsmethoden höherer Ordnung erzielt werden. Ein qualitativ hochwertiger Rekonstruktionsfilter unter der Zuhilfenahme von kubischer Faltung oder B-Spline-Interpolation benötigt bereits bei einem kleinen Kernel von  $4 \times 4 \times 4$  insgesamt 64 Texturzugriffe [Mei00] pro rekonstruierten Skalarwert.

[Pha05, Kap. 20] stellt einen qualitativ hochwertigen Rekonstruktionsfilter über kubische B-Splines vor, der sich die trilineare Filterung der GPU zunutze macht. Der tricubische Filter mit 64 Summanden lässt sich hierdurch mit lediglich acht Texturzugriffen ermitteln. [Lee10a] verwendet zusätzlich virtuelle Samples um die Bildqualität weiter zu erhöhen. In Abbildung 4.8 sind die Ergebnisse von drei unterschiedlich aufwändigen Rekonstruktionsfiltern gegenübergestellt.

Der Implementierungsaufwand sowie der negative Einfluss auf die Bildwiederholrate sind nicht zu vernachlässigen, während die erzielte Bildverbesserung erst bei einem genaueren Vergleich ersichtlich wird. Aufgrund dessen sei für weiterführende Informationen über die Theorie auf [Had06, S. 223] verwiesen.

**Prozedurale Volumendaten** Die jeweiligen konkreten Implementierungen der Shaderfunktionen müssen lediglich die Schnittstellenspezifikation erfüllen. Bisher wurde in diesem Abschnitt davon ausgegangen, dass die Volumendaten als Textur im Grafikkartenspeicher vorliegen. Während dies bei medizinischen Anwendungsbereichen in der Regel der Fall ist, schließt das in dieser Arbeit vorgestellte Volumen-Rendering System nicht aus, dass die Daten bei der

---

<sup>5</sup>Engl. 2-D-Texture Array

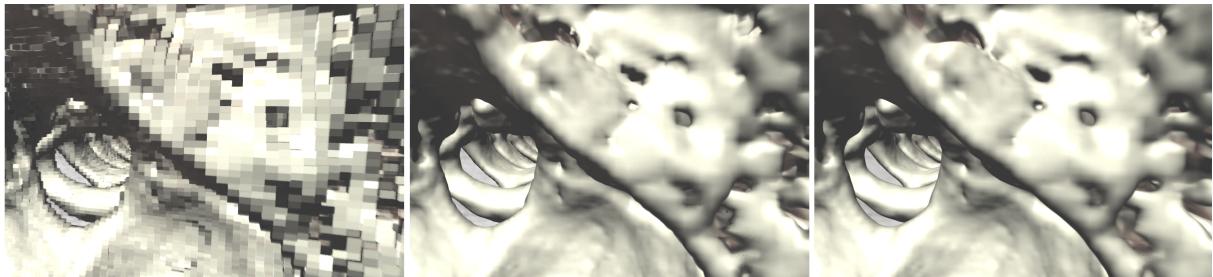


Bild 4.8: Drei Rekonstruktionsfilter im Vergleich, von links nach rechts: Nächster-Nachbar, Trilinear, kubische B-Splines. Aktiviertes künstliches Rauschen, Gradientenschätzung jeweils über zentrale Differenzen und Cook-Torrance-Beleuchtungsmodell. Volumendatensatz *Head (Visible Male)*.

Anfrage des Skalarwertes an einer bestimmten Position berechnet anstatt ausgelesen werden. Die vorgestellten qualitativ hochwertigen Rekonstruktionsfilter sind konzeptbedingt nicht fest mit den sich im Grafikkartenspeicher befindlichen Volumendaten verbunden und können somit ebenfalls auf prozedurale Daten angewandt werden. Jedoch haben über mathematische Funktionen zur Laufzeit berechnete Volumendaten den Vorteil, dass keine Signalrekonstruktion notwendig ist, da sich in der Regel beliebig feine Zwischenstufen in Funktionen errechnen lassen.

Sofern keine medizinischen Anforderungen wie beispielsweise Datentreue einzuhalten sind, könnte eine weitere Bildverbesserungsmaßnahme darin bestehen, diskretisierte Volumendaten mit prozeduralen Daten zu mischen. So könnten feingranulare Details in der Visualisierung vorgetäuscht werden. Diese sind im rekonstruierten Signal mangels eingeschränkter Auflösung nicht vorhanden.

## 4.6 Shading

Das Shading hat zum Ziel, aus einer Menge gegebener Parameter eine für den Menschen interpretierbare Farbe zu ermitteln. Hierfür existieren zahlreiche Ansätze mit jeweils unterschiedlichen Zielen. Eines dieser Ziele könnte darin bestehen, die Tiefenwahrnehmung zu verbessern indem Lichtquellen und Schatten miteinbezogen werden. Für die Beleuchtung wird beim Volumen-Rendering traditionell Gradienten-basierendes Shading verwendet. Dieses Verfahren versucht anhand von Schätzungen Gradienten zu ermitteln, die im Beleuchtungsschritt als Normalenvektor verwendet werden. Je nach eingesetztem Verfahren schwankt die Bildqualität sowie die erzielbare Bildwiederholrate. Aufgrund dessen wurden in den vergangenen Jahren weitere Shading-Modelle entwickelt, die ohne Gradientenschätzung auskommen [Šo10, Rop10b] sowie Hybrid-Modelle die klassische Beleuchtungsmodelle und Volumetrisches-Scattering mischen [Kro11]. Verglichen mit Gradienten-basierten Verfahren können diese modernen Mo-

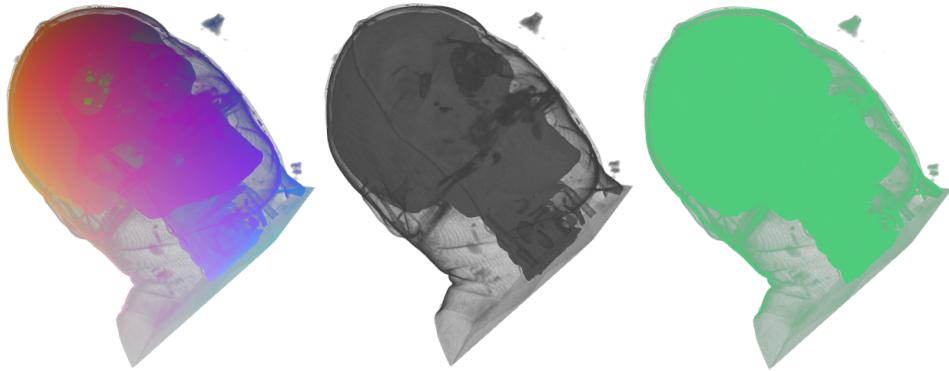


Bild 4.9: Shading Eingabe von links nach rechts: Position im Volumen, Skalarwert an Position, Blickrichtung. Volumendatensatz *Head (Visible Male)*.

delle eine überlegene Bildqualität sowie höhere Bildwiederholraten erzielen und gleichzeitig die Tiefenwahrnehmung verbessern [Rop10a]. Neben der Strahlverfolgung hat Shading den größten Einfluss auf das resultierende Bild. Aufgrund des Umfangs und der Komplexität der Shading-Thematik kann an dieser Stelle jedoch nur ein kurzer Überblick vorgestellt werden.

**Funktionssignatur und Ablauf** Die Shaderfunktion in Quellcode 4.11 erhält als Eingabe den aktuellen Skalarwert  $[0, 1]$  an der als zweiten Parameter übergebenen Position  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  im Volumen. Um eine Vielzahl an Implementierungsmöglichkeiten

```
vec4 Shading(float Scalar, vec3 Position, vec3 StepPositionDelta);
```

Quellcode 4.11: Funktionssignatur für Shading.

abzudecken, wird ebenfalls die Schrittänge  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  mitgeliefert. Wie anhand Abbildung 4.9 zu erkennen ist, steht es Implementierungen frei weitere Parameter, wie in diesem Fall die Blickrichtung, als gewöhnliche Shadervariablen zu verwenden. Im Rahmen des Shading werden von der gewählten Umsetzung in der Regel die folgenden Shaderfunktionen aufgerufen:

1. Aufruf der Shaderfunktion 4.7 für die Klassifikation des Skalarwertes
2. Aufruf der Shaderfunktion 4.8 für die Gradientenschätzung
3. Aufruf der Shaderfunktion 4.9 für die lokale Beleuchtung

**Realisierung** Shading lässt sich in globale, direkte und lokale Methoden untergliedern [Mei00]. Globale Methoden berücksichtigen den Lichtaustausch aller Objekte während direkte Methoden nur auf einem einzigen Objekt arbeiten. Beide Methoden sind im Feld der Echtzeitgrafik aktive und umfangreiche Forschungsgebiete, daher wird im Rahmen dieser Arbeit nur auf traditionelle lokale Methoden eingegangen.

Lokale Methoden nutzen für die Beleuchtungsberechnungen lediglich Informationen aus der unmittelbaren Umgebung. Das Beleuchtungsmodell wird in der Regel in die drei Bestandteile *Ambient*, *Diffuse* und *Glanzlicht* unterteilt, für jeden Bestandteil existieren unterschiedliche Ansätze. Ambient besteht im einfachsten Fall aus einer gegeben Farbe, die eine gleichförmige Grundbeleuchtung repräsentiert. Fortschrittlichere Verfahren nutzen eine Annäherung von Umgebungsverdeckung, um Strukturen besser erkennbar zu machen und die Tiefenwahrnehmung zu verbessern [Her10]. Im Gegensatz zu Ambient sind die Diffuse und Glanzlicht Anteile abhängig von den eingesetzten Lichtquellen. Das Glanzlicht ist zusätzlich Blickwinkelabhängig. Zur Vereinfachung wird an dieser Stelle nur eine einzige direktionale Lichtquelle berücksichtigt. Prinzipiell lassen sich ebenfalls mehrere Lichtquellen und Lichttypen realisieren. Aufgrund dessen wurden die lichtabhängigen Berechnungen in eine eigenständige Shaderfunktion ausgelagert, auf die am Ende dieses Kapitels in Abschnitt 4.9 eingegangen wird.

Ein Fragments shader für Shading mit lokaler Beleuchtung ist in Quellcode 4.12 zu finden. Volumendaten bestehen oftmals aus homogenen Flächen, in denen der Gradient undefiniert ist

```

1 uniform vec3 ViewingDir;
2 uniform vec3 LightDir;
3 uniform vec3 AmbientColor;
4 vec3 pn = vec3(0.0, 0.0, 0.0);
5 vec4 Shading(float Scalar, vec3 Position, vec3 StepDelta)
6 {
7     vec4 color = Classification(Scalar);
8     vec3 gradient = Gradient(Position);
9     float gradientLen = length(gradient);
10    vec3 n = (gradientLen > 0.0) ? gradient/gradientLen : pn;
11    pn = n;
12    color.rgb = AmbientColor + Illumination(color.rgb, n,
13        ViewingDir, LightDir);
14    return color;
}

```

Quellcode 4.12: Fragments shader für Shading mit lokaler Beleuchtung. Parameternamen in der Funktionssignatur aus Platzgründen gekürzt.

[Kra06]. Um eine Division durch Null zu verhindern, muss eine entsprechende Sicherheitsabfrage eingesetzt werden. Greift diese, wird der letzte gültige Gradient weiterverwendet.

Die Simulation der Realität ist nicht das einzige Ziel, das bei einer Shading-Strategie verfolgt werden kann. So stellt [Had06] ebenfalls eine Vielzahl weiterer Shading Verfahren vor, die eine nicht photorealistische Darstellung zum Ziel haben.

## 4.7 Klassifikation

Beim Volumen-Rendering sind 12 bit oder 8 bit pro Voxel üblich. Damit ergeben sich  $2^{12} = 4096$  beziehungsweise  $2^8 = 256$  darstellbare unterschiedliche Skalarwerte. Diese repräsentieren beispielsweise skalare Dichtewerte. Werden diese Daten ohne weitere Verarbeitung direkt zur Darstellung verwendet, so ergibt sich ein Graustufenbild. Die je nach Anwendungsfall unterschiedlichen relevanten Merkmale im Volumendatensatz sind bei einer derartigen Visualisierung schwer bis nicht zu erkennen. Das Ziel der Klassifikation besteht darin, die vorhandenen Skalarwerte derart zu interpretieren, dass die relevanten Merkmale deutlich erkennbar sind.

In den vergangenen Abschnitten wurden bereits zahlreiche Verfahren für den Volumenschnitt vorgestellt. Nach [Wei03] lässt sich die Klassifikation ebenfalls in den Aufgabenbereich des Volumenschnitt einordnen. Klassifikation ermöglicht es dem Benutzer, Strukturen in den Volumendaten zu finden, ohne dabei ausdrücklich die Form und die Ausmaße dieser Struktur zu definieren [Mei00]. Für die Strahlverfolgung MIP und dessen Variationen ist eine qualitativ hochwertige Klassifikation weniger ausschlaggebend, für Compositing hingegen ist die Klassifikation ein entscheidender Faktor [Bru09].

**Funktionssignatur und Ablauf** Die Shaderfunktion in Quellcode 4.13 erhält als Eingabe einen zu klassifizierenden normalisierten Skalarwert  $[0, 1]$  und liefert als Ergebnis einen Farbwert mit vier Komponenten zurück. Weitere Klassifikationsergebnisse wie beispielsweise Re-

```
vec4 Classification(float Scalar);
```

Quellcode 4.13: Funktionssignatur für die Klassifikation.

flektionseigenschaften lassen sich prinzipiell nach dem gleichen Schema hinzufügen [Mei00].

**Schwellwert** Im einfachsten Fall liefert die Klassifikation den gegebenen Skalarwert ohne weitere Verarbeitung zurück. Es findet keine Klassifikation statt. Die zweit einfachste Möglichkeit besteht in der Nutzung eines Schwellwertes. Der übergebene Skalarwert wird nur zurückgeliefert, wenn er größer als der Schwellwert ist, ansonsten wird ein Nullvektor zurückgegeben.

Dies ist vergleichbar zur Strahlverfolgung zum Erzeugen von Isoflächen, mit dem Unterschied, dass die Strahlverfolgung bei Auffinden eines gegebenen Schwellwertes nicht abgebrochen wird. Knochen besitzen eine höhere Dichte als umliegendes Gewebe, über einen entsprechend gewählten Schwellwert lässt sich das Gewebe ausblenden. Nach [Sab88] war im Jahre 1988 das Volumen-Rendering mit Hilfe eines Schwellwertes in medizinischen Anwendungen gängig. Diese arbeiten mit von CT-Scannern aufgezeichneten 3-D-Dichte Bildern.

**Transferfunktion** Für eine anspruchsvollere Klassifikation werden sogenannte Transferfunktionen eingesetzt. Klassischerweise bildet eine 1-D-Transferfunktion einen Skalarwert  $s$ , der beispielsweise eine Dichte repräsentiert, auf optische Eigenschaften ab. Die Emission wird üblicherweise über einen Farbwert mit den Komponenten Rot  $R$ , Grün  $G$  und Blau  $B$  über

$$T_c(s) = \begin{pmatrix} R \\ G \\ B \end{pmatrix}, s \in [0, 2^x - 1], 0 \leq R, G, B \leq 1 \quad (4.7)$$

mit der Anzahl  $x$  an unterschiedlich darstellbaren Skalarwerten, abgebildet. Absorption hingegen wird über einen Opazitätswert

$$T_a(s) = \alpha, s \in [0, 2^x - 1], 0 \leq \alpha \leq 1 \quad (4.8)$$

abgebildet. Die drei Farbkomponenten in  $T_c(s)$  und der Opazitätswert  $T_a(s)$  lassen sich der Einfachheit halber zu einer gemeinsamen Transferfunktion  $T_f(s)$ , mit vier voneinander unabhängigen Komponenten, verschmelzen. Vereinfacht lässt sich eine solche 1-D-Transferfunktion als eine Tabelle darstellen, der Skalarwert wird als Index auf einen Tabelleneintrag verwendet. In der Regel sind die Skalarwerte aufgrund der Rekonstruktion nicht diskret, sondern kontinuierlich. Einträge in der 1-D-Transferfunktion müssen folglich interpoliert werden. In der Praxis stellt dies kein Problem dar wenn diese Transferfunktion in einer 1-D-Textur gespeichert wird. Die Grafikkarte übernimmt bei entsprechenden Einstellungen automatisch die lineare Interpolation. Abbildung 4.10 stellt ein visualisiertes Volumen ohne und mit Klassifikation gegenüber. Ein Fragmentshader für Klassifikation über eine 1-D-Transferfunktion ist in Quellcode 4.14 abgebildet.

Bei dem beschriebenen Vorgehen handelt es sich um eine sogenannte Post-Interpolative Transferfunktion. Die Klassifikation erfolgt auf einem bereits rekonstruierten Skalarwert. Für GPU-basierendes Volumen-Rendering bietet sich dieses Vorgehen technisch bedingt an. Ebenfalls lässt sich damit eine gute Bildqualität erzielen [Had06, S. 90]. Alternativ besteht die Möglichkeit, die Skalarwerte im Volumen zuerst zu klassifizieren, und erst anschließend die

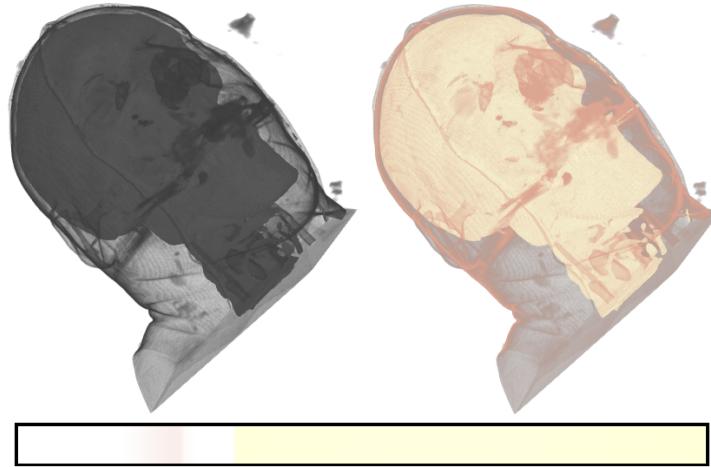


Bild 4.10: Links Skalarwerte des Datensatzes, Ergebnis der Klassifizierung rechts. Verwendete 1-D-Transferfunktion unten. Volumendatensatz *Head (Visible Male)*.

```

1 uniform sampler1D TransferFunctionTexture;
2 vec4 Classification(float Scalar)
3 {
4     return textureLod(TransferFunctionTexture, Scalar, 0.0);
5 }
```

Quellcode 4.14: Fragmentshader für die Klassifikation über eine 1-D-Transferfunktion.

Rekonstruktion auf diesen klassifizierten Daten auszuführen. Dies ist jedoch für eine GPU-basierende Realisierung weniger vorteilhaft. Entweder müssen die im Grafikkartenspeicher abgelegten Volumendaten bei jeder Änderung der Transferfunktion aktualisiert werden, was bedeutet das ein Volumen nicht effizient über verschiedene Transferfunktionen gleichzeitig dargestellt werden kann, oder es wird auf die trilineare Filterung der GPU verzichtet. Im letzteren Fall wird die Filterung für die Rekonstruktion komplett über einen eigenen Fragmentshader ausgeführt.

Bei der Transferfunktion ist zu beachten, dass es unter Umständen zu einer unerwünschten Farbmischung<sup>6</sup> kommt. Hat ein Eintrag eine Opazität von null, während ein Nachbareintrag hingegen eine Opazität von eins besitzt, so ergibt eine Abtastung zwischen beiden Einträgen eine Farbe, die zwischen den Farbwerten beider Einträge in der Transferfunktion liegt. Ungeachtet dessen, dass der Eintrag mit der Opazität von null nicht sichtbar ist, fließt dessen Farbwert in die Interpolation mit ein. Werden die Volumendaten vor der Rekonstruktion klassifiziert, so tritt dieser Effekt umso deutlicher ein [Wit98]. Um dieses Verhalten der Farbmischung zu beseitigen, muss der Farbwert in der Transferfunktion mit dessen Opazität multipliziert werden

---

<sup>6</sup>Engl. *Color Bleeding*

[Had06, S. 55]. Die technische Umsetzung wird aufwändiger, in durchgeführten Stichproben war keine optische Veränderung mit dem bloßen Auge erkennbar. Bei Volumendaten mit geringer Auflösung oder einer Transferfunktion mit abrupten Änderungen zwischen zwei Werten, beispielsweise von Gewebe zu Knochen hingegen ist es anzunehmen, dass die unerwünschte Farbmischung ersichtlich wird. Es gilt daher von Fall zu Fall abzuwägen, ob der, wenn auch geringe, zusätzliche Aufwand einen optischen Mehrwert bietet.

Bilddaten, wie beispielsweise Farbfotografien, liegen klassischerweise im nicht linearen Gamma-Farbraum vor. Gleiches gilt in der Regel für Texturdaten. Bis vor wenigen Jahren wurde traditionell direkt in den Bildpuffer gezeichnet, der anschließend auf einem handelsüblichen Bildschirm mit nicht linearen Gamma-Farbraum angezeigt wird. Eine Farbkomponente kann hierbei Festkommazahlen im Bereich [0, 255] annehmen. Es bestand daher keine unmittelbare Notwendigkeit einer Farbraumumrechnung beim Rendern von Bilddaten. Beim Aufstellen der Transferfunktionen wird das Thema der Gammakorrektur in der Regel in der Literatur nicht angesprochen. Moderne Grafikpipelines arbeiten mit HDR. Dies bedeutet, dass in einen Fließkommazahlenpuffer gezeichnet wird und anschließend ein Tone-Mapping Operator angewandt wird um Farbwerte zu erhalten, die von handelsüblichen Bildschirmen dargestellt werden können. Hierfür kann beispielsweise der Reinhard Tone-Mapping Operator [Rei02] verwendet werden. Diese Grafikpipelines arbeiten in einem linearen Farbraum, der unter anderem ein korrektes mischen von Farben erlaubt. Texturdaten, die im nicht linearen Gamma-Farbraum  $C_g$  vorliegen müssen aufgrund dessen über die Gammakorrektur

$$C_l = C_g^{2,2} \quad (4.9)$$

in den linearen Farbraum  $C_l$  umgerechnet werden [Ngu07, Kap. 24]. Gleiches gilt demzufolge auch für Transferfunktionen, nicht jedoch für die eigentlichen skalaren Volumendaten. Die Gammakorrektur lässt sich direkt in die Transferfunktion einrechnen, dies bedeutet jedoch, dass diese anschließend in einem Fließkommazahlen Format vorliegen muss. Zusammen mit Tone-Mapping wird dieser lineare Farbraum am Ende der Grafikpipeline in den nicht linearen Gamma-Farbraum, der von Bildschirmen verwendet wird, umgerechnet. Bei HDR-Bildschirmen ist kein Tone-Mapping am Ende der Grafikpipeline nötig [Gho05].

Die in Abschnitt 4.3 erwähnte Opazitätskorrektur [Fer04, Kap. 39] lässt sich direkt in die Transferfunktion einarbeiten. Hierdurch lassen sich im Fragmentshader bei der Strahlverfolgung einige Instruktionen pro Schritt entlang des Strahls einsparen. Es ist jedoch zu beachten, dass diese so veränderte Transferfunktion nicht mehr für beispielsweise MIP verwendet werden kann. Ebenfalls muss die Transferfunktion durch diese Änderung nun in einem für die GPU weniger effizient verarbeitbaren Fließkommaformat vorliegen. Im Zusammenspiel mit

weiteren möglichen Veränderungen einer Transferfunktion, wie einer Gammakorrektur, kann dies unüberschaubar und dadurch Fehleranfällig werden. Um die Anzahl der zu verwaltenden Transferfunktion- und Shader-Variationen nicht außer Kontrolle geraten zu lassen, wurde in der im Rahmen dieser Arbeit erstellten Volumen-Renderer Realisierung auf diese Optimierungsmöglichkeit verzichtet.

**Komplexe Klassifikation** Für die Verbesserung der Bildqualität schlägt [Eng01] Pre-Integration vor. Über eine veränderte Transferfunktion lässt sich die Abtastrate verringern, was im Allgemeinen in einer höheren Bildwiederholrate resultiert. Weitere Details lassen sich in [Had06, S. 92] nachschlagen. Pre-Integration in Kombination mit weiteren Volumen-Rendering Techniken zieht jedoch einige Schwierigkeiten nach sich. So kommt es im Zusammenspiel mit Shading [Mei02] und Volumenschnitt [Roe03] zu Problemen. Aufgrund dessen soll an dieser Stelle nicht weiter auf diese Optimierungsmöglichkeit eingegangen werden, da sich hierdurch zu viele weitere Fallstricke eröffnen.

Übergänge zwischen Materialien, wie beispielsweise von Gewebe zu Knochen, lassen sich mit 1-D-Transferfunktion nur schwer modellieren [Kni01]. Derartige Materialübergänge sind mit 2-D-Transferfunktionen besser abbildbar, neben dem Skalarwert wird hierbei ebenfalls die Gradientengewichtung als Index der Transferfunktion verwendet. Mehrdimensionale Transferfunktionen eignen sich unter anderem zur Visualisierung von Blutgefäßen in CT-Datensätzen [Kub12] oder zu Illustrationszwecken [Sva05]. Ein kompakter Überblick über weitere relevante Literatur zum Thema der Klassifikation lässt sich in [Bru09] finden.

**Aufstellen einer Transferfunktion** Nach [Pre00] ist eine intuitive und zugleich präzise Einstellung der Transferfunktion die wichtigste Interaktion beim Volumen-Rendering. Dies bedeutet zum einen, dass bei einer Änderung die Auswirkung spätestens nach wenigen Sekunden sichtbar sein sollte, zum anderen muss es das Graphical User Interface (GUI) dem Benutzer ermöglichen, die Änderung interaktiv und ohne viel Aufwand vorzunehmen. [Kin02] und [Had06, S. 268] geben eine kurze Übersicht über klassische Herangehensweisen beim Aufstellen von Transferfunktionen und dem Design entsprechender Benutzerschnittstellen. Eine gängige Herangehensweise besteht darin, dem Benutzer außerhalb der 3-D-Ansicht in einem Diagramm verschiedene Kurven ändern zu lassen. In einem CT-Datensatz repräsentieren Skalarwerte Dichten. Um beispielsweise die Struktur von Knochen sichtbar zu machen, muss dem Benutzer bewusst sein, dass bei dieser Modalität Knochen höhere Skalarwerte besitzen als umliegendes Gewebe. Bereits kleine Änderungen können hierbei unerwartet große Auswirkungen nach sich ziehen. Die Realisierungsmöglichkeit über Kurven ist technisch vergleichsweise einfach, erfordert jedoch erfahrene Benutzer um optisch ansprechende Ergebnisse zu erzie-

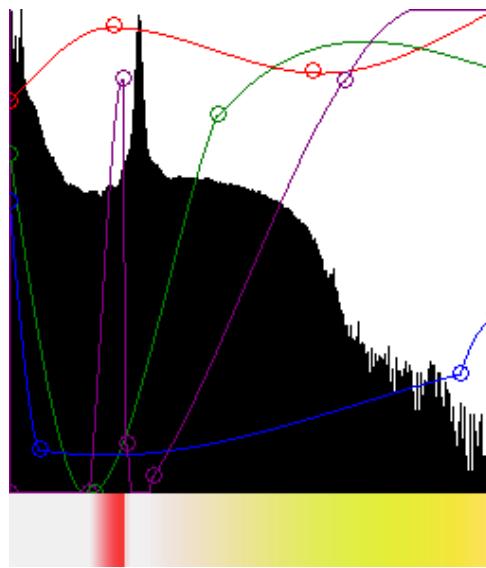


Bild 4.11: Transferfunktion GUI mit Histogramm im Hintergrund. Resultierende 1-D-Textur unten.

len. [Guo11] schlägt daher ein What You See is What You Get (WYSIWYG)-Verfahren vor, dass ähnlich wie Bildbearbeitungsprogramme, etliche visuelle Werkzeuge anbietet. So kann beispielsweise mit Hilfe des Radiergummis direkt in der 3-D-Ansicht die Opazität verändert werden. Es entsteht der Eindruck des Wegradierens. Mit der Komplexität der Transferfunktion wachsen gleichzeitig die Anforderungen an die Benutzerschnittstelle.

Um den Rahmen dieser Arbeit nicht zu sprengen, wurde der in Abbildung 4.11 zu sehende klassische Ansatz zum Bearbeiten von 1-D-Transferfunktion realisiert. Für eine bessere Bedienbarkeit ist es sinnvoll, im Hintergrund ein Histogramm des Datensatzes zu visualisieren. Hierdurch wird die Anzahl der Daten ersichtlich, welche von der aktuellen Einstellung betroffen sind [Mei00, Pre00]. Ein übliches lineares Histogramm ist bei Volumendaten nicht Aussagekräftig, durch ungleich verteilte Skalarwerte tritt oftmals eine deutliche Spitze auf. Durch die vertikale Skalierung fallen andere Skalarwerte nicht weiter ins Gewicht. Aufgrund dessen wurde das lineare Histogramm für die Visualisierung in ein logarithmisches Histogramm umgerechnet. Die Farbanteile Rot, Grün und Blau sowie die Opazität werden jeweils über voneinander unabhängige Kurven dargestellt. Die Kurven hingegen werden durch verschiebbare Kontrollpunkte definiert. Die x-Achse steht hierbei für den Skalarwert, die y-Achse für die Gewichtung der Komponente beim gewählten Skalarwert. Die restlichen Werte werden automatisch interpoliert. Das fertige Ergebnis wird in einer, von der Grafikkarte nutzbaren, 1-D-Textur gespeichert.

In beispielsweise MRI- oder CT-Datensätzen werden verschiedene Gewebetypen von den gleichen Skalarwerten repräsentiert. Aufgrund dessen ist es dem Benutzer nicht immer möglich, unter Zuhilfenahme von Klassifikation relevante Strukturen sichtbar zu machen [Mei00, RS06].

Um so geartete Strukturen sichtbar zu machen, müssen diese in den Volumendaten über spezielle Verfahren gekennzeichnet werden. Dieser Vorgang ist unter dem Begriff *Segmentierung* bekannt, kann je nach Situation sehr einfach oder sehr komplex sein und ist nicht immer automatisierbar [Sha99, Mei00].

## 4.8 Gradientenschätzung

Die Gradientenberechnung im zweidimensionalen Raum ist wichtig für die Aufgabenstellung der Kantenerkennung. Im dreidimensionalen Raum hingegen relevant für Gradientenbasierendes Shading [Mih03]. [Lev88] schlug als einer der ersten den Einsatz von Gradienten für die lokale Beleuchtung beim Volumen-Rendering vor. Der Gradient wird normalisiert und anschließend bei der Beleuchtung als Normalenvektor verwendet. Gradienten finden beim Volumen-Rendering auch abseits des Shading Verwendung. So nutzt bei der Strahlverfolgung die MIP-Variante GMIP [Mor] für die Maximum-Suche die Gradientengewichte anstatt Skalarwerte. Zur besseren Modellierung von Übergängen zwischen Materialien, wie beispielsweise von Gewebe zu Knochen, lassen sich Gradientengewichte als Index in multidimensionalen Transferfunktionen einsetzen [Kni01].

**Funktionssignatur und Ablauf** Die Shaderfunktion in Quellcode 4.15 erhält als Eingabe eine Position  $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T, \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$  im Volumen und liefert als Ergebnis einen nicht normalisierten Gradientenvektor zurück. Für die Gradientenschätzung existiert keine allgemeingültige

```
1 vec3 Gradient(vec3 Position);
```

Quellcode 4.15: Funktionssignatur für die Gradientenschätzung.

Lösung. In Abhängigkeit des verwendeten Datensatzes und den Anforderungen an die Bildqualität sowie Bildwiederholrate ist jeweils ein anderes Verfahren geeigneter [Mei02]. Aufgrund dessen sollen in diesem Abschnitt verschiedene gängige Verfahren für die Schätzung der drei Komponenten des Gradientenvektors mit dem *Nabla*-Operator  $\Delta f(x, y, z)$  vorgestellt werden.

### Vorwärtsdifferenzen und Rückwärtsdifferenzen Vorwärtsdifferenzen

$$\Delta f(x, y, z) \approx \begin{pmatrix} f(x+1, y, z) - f(x, y, z) \\ f(x, y+1, z) - f(x, y, z) \\ f(x, y, z+1) - f(x, y, z) \end{pmatrix} \quad (4.10)$$

sowie Rückwärtsdifferenzen

$$\Delta \mathbf{f}(x, y, z) \approx \begin{pmatrix} f(x, y, z) - f(x - 1, y, z) \\ f(x, y, z) - f(x, y - 1, z) \\ f(x, y, z) - f(x, y, z - 1) \end{pmatrix} \quad (4.11)$$

liefert einen Gradienten bei nur vier Zugriffen auf die Volumendaten. Aufgrund der stark eingeschränkten Berücksichtigung der unmittelbaren Nachbarschaft hat Rauschen im Datensatz eine entsprechend große Auswirkung auf die resultierenden Gradienten. Die Herleitung dieser Gradientenschätzung ist in [Had06, S. 109] zu finden. Ein Fragmentshader für die Gradientenschätzung über Vorwärtsdifferenzen wurde in Quellcode 4.16 abgebildet. Die *Gradient*-

```

1 vec3 Gradient(vec3 Position)
2 {
3     float value = GradientInput(Position);
4     float valueX = GradientInputOffset(Position, ivec3(1, 0, 0));
5     float valueY = GradientInputOffset(Position, ivec3(0, 1, 0));
6     float valueZ = GradientInputOffset(Position, ivec3(0, 0, 1));
7     return vec3(valueX - value, valueY - value, valueZ - value);
8 }
```

Quellcode 4.16: Fragmentshader für die Gradientenschätzung über Vorwärtsdifferenzen.

*tInput*-Shaderfunktion ruft im einfachsten Fall direkt die *FetchScalar*-Shaderfunktion auf. In Quellcode 4.17 ist zu sehen, dass durch diese Abkapselung des Zugriffs auf die Volumendaten eine Gradientenschätzung auf den klassifizierten Skalarwerten ebenfalls möglich ist, ohne hierzu komplett neue Gradientenschätzungen zu implementieren. Gleiches gilt für die *GradientInputOffset*-Shaderfunktion.

```

1 float GradientInput(vec3 Position)
2 {
3     return Classification(FetchScalar(Position)).a;
4 }
```

Quellcode 4.17: Fragmentshader für die Gradientenschätzung auf klassifizierten Skalarwerten.

## Zentrale Differenzen

Die zentrale Differenzen

$$\Delta \mathbf{f}(x, y, z) \approx \frac{1}{2} \begin{pmatrix} f(x + 1, y, z) - f(x - 1, y, z) \\ f(x, y + 1, z) - f(x, y - 1, z) \\ f(x, y, z + 1) - f(x, y, z - 1) \end{pmatrix} \quad (4.12)$$

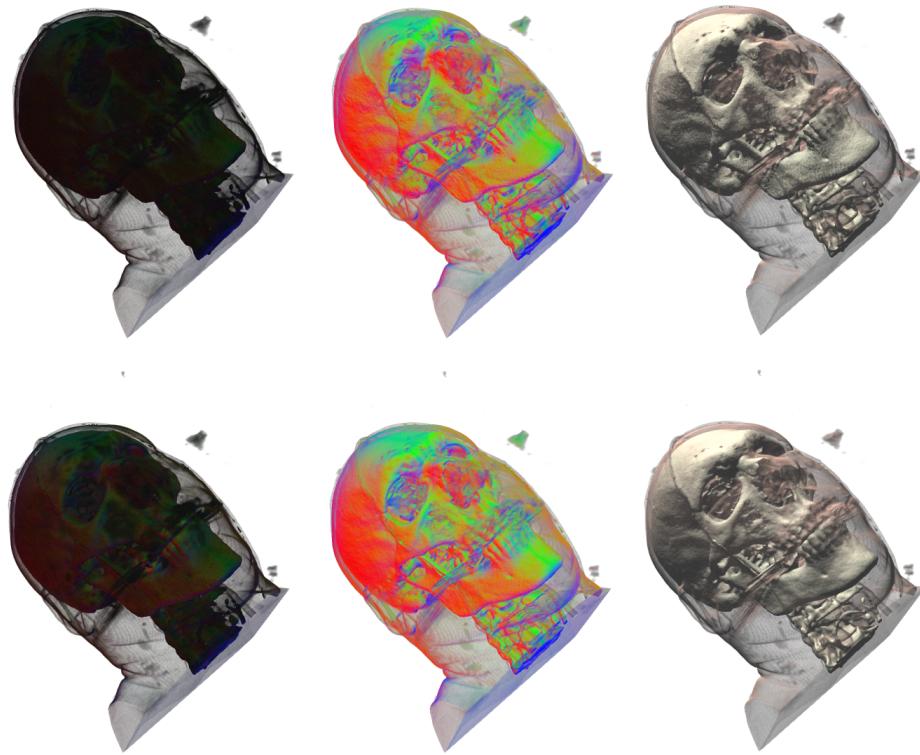


Bild 4.12: Von links nach rechts: Gradient, normalisierter Gradient und Beleuchtung mithilfe des normalisierten Gradienten. Oben Vorwärtsdifferenzen, unten zentrale Differenzen. Volumendatensatz *Head (Visible Male)*.

mit sechs Zugriffen auf die Volumendaten ist einer der am häufigsten eingesetzten Ansätze und liefert qualitativ bessere Ergebnisse als die Vorwärtsdifferenzen [Had06, S. 111]. Für jede Dimension des Volumens wird jeweils die zentrale Differenz zwischen zwei benachbarten Voxeln berechnet. Dies resultiert in einer Annäherung der lokalen Änderung der Skalarwerte [Mei00]. Abbildung 4.12 zeigt, dass bei Vorwärtsdifferenzen an einigen Stellen bei der Beleuchtung ein Rauschen zu erkennen ist, während zentrale Differenzen zu etwas weicheren Farbverläufen führen.

**Qualitativ hochwertige Gradienten** Nach [Rop10a] sind Gradienten-basierende Verfahren moderneren Beleuchtungsmodellen, die ohne Gradientenschätzung auskommen, in Bildqualität, Tiefenwahrnehmung sowie Bildwiederholrate unterlegen. Aufgrund dessen sollte je nach Anwendungsfall geprüft werden, ob die Berechnung von qualitativ hochwertigen Gradienten Sinnvoll ist oder nicht besser auf Gradienten-basierende Verfahren verzichtet wird. Da Gradi-

enten auch abseits der Beleuchtung Einsatz finden, sollen im folgenden Verfahren zum ermitteln qualitativ hochwertiger Gradienten nicht unerwähnt bleiben.

Ein naiver Ansatz zur Steigerung der Gradientenqualität besteht darin, zentrale Differenzen für die aktuelle Position und deren acht Nachbarn an den jeweiligen Eckpunkten zu ermitteln. Das Ergebnis wird anschließend gemittelt um einen Gradienten zu erhalten, der weniger anfällig für Rauschen im Datensatz ist. Dies bedeutet jedoch, dass pro ermittelten Gradienten neun mal die Berechnung für zentrale Differenzen ausgeführt werden muss, die wiederum jeweils sechs Texturzugriffe erfordern. Damit ergeben sich 54 Texturzugriffe um einen Gradienten an einer gegebenen Position zu ermitteln. Die gesteigerte Gradientenqualität geht auf Kosten einer deutlich verringerten Bildwiederholrate und ist daher für eine interaktive Darstellung weniger geeignet.

Eine weitere Möglichkeit zur Ermittlung qualitativ hochwertiger Gradienten besteht in einer Faltung anhand eines diskreten  $3 \times 3 \times 3$  Filterkernel. Hierzu müssen Texturzugriffe für die 26 Nachbarn erfolgen, was zur Folge hat das insgesamt 27 Texturzugriffe benötigt werden. Nach [Mei00] ist der Sobel-Operator besonders geeignet um die hierfür benötigten Gewichtungen aufzustellen. Der für 2-D definierte Sobel-Operator wird in der Regel zur Kantenerkennung verwendet, [Had06, S. 112] zeigt, wie der Operator in die dritte Dimension gehoben werden kann.

[Pha05, Kap. 20] präsentiert eine effiziente Gradientenschätzung über kubische B-Splines, die sich ebenfalls als Rekonstruktionsfilter zum Ermitteln von Skalarwerten nutzen lassen. Die trilineare Filterung der GPU wird dazu verwendet, um die 64 Summanden für den tricubischen Filter mit lediglich acht Texturzugriffen zu verarbeiten.

Die Anzahl der zusätzlichen Texturzugriffe zur Ermittlung des Gradienten lässt sich auf einen Zugriff reduzieren, gleichzeitig sind jedoch deutlich aufwändigere Gradientenberechnungen als hier vorgestellt möglich. Hierzu werden die Gradienten einmalig berechnet und in einer zusätzlichen Textur abgelegt. Früher fand diese Vorberechnung auf der CPU statt, heutzutage bietet es sich für diese Aufgabe ebenfalls die GPU an. Zu den Anfängen des GPU-basierten Volumen-Renderings war dies die gängige Lösung, da die damaligen Grafikkarten für eine *On-The-Fly*-Gradientenschätzung nicht Leistungsfähig genug waren. Um Grafikkartenspeicher zu sparen, ohne jedoch einen deutlichen Qualitätsverlust hinnehmen zu müssen, bietet sich die Nutzung der *LATC2*<sup>7</sup> Kompression an. Dies ist jedoch nur bei normalisierten Gradienten möglich, die Gradientengewichtung geht hierdurch verloren. Während diese Lösungsmöglichkeit auf den ersten Blick attraktiv erscheinen mag, so macht diese lediglich für statische Volumendaten Sinn. Wird die Gradientenschätzung auf bereits klassifizierten Daten angewandt, so muss bei

---

<sup>7</sup>Früher bekannt als *3DC+* und *ATI2N*, in DirectX 10 läuft diese Texturkompression unter dem Namen *BC5*

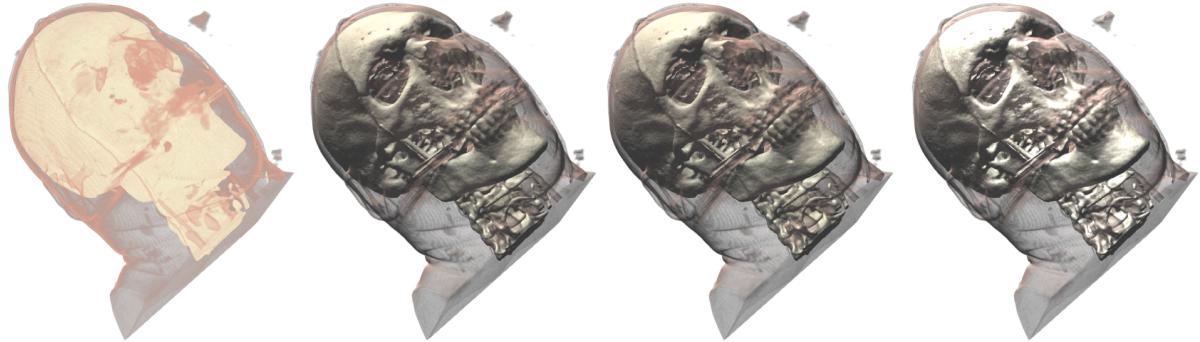


Bild 4.13: Von links nach rechts: Keine Beleuchtung, Lambertian, Blinn-Phong, Cook-Torrance. Volumendatensatz *Head (Visible Male)*.

jeder Änderung bei der Klassifizierung, die Gradienten-Textur aktualisiert werden. Der erhöhte Speicherbedarf kann je nach Größe der Volumendaten ebenfalls ein Problem darstellen.

## 4.9 Lokale Beleuchtung

Der letzte Abschnitt dieses Kapitels geht auf die lokale Beleuchtung eines Punktes entlang des durch das Volumen geworfenen Strahls ein. Anhand Abbildung 4.13 wird ersichtlich das die Beleuchtung einen nicht unerheblichen Einfluss auf das Gesamtergebnis hat. Durch die Abstraktion unterscheidet sich die hier behandelte Shaderfunktion nicht von der traditionellen polygonbasierenden Computergrafik. Aufgrund dessen soll in diesem Abschnitt nur ein kurzer Überblick vermittelt werden, ohne allzu sehr in die Tiefe zu gehen.

**Funktionssignatur und Ablauf** Die Shaderfunktion in Quellcode 4.18 erhält als Eingabe eine Oberflächenfarbe, einen normalisierten Oberflächenvektor, einen normalisierten Blickrichtungsvektor sowie einen normalisierten Lichtrichtungsvektor. Ergebnis der Beleuchtung ist ei-

```
vec3 Illumination(vec3 SurfaceColor, vec3 SurfaceNormal, vec3  
ViewingDirection, vec3 LightDirection);
```

Quellcode 4.18: Funktionssignatur für die lokale Beleuchtung.

ne Farbe die vom gewählten Beleuchtungsmodell abhängt. Gängige Beleuchtungsmodelle bestehen in der Regel aus einem Anteil  $I_{Ambient}$  für globales Umgebungslicht sowie den lichtabhängigen Bestandteilen  $I_{Diffuse}$  und  $I_{Glanzlicht}$ . Das Glanzlicht ist ebenfalls vom Blickwin-

kel abhängig. Zusammengesetzt über

$$I = I_{\text{Ambient}} + I_{\text{Diffuse}} + I_{\text{Glanzlicht}} \quad (4.13)$$

ergibt sich die resultierende Beleuchtung. Bei traditionellen 8 bit pro Farbkomponente im Bildpuffer, muss sichergestellt werden, dass  $I$  im Intervall  $[(0 \ 0 \ 0)^T, (1 \ 1 \ 1)^T]$  liegt. Bei HDR fällt diese Einschränkung weg.

Um es der aufrufenden Instanz zu ermöglichen diese Shaderfunktion mehrmals für unterschiedliche Lichtquellen aufzurufen, sollte eine Implementierung dieser Shaderfunktion lediglich  $I_{\text{Diffuse}}$  und  $I_{\text{Glanzlicht}}$  berücksichtigen. Ein Fragmentshader für eine lokale Beleuchtung über das Lambertsche Gesetz wurde in Quellcode 4.19 abgebildet, ein Glanzlicht ist hier nicht vorhanden.

```

1 uniform vec3 LightColor;
2 vec3 Illumination(vec3 c, vec3 n, vec3 vDir, vec3 lDir)
3 {
4     float lightIntensity = clamp(dot(lDir, n), 0.0, 1.0);
5     return c*lightIntensity*LightColor;
6 }
```

Quellcode 4.19: Fragmentshader für die lokale Beleuchtung über das Lambertsche Gesetz. Parameternamen in der Funktionssignatur aus Platzgründen gekürzt.

**Gängige Beleuchtungsmodelle** Bereits im Jahre 1760 wurde mit dem Lambert-Beleuchtungsmodell eine Simulation der Licht-Oberflächen Interaktion beschrieben. Diese kann im einfachsten Fall mit einem Skalarprodukt zwischen einem normalisierten Oberflächenvektor sowie einen normalisierten Lichtrichtungsvektor berechnet werden. Für die meisten gängigen Beleuchtungsmodelle bildet die Bidirectional Reflectance Distribution Function (BRDF) das Fundament für das Reflexionsverhalten von Oberflächen eines Materials in Abhängigkeit einer Lichtquelle. Die BRDF wurde von Fred Nicodemus im Jahre 1965 definiert [Nic65] und wird in [Nic77] beschrieben. In der Echtzeitcomputergrafik werden vereinfachte Formen der BRDF verwendet. Das Phong-Beleuchtungsmodell [Pho75] aus dem Jahre 1975 stellt die Grundlage zahlreicher Beleuchtungsmodelle dar und fügt ein Glanzlicht hinzu das sich positiv auf die räumliche Darstellung von Objekten auswirkt. Dieses Modell wurde 1977 durch das Blinn-Beleuchtungsmodell<sup>8</sup> [Bli77] erweitert um die notwendigen Berechnungen zu beschleunigt. Das Blinn-Beleuchtungsmodell ist in der Echtzeitcomputergrafik weit verbrei-

---

<sup>8</sup>Auch bekannt als Blinn-Phong-Beleuchtungsmodell

tet. Das zwischen 1967-1982 entwickelte Cook-Torrance-Beleuchtungsmodell<sup>9</sup> [Tor67, Coo82] ist näher an der Realität als die bisher erwähnten Modelle, benötigt allerdings ebenfalls mehr Rechenleistung. Moderne Grafikkarten besitzen ausreichende Rechenleistung für den Einsatz rechenintensiver Beleuchtungsmodelle. Weiterführende Informationen zu den in der Computergrafik üblicherweise eingesetzten Beleuchtungsmodelle sind in [Fer03] und [AM08] zu finden.

---

<sup>9</sup>Auch bekannt als Torrance-Sparrow-Beleuchtungsmodell



# Kapitel 5

## Experimente

Bereits im Jahre 1988 war Volumen-Rendering ein aktives Forschungsgebiet, auch wenn das Rendern kleiner Datensätze mehrere Minuten bis hin zu einer Stunde dauern konnte [Lev88, S. 34].

Die Zielsetzung dieser Arbeit besteht in der Erstellung eines skalierbaren und Echtzeitfähigen Volumen-Renderers. Das Experimente Kapitel demonstriert die Bandbreite der erzielbaren Bildwiederholraten.

### 5.1 Versuchsumgebung und Datenmaterial

**Hard- und Software** In Tabelle 5.1 ist die für die Experimente verwendete Hard- und Software aufgelistet.

|             | System 1                          | System 2                        |
|-------------|-----------------------------------|---------------------------------|
| Typ         | Desktop-PC                        | Notebook <sup>1</sup>           |
| GPU         | NVIDIA GeForce 285 GTX (2048 MiB) | AMD Radeon HD 6970M (2048 MiB)  |
| CPU         | Intel Core 2 Quad Q9550 2,83 GHz  | Core i7-2630QM 2,00 – 2,90 GHz  |
| RAM         | 8 GiB                             | 16 GiB                          |
| OS          | Windows 7 Enterprise (64 bit)     | Windows 7 Professional (64 bit) |
| GPU-Treiber | 296.10 WHQL                       | Catalyst 12.3                   |

<sup>1</sup> Typbezeichnung *Samsung Serie 7 Gamer 700G7A*

Tabelle 5.1: Die für die Leistungstests verwendete Hard- und Software.

**Integration in eine 3-D-Engine** Der Volumen-Renderer ist als Plugin realisiert. Wesentliche Komponenten sind der Volumenknoten, Szenenknoten für Schnittelelemente sowie der Szenen-

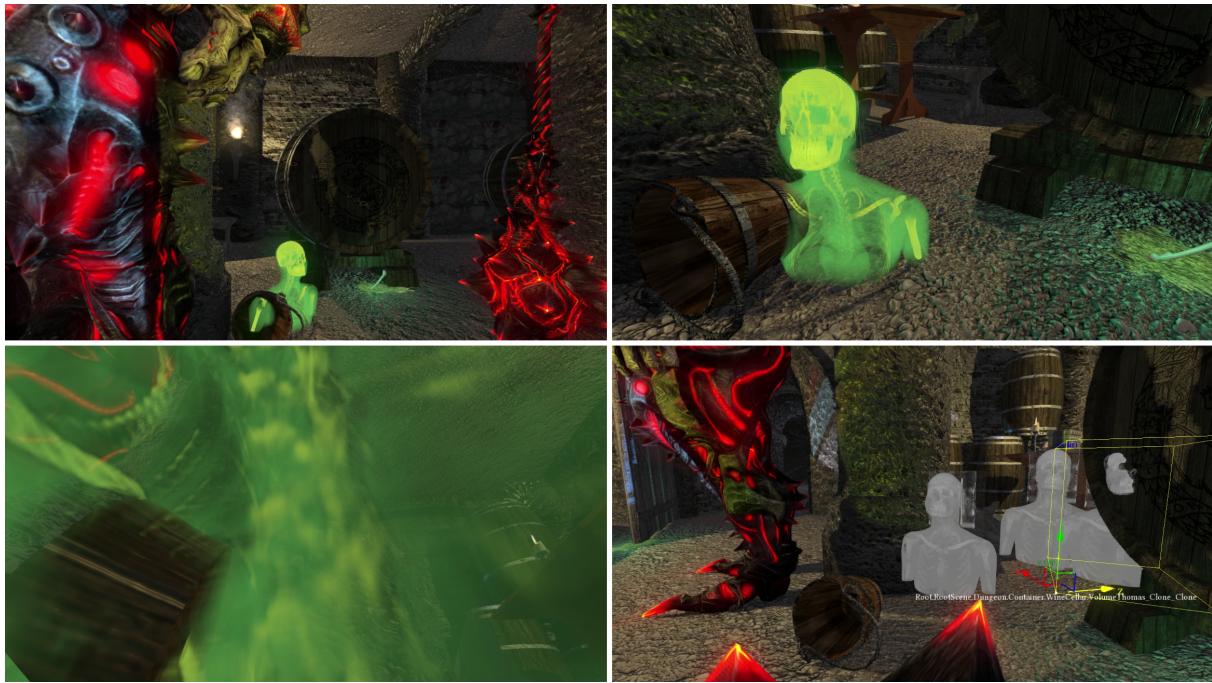


Bild 5.1: Volumen-Rendering Integration in eine 3-D-Engine. Volumendatensatzes *Thomas*.

Renderschritt. Wie in Abbildung 5.1 zu sehen, lassen sich die erarbeiteten Komponenten in bereits bestehende 3-D-Szenen einfügen. Eine Licht und Schatteninteraktion mit dem Rest der 3-D-Szene konnte aufgrund des Komplexität im Rahmen dieser Arbeit nicht realisiert werden. Die verwendete 3-D-Engine sortiert Szenenknoten vor dem Zeichnen automatisch in Abhängigkeit von der Entfernung zur Kamera. Sich räumlich nicht überschneidende transparente Szenenknoten lassen sich somit korrekt darstellen. Unter Zuhilfenahme des Tiefenpuffers lassen sich nicht transparente Objekte in ein Volumen schieben. Eine räumliche Überlagerung von transparenten Objekten resultiert erwartungsgemäß in falschen Ergebnissen. Eine Lösung dieser Problematik ist Aufwändig.

In den folgenden Experimenten befindet sich in der Szene lediglich ein Volumen sowie eine Kamera und eine direktionale Lichtquelle.

**FPS vs. Millisekunden** Die Leistungsergebnisse werden in der Volumen-Rendering Literatur häufig als Bilder pro Sekunde<sup>1</sup> angegeben. Aufgrund dessen wurde diese Einheit ebenfalls für die Experimente in diesem Kapitel gewählt. Im Gegensatz zu einer Angabe der Berechnungszeit in Millisekunden pro Bild, ist die Einheit FPS nicht linear. Ein Ergebnis von 42 FPS gegenüber 21 FPS bedeutet nicht, dass ein Bild in der Hälfte der Zeit dargestellt werden konnte. Tendieren die Bilder pro Sekunde gegen Null, so sind keine aussagekräftigen Ergebnisse möglich. Bei

---

<sup>1</sup>Engl. Frames Per Second (FPS)

tendenziell größeren Zahlen hingegen besitzen bereits kleine Änderungen am Algorithmus eine deutlich sichtbare Auswirkung auf die Bildwiederholrate.

**Leistungsunterschied zwischen Prozessorarchitekturen** In ersten Versuchen wurde der Leistungsunterschied zwischen einer 32 bit und 64 bit Volumen-Renderer Version untersucht. Erst bei annähernd 1000 FPS wurde ersichtlich, dass die 64 bit Version geringfügig schneller ist. Der Unterschied ist vermutlich auf die auf der CPU laufende Anwendungslogik zurückzuführen. Ein Volumen-Renderer hingegen ist primär durch die Datentransfer und Füllratenleistung der Grafikkarte begrenzt. Die folgenden Experimente fanden in einer als 64 bit übersetzten Version der erstellten Implementierung statt.

**Windows Aero Style** Die Erstellung der Volumen-Renderer Umsetzung wurde auf einer *ATI Mobility Radeon HD 4850*<sup>2</sup> Grafikkarte mit 512 MiB Speicher begonnen. Der  $512 \times 512 \times 1743$  Voxel große Volumendatensatz *Schwein* belegt LATC1 komprimiert 435 MiB (446 208 KiB) und muss sich den Grafikkartenspeicher mit weiteren Ressourcen und Anwendungen teilen. Somit bestand eine Randsituation, in der gerade ausreichend Grafikkartenspeicher vorhanden war, um die Volumendaten zu speichern. Diese Randsituation führte unter Windows 7 zu Leistungsanomalien, unabhängig von Bildauflösung oder Volumen-Rendering Einstellungen.

Bei aktiviertem *Windows Aero Style* wurden 4 FPS im Fenstermodus gemessen. Wurde die Anwendung während der Ausführung in den Vollbildmodus mit gleicher Bildauflösung versetzt, so ergaben sich 45 FPS. Ein Umschalten zurück in den Fenstermodus resultierte wieder in 4 FPS. Wurde die Anwendung mehrmals im Fenstermodus gestartet, so konnte es vorkommen, dass ebenfalls 45 FPS gemessen wurden.

In einem weiteren Versuch wurde *Windows Aero Style* deaktiviert und der oben beschriebene Ablauf wiederholt. Sowohl im Fenstermodus als auch im Vollbildmodus konnten 45 FPS gemessen werden. Wurde die Anwendung mehrmals gestartet, ergaben sich in seltenen Fällen, vom Bildmodus unabhängig 10 FPS.

Wird eine Grafikkarte an die Grenzen ihrer Speicherressourcen gebracht, kann es beim Einsatz umfangreicher Volumendaten zu unberechenbaren Sprüngen im Leistungsverhalten kommen. Die genannten Bildwiederholraten entstanden zu Beginn dieser Arbeit unter günstigen Blickwinkeln sowie einer verringerten Abtastrate und sind daher nicht vergleichbar zu den restlichen Ergebnissen dieses Kapitels. In den folgenden Experimenten wurde *Windows Aero Style* deaktiviert um stabilere Ergebnisse zu erzielen.

---

<sup>2</sup>MSI GT725Q-9047VHP Notebook, Grafikkartentreiber Catalyst 11.11

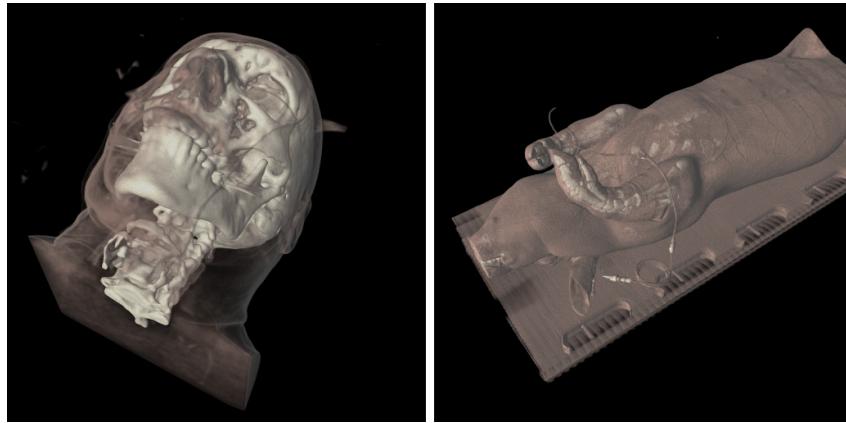


Bild 5.2: Der für die Experimente gewählte Blickwinkel für die Darstellung des Volumendatensatzes *Head (Visible Male)* links und Volumendatensatzes *Schwein* rechts.

## 5.2 Leistungsergebnisse

**Blickwinkel** Abhängig von der GPU-Architektur besitzt die Reihenfolge des Texturzugriffes eine unterschiedliche Auswirkung auf die Effektivität der GPU-Speicherverwaltung. Daher wurde die in Abbildung 5.2 zu sehende schräge Perspektive gewählt.

**Konfigurationsoptionen: Minimum, Medium und Maximum** Konzeptbedingt sind mit dem vorgestellten skalierbaren Volumen-Renderer eine Vielzahl verschiedenartiger Shaderbaustein-Kombinationen denkbar. Prinzipiell ist es möglich, alle Shaderpermutationen automatisiert erstellen zu lassen. Der Versuch, diese anschließend im Detail zu analysieren und leistungstechnisch miteinander zu vergleichen, würde den Rahmen dieser Arbeit jedoch sprengen. Aufgrund dessen sollen in diesem Kapitel Stichproben und besonders herausstehende Kombinationen näher betrachtet werden. Für die Datenerhebung wurden GLSL-Shader verwendet.

Um die Bandbreite der, durch unterschiedliche Konfigurationen erzielbaren Bildwiederholrate auszuloten, wurden in Tabelle 5.2 die Standardeinstellungen jeweils Minimum- und Maximumkonfigurationen gegenübergestellt. Die Messungen wurden für den umfangreicheren Volumendatensatz *Schwein* mit der gleichen Bildauflösung von  $512 \times 512$  wiederholt und in Tabelle 5.3 zusammengefasst.

Die Medium-Standardeinstellungen des Volumen-Rendering Systems beruhen auf einem Kompromiss zwischen Bildqualität und Bildwiederholrate. Für die Rekonstruktion kommt trilineare Filterung zum Einsatz. Die Skalarwerte werden anhand einer 1-D-Transferfunktion klassifiziert und anschließend über eine direktionale Lichtquelle beleuchtet. Für die Gradientenschätzung werden zentrale Differenzen verwendet, die im Blinn-Phong Beleuchtungsmodell

| Konfiguration | System 1 | System 2 |
|---------------|----------|----------|
| Null          | 1050     | 855      |
| Minimum       | 271      | 219      |
| Medium        | 84       | 82       |
| Maximum       | 17       | 17       |

Tabelle 5.2: Konfigurationsoptionen: Minimum, Medium und Maximum. Ergebnisse in FPS. Volumendatensatz *Head (Visible Male)*.

| Konfiguration | System 1 | System 2 |
|---------------|----------|----------|
| Null          | 955      | 855      |
| Minimum       | 4        | 29       |
| Medium        | 2        | 8        |
| Maximum       | 1        | 2        |

Tabelle 5.3: Konfigurationsoptionen: Minimum, Medium und Maximum. Ergebnisse in FPS. Volumendatensatz *Schwein*.

als Normalenvektor dienen. Um das Nyquist-Shannon-Abtasttheorem [Sha49, Wyn98, Uns00] nicht zu verletzen, beträgt die Standardeinstellung der Abtastrate 100 %. In den folgenden Experimenten wird jeweils nur eine Einstellung variiert, die restlichen Konfigurationsmöglichkeiten verbleiben auf den Standardeinstellungen.

Die höchste Bildwiederholrate in der Minimum-Konfiguration geht auf Kosten der Bildqualität. Für die Rekonstruktion wurde der nächster-Nachbar-Filter gewählt. Die Klassifikation gibt den originalen Skalarwert zurück, somit findet keine Klassifikation statt. Ebenfalls wird kein Shading eingesetzt, eine Gradientenschätzung oder ein Beleuchtungsmodell kommt somit nicht zum Einsatz. Anhand dieser Minimal-Konfiguration lässt sich der für die Strahlverfolgung benötigte Zeitaufwand messen. Zum Vergleich wurde ebenfalls eine Null-Konfiguration in Tabelle 5.2 aufgelistet, die keine Strahlverfolgung vornimmt sondern die Strahlstartposition als Farbwert zurückliefert.

Die beste Bildqualität in der Maximum-Konfiguration geht auf Kosten der Bildwiederholrate. Für eine qualitativ hochwertige Rekonstruktion wurden kubische B-Splines gewählt. Für weichere Gradienten im Cook-Torrance Beleuchtungsmodell wurde ein naiver Ansatz gewählt: Es werden zentrale Differenzen für die aktuelle Position und deren acht Nachbarn an den jeweiligen Eckpunkten zu ermitteln, die neun resultierenden Gradienten werden gemittelt.

**Texturkompression** Der Einsatz von Texturkompression verringert den Speicherbedarf, gleichzeitig wird der Datentransfer verringert. Dies wirkt sich im Allgemeinen positiv auf die Bildwiederholrate aus. Die Volumendaten in diesem Kapitel sind LATC1 komprimiert. Zum Vergleich werden die Experimente des vorherigen Abschnittes mit deaktivierter Texturkom-

pression wiederholt. Die Ergebnisse sind in Tabelle 5.4 und Tabelle 5.5 zusammengetragen. Für einen Direktvergleich sind jeweils rechts in einer Spalte die Ergebnisse aus Tabelle 5.2

| Konfiguration | System 1 | System 2 |     |
|---------------|----------|----------|-----|
| Minimum       | 235      | 271      | 216 |
| Medium        | 74       | 84       | 79  |
| Maximum       | 16       | 17       | 17  |

Tabelle 5.4: Konfigurationsoptionen: Minimum, Medium und Maximum. Ergebnisse in FPS. Bildwiederholrate bei Texturkompression jeweils rechts. Volumendatensatz *Head (Visible Male)* ohne Kompression.

| Konfiguration | System 1 | System 2 |    |
|---------------|----------|----------|----|
| Minimum       | 8        | 4        | 20 |
| Medium        | 4        | 2        | 5  |
| Maximum       | 1        | 1        | 2  |

Tabelle 5.5: Konfigurationsoptionen: Minimum, Medium und Maximum. Ergebnisse in FPS. Bildwiederholrate bei Texturkompression jeweils rechts. Volumendatensatz *Schwein* ohne Kompression.

und Tabelle 5.3 eingefügt. Die Bildwiederholraten in Tabelle 5.4 entsprechen den Erwartungen, die Bildwiederholrate steigert sich beim Einsatz von Texturkompression. Bei Volumendatensatz *Schwein* in Tabelle 5.5 wirkt sich die Texturkompression auf Testsystem 2 weiterhin positiv aus, Testsystem 1 hingegen zeigt eine unerwartete Verschlechterung der Bildwiederholrate. Eventuell verlagerte sich in diesen Tests der Flaschenhals von der Transferrate hin zum Rechenwerk, das möglicherweise für die Texturdekompresion verantwortlich ist. Dies ist jedoch nur reine Spekulation da derartige Details über die Hardware nicht vorliegen.

**Bildauflösung** Beim Volumen-Rendering über Raycasting wird pro Bildpunkt jeweils ein Strahl durch das Volumen verfolgt. Für den  $512 \times 512 \times 1743$  Voxel großen Volumendatensatz *Schwein* müssen gemäß Nyquist-Shannon-Abtasttheorem für die Rekonstruktion mindestens 3486 Abtastpunkte entlang des Strahls vorhanden sein. Als bildbasiertes Verfahren ist beim Volumen-Raycasting die Bildwiederholrate maßgeblich von der Bildauflösung abhängig. In Tabelle 5.6 wird der  $128 \times 256 \times 256$  Voxel große Volumendatensatz *Head (Visible Male)* verwendet, während in Tabelle 5.7 der Volumendatensatz *Schwein* zum Erheben der Ergebnisse verwendet wurde. Um beim Variieren von Einstellungen messbare Unterschiede zu erhalten, viel die Wahl für die weiteren Experimente auf die Bildauflösung  $512 \times 512$ .

| Bildauflösung      | System 1 | System 2 |
|--------------------|----------|----------|
| $1024 \times 1024$ | 27       | 24       |
| $512 \times 512$   | 84       | 82       |
| $256 \times 256$   | 174      | 222      |
| $128 \times 128$   | 286      | 475      |

Tabelle 5.6: Variieren der Bildauflösung. Ergebnisse in FPS. Volumendatensatz *Head (Visible Male)*.

| Bildauflösung      | System 1 | System 2 |
|--------------------|----------|----------|
| $1024 \times 1024$ | 2        | 4        |
| $512 \times 512$   | 2        | 8        |
| $256 \times 256$   | 4        | 15       |
| $128 \times 128$   | 8        | 42       |

Tabelle 5.7: Variieren der Bildauflösung. Ergebnisse in FPS. Volumendatensatz *Schwein*.

**Abtastrate** Anhand Tabelle 5.8 und Tabelle 5.9 wird ersichtlich, dass die Wahl der Abtastrate einen großen Einfluss auf die Bildwiederholrate hat. Die Verringerung der Abtastrate besitzt

| Abtastrate | System 1 | System 2 |
|------------|----------|----------|
| 100 %      | 84       | 82       |
| 50 %       | 158      | 149      |
| 20 %       | 325      | 294      |
| 10 %       | 495      | 449      |
| 1 %        | 1020     | 820      |

Tabelle 5.8: Variieren der Abtastrate. Ergebnisse in FPS. Volumendatensatz *Head (Visible Male)*.

eine deutlich messbare Auswirkung auf die Bildwiederholrate. Selbst bei einer geringen Abtastrate von 10 % ist das Schwein noch als solches zu erkennen. Eine feingranulare Variation der Abtastrate ist problemlos möglich.

Der naheliegende Vorschlag von [Cor11], die Abtastrate dynamisch zur Laufzeit zu regulieren um eine stabile Bildwiederholrate zu erzielen, lässt sich prinzipiell auf das gesamte Volumen-Rendering System übertragen. So könnte auf ausdrücklichen Wunsch des Benutzers hin ein automatisches System eingreifen, das dynamisch auf einfache Verfahren für beispielsweise die Gradientenschätzung umschaltet, wenn die Bildwiederholrate in einen kritischen Bereich kommt. Es ist abzusehen, dass neben der Bildwiederholrate weitere Informationen wie beispielsweise Kamerabewegung mit einfließen sollten. Bei Bewegung kann das menschliche Auge weniger Details ausmachen, gleichzeitig fällt in Bewegung eine niedrige Bildwiederholrate umso mehr ins Gewicht.

| Abtastrate | System 1 | System 2 |
|------------|----------|----------|
| 100 %      | 2        | 8        |
| 50 %       | 5        | 16       |
| 20 %       | 11       | 38       |
| 10 %       | 23       | 71       |
| 1 %        | 157      | 324      |

Tabelle 5.9: Variieren der Abtastrate. Ergebnisse in FPS. Volumendatensatz *Schwein*.

Bei einer zu hohen Abtastrate reagiert die GPU nicht mehr korrekt. In Microsoft Windows eingebaute Sicherheitsvorkehrungen führen einen Neustart des Grafikkartentreibers durch. In ungünstigen Situationen ist ein kompletter Systemneustart notwendig. Es gilt daher sicherzustellen, dass die Abtastrate niemals zu hoch eingestellt wird.

**Volumengröße** Die Anzahl der Abtastpunkte entlang eines Strahles ist abhängig von der Volumengröße. Um eine Abtastrate von 100 % beibehalten zu können, lässt sich wie in Tabelle 5.10 aufgelistet ebenfalls die Volumengröße variieren. Für die  $1 \times 1 \times 3$  Voxel große Mip-

| Mipmap | Volumengröße                 | System 1    | System 2   |
|--------|------------------------------|-------------|------------|
| 0      | $512 \times 512 \times 1743$ | 2           | 8          |
| 1      | $256 \times 256 \times 871$  | 12          | 23         |
| 2      | $128 \times 128 \times 435$  | 55          | 48         |
| 3      | $64 \times 64 \times 217$    | 104         | 90         |
| 4      | $32 \times 32 \times 108$    | 173         | 158        |
| 5      | $16 \times 16 \times 54$     | 292         | 275        |
| 6      | $8 \times 8 \times 27$       | 482         | 461        |
| 7      | $4 \times 4 \times 13$       | 776         | 724        |
| 8      | $2 \times 2 \times 6$        | 1130        | 830        |
| 9      | $1 \times 1 \times 3$        | $\sim 1130$ | $\sim 830$ |
| 10     | $1 \times 1 \times 1$        | $\sim 1130$ | $\sim 830$ |

Tabelle 5.10: Variieren der Volumengröße. Ergebnisse in FPS. Volumendatensatz *Schwein*.

map 9, sowie die  $1 \times 1 \times 1$  Voxel große Mipmap 10 lag der Unterschied zur Mipmap 8 im Bereich der Messungenauigkeit. Die Bildqualität von Mipmap 1 ist noch akzeptabel, während ab Mipmap 2 das Orientieren im Datensatz bereits schwerfällt. Ab Mipmap 5 sind die Volume-Rendering Ergebnisse nicht mehr als Schwein zu identifizieren.

Um die Bildwiederholrate während einer Interaktion zu steigern, bietet es sich an, die Volumengröße durch die Wahl einer kleineren Mipmap zu verringern. Das direkte Umschalten zwischen zwei Mipmaps fällt jedoch optisch unangenehm auf. Grafikkarten unterstützen ein Interpolieren zwischen zwei Mipmaps. Wie in Tabelle 5.11 zu sehen ist, hat das weiche Über-

blenden von einer Mipmap zu einer anderen jedoch einen negativen Einfluss auf die Bildwiederholrate. Dies lässt sich vermutlich dadurch erklären, dass die Grafikkarte statt eines Tex-

| Mipmap | Volumengröße                 | System 1 | System 2 |
|--------|------------------------------|----------|----------|
| 0      | $512 \times 512 \times 1743$ | 2        | 8        |
| 0,5    | $384 \times 384 \times 1307$ | 2        | 4        |
| 1      | $256 \times 256 \times 871$  | 12       | 23       |
| 1,5    | $192 \times 192 \times 653$  | 10       | 21       |
| 2      | $128 \times 128 \times 435$  | 55       | 48       |

Tabelle 5.11: Variieren der Volumengröße mit Zwischenstufen. Ergebnisse in FPS. Volumendatensatz *Schwein*.

turzugriffes nun zwei Texturzugriffe benötigt um in der Lage zu sein, einen Zwischenwert zu berechnen. Während das Ergebnis optisch ansprechend ist, wird bei der Verwendung von Mipmap-Zwischenstufen statt einer Beschleunigung, eine Verringerung der Bildwiederholrate erreicht. Der Nutzen dieser Bildverbesserungsmaßnahme hält sich daher in Grenzen.

**Rekonstruktion** Für jeden Abtastpunkte entlang des Strahls findet eine Skalarwert Rekonstruktion statt. Der im Rahmen dieser Arbeit entstandenen Implementierung stehen für diese Aufgabenstellung vier Alternativen zur Wahl, deren Auswirkung auf die Bildwiederholrate in Tabelle 5.12 und Tabelle 5.13 aufgelistet wurde. Qualitativ hochwertige Rekonstruktionsfil-

| Implementierung                | System 1 | System 2 |
|--------------------------------|----------|----------|
| Nächster-Nachbar-Filterung     | 88       | 86       |
| Trilineare Filterung           | 84       | 82       |
| Kubische B-Splines             | 17       | 28       |
| Kubische B-Splines über Textur | 11       | 23       |

Tabelle 5.12: Variieren der Rekonstruktion. Ergebnisse in FPS. Volumendatensatz *Head (Visible Male)*.

| Implementierung                | System 1 | System 2 |
|--------------------------------|----------|----------|
| Nächster-Nachbar-Filterung     | 3        | 13       |
| Trilineare Filterung           | 2        | 8        |
| Kubische B-Splines             | 1        | 3        |
| Kubische B-Splines über Textur | -        | 3        |

Tabelle 5.13: Variieren der Rekonstruktion. Ergebnisse in FPS. Volumendatensatz *Schwein*.

ter, wie beispielsweise kubische B-Splines, werden nur beim Aufruf der Shaderfunktion für

die Rekonstruktion ausgeführt. Ein direkter Zugriff auf die von der GPU bereitgestellten Volumendaten existiert weiterhin. Ein Wechsel zwischen nächster-Nachbar-Filterung und trilinearer Filterung hat einen Einfluss auf die Gradientenschätzung, da es sich hierbei um Texturfilter Einstellungen handelt. Für die realisierten Gradientenschätzungen wird direkt auf die Volumendaten zugegriffen, somit wird auch bei aktivierten kubischen B-Splines weiterhin trilineare Filterung verwendet.

Der Leistungsunterschied zwischen nächster-Nachbar-Filterung und trilinear Filterung fällt im Volumendatensatz *Head (Visible Male)* vergleichsweise gering aus. Dies lässt sich vermutlich auf die direkte hardwareseitige Unterstützung zurückführen. Im umfangreichen Volumendatensatz *Schwein* hingegen ist zu erkennen, dass trotz dessen bei hoher Anzahl an Texturfilterungen ein Leistungsunterschied vorhanden ist. Für die Rekonstruktion wird eine Faltung eingesetzt. In der Computergrafik wird eine Faltung üblicherweise über eine Maske beschrieben die Gewichtungen für die Nachbarn enthält. Die Maske wird in einer Textur gespeichert. Durch Ändern des Texturinhaltes lassen sich über einen gleichbleibenden Fragmentshader unterschiedliche Faltungen realisieren. Zum Vergleich wurde die Faltung unter Zuhilfenahme von kubischen B-Splines ebenfalls komplett ohne zusätzliche Texturzugriffe im Fragmentshader realisiert. Dies führt auf den Testsystemen zu einer messbaren Steigerung der Bildwiederholrate bei optisch gleichbleibenden Ergebnissen. In den vergangenen Jahren stieg bei Grafikkarten die Leistungsfähigkeit der Rechenwerke schneller als die Speichertransferrate. Als die ersten Shadersprachen verfügbar wurden bestand eine gängige Optimierung darin, soviel Berechnungen und Zwischenergebnisse wie möglich einmalig auf der CPU auszuführen und in einer Textur zu speichern. Bei modernen Grafikkarten verhält es sich heutzutage entgegengesetzt und die damaligen Optimierungen führen zu einer Verschlechterung der Bildwiederholrate.

Während auf System 1 der Volumendatensatz *Schwein* mit kubischen B-Splines ohne zusätzliche Texturzugriffe noch Darstellbar war, wurde bei kubischen B-Splines über eine Textur mit gespeicherten Gewichtungen der Bildschirm schwarz und das System reagierte nicht mehr. Ein Systemneustart wurde erforderlich. In der Regel wird auf Windows 7 ein automatischer Grafikkartentreiber Neustart durchgeführt um eine derartige Situation zu verhindern. Augenscheinlich existieren Situationen in denen dies nicht möglich ist. In Sicherheitskritischen Anwendungen muss sichergestellt werden, dass die Grafikkarte nicht überfordert wird.

**Gradientenschätzung** Der Einfluss verschiedener Gradientenschätzungen auf die Bildwiederholrate wird in Tabelle 5.14 und Tabelle 5.15 zusammengefasst. Die Vorwärtsdifferenzen benötigen vier, zentrale Differenzen sechs Texturzugriffe. Der FPS-Unterschied fällt daher nicht allzu unterschiedlich aus, während die optische Verbesserung in der Regel auch ohne genaues hinsehen sichtbar ist. Die qualitativ hochwertigen Gradienten in der letzten Zeile

| Implementierung        | System 1 | System 2 |
|------------------------|----------|----------|
| Vorwärtsdifferenzen    | 88       | 84       |
| Zentrale Differenzen   | 84       | 82       |
| Zentrale Differenzen 9 | 32       | 29       |

Tabelle 5.14: Variieren der Gradientenschätzung. Ergebnisse in FPS. Volumendatensatz *Head (Visible Male)*.

| Implementierung        | System 1 | System 2 |
|------------------------|----------|----------|
| Vorwärtsdifferenzen    | 2        | 8        |
| Zentrale Differenzen   | 2        | 8        |
| Zentrale Differenzen 9 | 1        | 5        |

Tabelle 5.15: Variieren der Gradientenschätzung. Ergebnisse in FPS. Volumendatensatz *Schwein*.

der Tabelle hingegen benötigen 54 Textzugriffe, die resultierende deutliche Verschlechterung der Bildwiederholrate war daher abzusehen. Während die Visualisierung des Volumendatensatz *Head (Visible Male)* weiterhin Interaktiv möglich ist, ist der Einsatz einer derart aufwändigen Gradientenschätzung für größere Volumendatensätze oder höhere Bildauflösungen während einer Interaktion wenig empfehlenswert.

**Beleuchtung** Die Wahl des Beleuchtungsmodells betrifft primär das Rechenwerk der Grafikkarte, zusätzliche Texturzugriffe finden bei den in Tabelle 5.16 und Tabelle 5.17 aufgeführten Implementierungen keine statt. Die Null-Funktion in der ersten Zeile besteht aus einer

| Implementierung | System 1 | System 2 |
|-----------------|----------|----------|
| Null            | 121      | 105      |
| Lambertian      | 87       | 83       |
| Blinn-Phong     | 84       | 82       |
| Cook-Torrance   | 80       | 61       |

Tabelle 5.16: Variieren der Beleuchtung. Ergebnisse in FPS. Volumendatensatz *Head (Visible Male)*.

Implementierung die als Ergebnis immer Schwarz zurückliefert. Die Shaderfunktion für die Beleuchtung erhält als Eingabeparameter unter anderem einen Normalenvektor, der aus einem Gradienten abgeleitet wird. Der Gradient wiederum wird über die bereits behandelte Gradientenschätzung ermittelt. In der Standardkonfiguration des erstellten Volumen-Renderers wird ein Gradient ausschließlich für die lokale Beleuchtung eingesetzt. Ein guter Shadercompiler ist in der Lage zu erkennen, wenn Berechnungen keinen Einfluss auf das finale Ergebnis besitzen. Das Lambertian-Beleuchtungsmodell besteht aus einem einfachen Skalarprodukt, der

| Implementierung | System 1 | System 2 |
|-----------------|----------|----------|
| Null            | 2        | 9        |
| Lambertian      | 2        | 8        |
| Blinn-Phong     | 2        | 8        |
| Cook-Torrance   | 2        | 8        |

Tabelle 5.17: Variieren der Beleuchtung. Ergebnisse in FPS. Volumendatensatz *Schwein*.

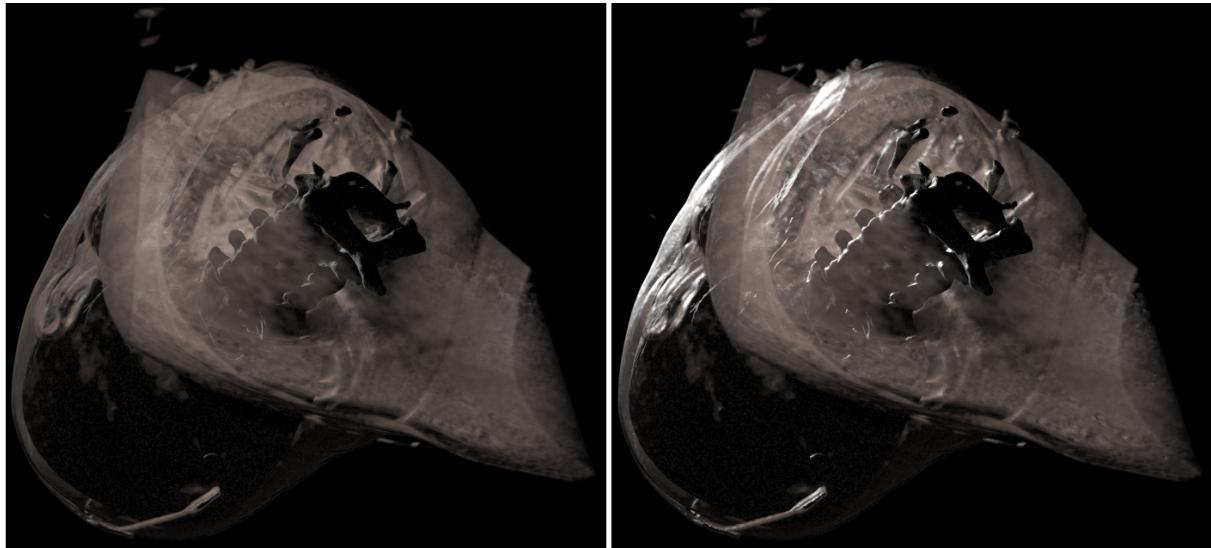


Bild 5.3: Links Blinn-Phong, rechts Cook-Torrance. Volumendatensatz *Head (Visible Male)*.

Leistungsunterschied zur Null-Funktion ist jedoch erwähnenswert. Hier liegt die Vermutung nahe, das der Shadercompiler beim Einsatz der Null-Funktion in der Lage war, die Gradientenschätzung vollständig aus dem resultierenden Shader zu entfernen. Gewissheit ergibt allerdings erst eine Untersuchung des übersetzten Shadercodes. Verglichen mit dem Lambertian-Beleuchtungsmodell, ist die Cook-Torrance Implementierung sehr Rechenintensiv und wirkt sich im Volumendatensatz *Head (Visible Male)* dementsprechend nicht unerwartet auf die Bildwiederholrate aus. Im Volumendatensatz *Schwein* hingegen ist vermutlich die Speichertransferrate der Flaschenhals. Der optische Unterschied ist, abhängig von vielen Faktoren wie beispielsweise des Blickwinkels, nicht immer deutlich zu erkennen. Wie in Abbildung 5.3 zu erkennen, kann das resultierende Bild durch das Cook-Torrance-Beleuchtungsmodell allerdings ebenfalls deutlich interessanter werden. Vergleicht man die Cook-Torrance Leistungsergebnisse zwischen den beiden Testsystemen, so fällt auf, das der FPS-Unterschied zum Blinn-Phong-Beleuchtungsmodell auf Testsystem 2 größer ausfällt als auf Testsystem 1. Der Flaschenhals hat sich augenscheinlich von der Speichertransferrate hin zum Rechenwerk verschoben.

**Back-To-Front vs. Front-To-Back** Abhängig von der gewählten Strahlverfolgung sind weitere Optimierungen möglich. So stabilisiert sich beim Front-To-Back-Compositing die Farbe sobald eine gewisse Opazität überschritten wird und die Strahlverfolgung kann frühzeitig Abgebrochen werden. Als Standardeinstellung wurde ein Schwellwert von 0,95 gewählt. Tabelle 5.18 und Tabelle 5.19 stellen Back-To-Front und Front-To-Back gegenüber. Durch eine frühzeitige

| Implementierung | System 1 | System 2 |
|-----------------|----------|----------|
| Back-To-Front   | 62       | 60       |
| Front-To-Back   | 83       | 82       |

Tabelle 5.18: Back-To-Front vs. Front-To-Back. Ergebnisse in FPS. Volumendatensatz *Head (Visible Male)*.

| Implementierung | System 1 | System 2 |
|-----------------|----------|----------|
| Back-To-Front   | 1        | 3        |
| Front-To-Back   | 2        | 8        |

Tabelle 5.19: Back-To-Front vs. Front-To-Back. Ergebnisse in FPS. Volumendatensatz *Schwein*.

Strahlterminierung kann bei einer CPU-basierenden Realisierung in der Regel ein Beschleunigungsfaktor zwischen 1,3 und 2,2 erzielt werden [Lev90]. Bei der im Rahmen dieser Arbeit erstellten GPU-basierenden Realisierung konnten vergleichbare Beschleunigungsfaktoren festgestellt werden.

### 5.3 Fehlerquellen

**Textur Kompression** Für 8 bit Volumendaten bietet sich die qualitativ hochwertige LATC1-Texturkompression an, um den Speicherbedarf um die Hälfte zu reduzieren. Jedoch ist zu beachten, dass gleichzeitig die Implementierung für das hoch und herunterladen der Daten zu und von der GPU aufwändiger wird. Sobald die OpenGL *GL\_NV\_texture\_compression\_vtc*-Erweiterung vorhanden ist, muss beachtet werden, dass die Blöcke der komprimierten Volumendaten vor dem Hochladen auf die Grafikkarte gemäß der Volume Texture Compression (VTC)-Spezifikation umsortiert werden müssen. Ein nicht beachten führt zu dem in Abbildung 5.4 sichtbaren Ergebnis.

**Division durch Null** Das praktische Umsetzen von komplexeren theoretischen Beleuchtungsmodellen kann sich als aufwändig und fehleranfällig erweisen. Eine Division durch Null kann in Randfällen vorkommen. Beim Rendern in einen R8G8B8-Bildpuffer kann dies unter

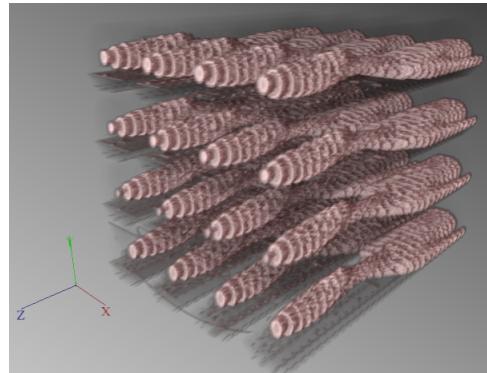


Bild 5.4: VTC ohne Anpassung der Kompressionsblöcke. Volumendatensatz *Thomas*.

Umständen nicht sofort auffallen, da hier kein Not a Number (NaN) im Bildpuffer gespeichert werden kann. Beim Verwenden eines Fließkomma-Bildpuffer, wie dies beispielsweise bei HDR der Fall ist, wird NaN mit abgespeichert. Dies bedeutet, das NaN sich durch alle späteren Renderschritte durchzieht und in unberechenbaren Bildfehlern resultieren kann. Der Ort der Fehlerquelle ist in diesem Fall nicht immer einfach feststellbar und kann nicht selten nur über zeitaufwändige Analysen ermittelt werden. Dabei ist zu beachten, das je nach GPU-Treiber ein einzelner Pixel mit NaN beispielsweise zu einem größeren schwarzen oder weißen Block werden kann. Bei der Shader-Programmierung ist aufgrund dessen unbedingt darauf zu achten, das niemals eine Division durch Null auftreten kann. So ist beispielsweise die Cook-Torrance-Beleuchtungsmodell Implementierung von [Had06, S. 121] offensichtlich fehlerhaft: Es kommt zu sichtbaren Beleuchtungsanomalien und ebenfalls NaN's. [Mik09] analysiert dieses Beleuchtungsmodell und präsentiert eine praktisch einsetzbare Lösung.

# Kapitel 6

## Zusammenfassung

Das Feld des Volumen-Renderings befasst sich mit der visuellen Darstellung von Volumen-basierenden Objekten ohne eindeutig definierbare Oberfläche. Traditionelle Einsatzgebiete von Volumen-Rendering finden sich in der Medizin und Wissenschaft, sowie für visuelle Effekte in der Filmindustrie. Aufgrund der Datenmengen und den rechenaufwändigen Darstellungsme-thoden war der Einsatz von Volumen-Rendering auf handelsüblicher Hardware lange Zeit für interaktive Computergrafik nicht sinnvoll nutzbar. Dies änderte sich mit der Verfügbarkeit von frei programmierbaren hochleistungsfähigen Grafikkarten. Angetrieben von den stets wachsenden Ansprüchen an moderne Videospiele, wächst die Leistungsfähigkeit von GPUs schneller als die von CPUs. Die Darstellung volumetrischer Effekte wie beispielsweise Feuer, Nebel, Rauch und Wolken über Volumen-Rendering ist heutzutage eine praktisch nutzbare Lösung.

Polygonbasierende 3-D-Modelle werden über Vertices, Indices und 2-D-Texturen beschrie-ben und über Rasterisierung visualisiert. Volumen-basierende Daten hingegen bestehen aus ein-zelnen Volumen-Elementen, den sogenannten Voxeln. Diese sind vergleichbar zu den Bildpunk-ten in einer 2-D-Rastergrafik, jedoch besitzt die Rastergrafik eine dritte Dimension. Die unmit-telbaren Quellen von Volumendaten sind zahlreich und reichen von medizinischen Scannern bis hin zu Simulationen. Ein 3-D-Datensatz kann auch das Ergebnis einer Konvertierung polygon-basierender Modelle sein. Umgekehrt lassen sich Volumen-basierende Daten über oberflächen-extrahierende Methoden in eine polygonbasierende Repräsentation überführen. Polygone sind unmittelbar von einer Grafikkarte darstellbar. Für medizinische Datensätze oder volumetrische Effekte ist eine reine Oberflächendarstellung unzureichend.

In der Literatur werden zahlreiche Verfahren zur direkten Visualisierung von Volumendaten beschrieben. Der bekannteste und einer der ältesten Vertreter ist das Volumen-Raycasting, dass sich auf Grafikkarten effizient über Shader einsetzen lässt. Für jeden Bildpunkt wird ein Strahl generiert, durch einen Strahlverfolgung durch das Volumen wird der Farbwert des Bildpunktes

berechnet. Für die Strahlverfolgung, sowie zur Berechnung der Farbe eines Abtastpunktes entlang des Strahls existieren zahlreiche Ansätze. Die Anforderungen an das Volumen-Rendering reichen von einer interaktiven bis hin zu einer qualitativ hochwertigen Darstellung ohne Echtzeitanforderungen. Im medizinischen Umfeld ist eine wahrheitsgetreue Visualisierung von entscheidender Bedeutung. In der Unterhaltungsbranche hingegen sind ästhetische Faktoren ausschlaggebend. Diese Arbeit soll aufzeigen, dass die zahlreichen Anforderungen durch ein skalierbares System abgedeckt werden können.

Im Rahmen dieser Arbeit wurde als praktischer Teil die Open-Source 3-D-Engine Pixel-Light um einen Volumen-Renderer erweitert. Szenen werden über einen hierarchischen Szenengraphen abgebildet und bestehen aus Knoten und Container. Neue Knotentypen für Volumen sowie Schnittelemente wurden hinzugefügt. Die Volumendaten werden unabhängig von der Szene verwaltet, hierdurch wird es möglich ein Volumen mehrmals über verschiedene Volumenknoten in der Szene zu platzieren. Dies ermöglicht eine zeitgleiche Visualisierung von Volumendaten anhand unterschiedlicher Konfigurationen.

In Abhängigkeit der Art der vorliegenden 3-D-Daten, der gewünschten Darstellung sowie den Anforderungen an die Bildwiederholrate und Bildqualität sind verschiedene Kombinationen von Algorithmen nötig. Für einen vielseitig einsetzbaren Volumen-Renderer ist das manuelle Schreiben eines Shaders ist somit nicht sinnvoll. Aufgrund dessen wurde der Shader in einzelne Shaderfunktionen zerlegt, die durch ein Konfigurationssystem automatisch zu einem lauffähigen Shader zusammengestellt werden. Durch die kontinuierliche Weiterentwicklung von Grafikkarten und Grafikschnittstellen können ehemals umständliche Realisierungen durch einfache und verständliche ersetzt werden. Unter Zuhilfenahme echter Schleifen in Fragmentshadern müssen Renderschritte nicht mehr künstlich in mehrere Renderschritte unterteilt werden. Trotzdem lassen sich dynamische Verzweigungen sowie frühzeitiges Abbrechen des Fragmentshaders realisieren.

Die Aufgabe der Strahlgenerierung besteht darin, über eine Hilfsgeometrie Fragmente zu erzeugen. Ein Fragment entspricht einem Strahl durch das Volumen. Für jedes Fragment wird ein Fragmentshader durchlaufen der wiederum Shaderfunktionen aufruft. Jede Shaderfunktion besitzt eine wohldefinierte Schnittstelle sowie eine vorgegebene Aufgabenstellung zu deren Lösung weitere Shaderfunktionen hinzugezogen werden können. Bei der Umsetzung neuer Algorithmen für bestimmte Teilschritte des Volumen-Renderings kann somit der Schwerpunkt auf die unmittelbare Aufgabenstellung gelegt werden.

Vor der eigentlichen Strahlverfolgung ist ein optionaler Strahlschnitt möglich. Anhand des Tiefenpuffers findet eine Stahlverkürzung statt um eine korrekte Interaktion zwischen nicht transparenten Objekten in der Szene zu ermöglichen. Über eine oder mehrere Schnittebenen lassen sich Teile des Volumen ausblenden.

Die Wahl der Abtastrate hat einen großen Einfluss auf die Bildwiederholrate. Für eine interaktive Darstellung darf die Abtastrate nicht zu hoch sein. Durch eine geringe Abtastrate entstehen allerdings erkennbare kreisförmige und wandernde Muster, die sogenannten Holz-Korn-Artefakte. Indem die Strahlstartposition geringfügig entlang der Strahlrichtung verschoben wird, entsteht ein zufälliges künstliches Rauschen. Die kreisförmigen Muster sind nicht mehr sichtbar.

Nach der Strahlgenerierung und Modifikation des Strahls findet die Strahlverfolgung statt. Entlang der Strahlrichtung werden in Abhängigkeit der Abtastrate zahlreiche Abtastpunkte verarbeitet. Die Wahl des Algorithmus für die Traversierung ist ausschlaggebend für die resultierende Visualisierung. Für die Darstellung von Isoflächen wird die Strahlverfolgung abgebrochen, sobald ein Skalarwert auftritt der größer ist als ein gegebener Schwellwert. MIP hingegen hat eine Röntgenbild-Darstellung zum Ziel, während mit Compositing eine Vielzahl an optischen Zielen verfolgt werden kann.

Für einen Schnitt innerhalb des Volumens wird jeder Abtastpunkt daraufhin überprüft, ob sich dieser innerhalb einer oder mehrerer Schnittelemente befindet. Hierdurch lassen sich Abtastpunkte innerhalb einer Würfelgeometrie ausblenden. Durch Invertierung des Schnittes wird alles außerhalb der Schnittelemente ausgeblendet.

Die Ermittlung des Skalarwertes an der aktuellen Position entlang des Strahls kann auf verschiedenen Wegen erfolgen. In der Regel liegen diskretisierte Volumendaten vor, die anhand eines Filters rekonstruiert werden. Für 3-D-Texturen bieten Grafikkarten eine nächster-Nachbar-Filterung sowie eine bilineare Filterung. Für qualitativ hochwertige Rekonstruktionsfilter kann eine Faltung über kubische B-Splines erfolgen. Bei prozeduralen Volumendaten hingegen wird der aktuelle Skalarwert über mathematische Funktionen berechnet.

Um eine für den Menschen interpretierbare Farbe zu ermitteln, wird pro Abtastpunkt ein Shading ausgeführt. Für das Shading existieren zahlreiche Ansätze. Diese reichen von einer photorealistischen Darstellung bis hin zu einer Comic ähnlichen Visualisierung. Eine Shading-Strategie wiederum besteht in der Regel aus einzelnen Teilschritten wie einer Klassifikation, Gradientenschätzung sowie einer Beleuchtungsberechnung, um die Helligkeit an der aktuellen Position im Volumen zu berechnen. Indem Lichtquellen und Schatten miteinbezogen werden, lässt sich die Tiefenwahrnehmung verbessern. Das Ergebnis der Klassifikation des Abtastpunktes wird mit der Helligkeit multipliziert und als resultierende Farbe des Shading zurückgeliefert.

Die Bedeutung eines Skalarwertes entlang des Strahls hängt von der Modalität der Volumendaten ab. Bei CT-Daten repräsentiert ein Skalarwert einen gemessenen Dichtewert. Die Werte von Luft, Fettgewebe oder Knochen sind durch die Hounsfield-Skala bekannt. So besitzen Knochen eine höhere Dichte als das umliegendes Gewebe. Bei einer Darstellung ohne weitere Verarbeitung, ergibt sich ein Graustufenbild. Für eine jeweilige Aufgabenstellung relevante

Merkmale sind schlecht bis nicht zu erkennen. Abhilfe schafft eine Klassifikation. Skalarwerte können unter Zuhilfenahme eines Schwellwertes vollständig ausblendet werden. Praktisch sinnvoller ist die Verwendung einer Transferfunktion. Einem Skalarwert wird hierüber eine Farbe sowie eine Opazität zugewiesen.

Lokale Beleuchtungsmodelle, wie beispielsweise das von Blinn-Phong, benötigen in der Regel einen Normalenvektor der die Ausrichtung der zu beleuchtenden Oberfläche spezifiziert. Bei Volumen existiert keine explizite Oberfläche. Aufgrund dessen findet eine Gradientenschätzung statt. Das Verfahren der zentralen Differenzen stellt einen Kompromiss zwischen Bildqualität und Bildwiederholrate dar. Zum Ermitteln eines Gradienten für die gegebene Position auf dem Strahl werden die unmittelbaren Nachbarn entlang jeder Achse in jeweils positiver und negativer Richtung hinzugezogen. Dies resultiert in sechs Zugriffen auf die Volumendaten. Der Gradient wird normalisiert und anschließend bei der Beleuchtung als Normalenvektor verwendet. Im einfachen Lambert-Beleuchtungsmodell findet ein Skalarprodukt zwischen einem normalisierten Oberflächenvektor sowie einen gegebenen normalisierten Lichtrichtungsvektor statt um die Helligkeit an der aktuellen Position im Volumen zu berechnen.

Über Experimente wurde ermittelt welche Bildwiederholraten sich durch unterschiedliche Konfigurationen des erstellten Volumen-Renderers erzielen lassen. Bei einem  $128 \times 256 \times 256$  Voxel großen Datensatz wurden beim variieren von Implementierungen von Shaderfunktionen 17-219 FPS gemessen. Die gleichen Versuche wurden mit einem  $512 \times 512 \times 1743$  Voxel großen Datensatz wiederholt und resultierte in 2-29 FPS. Unabhängig vom vorgestellten Shadersystem wurde in einem Versuch die Bildauflösung schrittweise Verändert. Dies resultierte in einer FPS von 24 bis hin zu 475. Einen großen Einfluss auf die Bildqualität und Bildwiederholrate hat ebenfalls die Abtastrate, die in einem Experiment variiert wurde um 82 bis hin zu 820 FPS zu erzielen.

Es wurde gezeigt das im realisierten Volumen-Rendering System eine große Bandbreite an Konfigurationsmöglichkeiten vorhanden ist, die in Abhängigkeit der Anforderungen feingranular Justiert werden kann. Weitere Implementierungen einzelner Shaderfunktionen lassen sich hinzufügen, ohne das sämtliche Shader angepasst werden müssen.

# Kapitel 7

## Ausblick

Das realisierte Volumen-Rendering System verwendet Shader und wurde als separates und somit austauschbares Plugin realisiert. Denkbar wäre ein weiteres Volumen-Rendering Plugin, das CUDA oder OpenCL nutzt. Zahlreiche Shaderbausteine könnten portiert werden, ein direkter Leistungsvergleich zwischen der Realisierung über Shader und einer GPGPU-Hochsprache wie beispielsweise OpenCL wäre interessant.

[Roe08] stellt Multi-Volumen-Rendering für Textur-Slicing vor. Das gleichzeitige Rendern von mehreren Volumen funktioniert beim im Rahmen dieser Arbeit erstellten Volumen-Raycasting, solange die einzelnen Volumenknoten räumlich voneinander getrennt sind. Sobald eine räumliche Überschneidung stattfindet, kann bei der Sichtbarkeitsbestimmung von Szenenknoten keine eindeutige Renderreihenfolge ermittelt werden. Um dieses Problem zu lösen, müssten die sich überschneidenden Volumenknoten in einem gemeinsamen Volumen-Raycasting Shader verarbeitet werden. Dies würde in einer gemeinsamen Hilfsgeometrie für die Strahlenerzeugung resultieren, die beide Volumen umschließt. Anschließend gäbe es eine Strahlverfolgung, die jeweils die Shading-Shaderfunktion für jedes beteiligte Volumen aufruft. Das Ergebnis der getrennten Berechnungen müsste kombiniert und anschließend wie gehabt beim Strahlverfolgungs-Compositing gemischt werden. Prinzipiell bliebe folglich der Ablauf und die Implementierung der im Rahmen dieser Arbeit vorgestellten Shaderfunktionen gleich, beim dynamischen erzeugen des Shaders müssten Funktionsnamen sowie Shaderparameter automatisch angepasst werden, damit es zu keinen Namenskonflikten in den vom Shader gleichzeitig verarbeiteten mehreren Volumen kommt.

Um den Rahmen dieser Arbeit nicht zu sprengen, wurden nur einfache Beleuchtungsmodelle gewählt. Ebenfalls wird nur eine einzige direktionale Lichtquelle berücksichtigt und die Beleuchtungsfunktion wurde nicht weiter zerlegt. Beim Shading könnten ebenfalls wieder Templates eingesetzt werden um mehrere Lichtquellen zu unterstützen, mit jeweils unterschiedlichen

Lichttypen sowie spezielleren Lichteigenschaften. Das Beleuchtungsmodell hingegen könnte in Ambient, Diffuse und Glanzlicht aufgeteilt werden um ein besseres mischen von Beleuchtungsmodellen zu ermöglichen. Subsurfacescattering und Schatten könnten die Bildqualität weiter verbessern.

Das Themengebiet der Volumen-Animation wurde in Kapitel 2 nur kurz am Rande erwähnt. Die im Rahmen dieser Arbeit behandelte Volumen-Renderer Realisierung ist prinzipiell 4-D fähig. Auf der CPU Seite werden keine statischen Datenstrukturen erzeugt, die aufwändig aktualisiert werden müssten sobald sich die Volumendaten ändern. Ebenfalls wurde darauf geachtet, dass alle relevanten Parameter zur Laufzeit interaktiv veränderbar sind. Der Vollständigkeit halber wurde an den entsprechenden Stellen auf Optimierungsmöglichkeiten hingewiesen, die sich bei statischen Volumendaten ergeben. Traditionelle Grafikschnittstellen, wie beispielsweise OpenGL, sind nicht darauf ausgelegt 3-D-Daten direkt zu Rendern. Um eine 3-D-Textur dynamisch zur Laufzeit zu verändern, müssen die einzelnen z-Schichten umständlich und ineffizient in separaten Renderschritten gefüllt werden. Ein kontinuierliches verändern der Volumendaten über die CPU mit anschließendem aktualisieren der 3-D-Textur im Grafikkartenspeicher ist prinzipiell möglich, allerdings stellt die Datentransferrate zwischen CPU und GPU ein nicht zu vernachlässigender Flaschenhals da. Dank der GPGPU-Hochsprachen OpenCL und CUDA existieren weitere GPU-Programmierschnittstellen, die es ermöglichen Texturen als einfachen Puffer anzusprechen. Eine interessante Erweiterung könnte darin bestehen, über OpenCL dynamisch den 3-D-Texturinhalt, beispielsweise durch eine Flüssigkeitssimulation aktualisieren zu lassen, während der hier vorstellte Volumen-Renderer OpenGL zur Visualisierung der Volumendaten einsetzt wird.

In [Sme09] stellen Intel-Mitarbeiter einen Vergleich zwischen Volumen-Renderer Implementierungen auf, die jeweils im speziellen für CPU sowie GPU optimiert wurden. Die GPU-Lösung war hierbei nicht deutlich schneller als die CPU-Lösung. In [Lee10b] versuchen ebenfalls Intel-Mitarbeiter den Mythos zu beseitigen der besagt, dass GPU beschleunigte Algorithmen generell deutlich schneller als CPU-Realisierungen seien. Diese Aussage sei primär darauf zurückzuführen, dass bei den meisten Leistungsvergleichen die CPU-Realisierung nicht ausreichend optimiert worden sei. Während derartige, von Intel-Mitarbeitern durchgeführte Studien kritisch betrachtet werden müssen, ist unabhängig davon generell der Trend zu beobachten, dass CPU- und GPU-Architekturen kontinuierlich weiter zusammenwachsen. Die ersten Volumen-Renderers Raycaster liefen auf CPU's, aktuelle Realisierungen bevorzugen GPU's, langfristig besteht eine Chance, dass CPU-basierende Realisierungen wieder attraktiver werden.

# **Anhang A**

## **Verwendete Datensätze**

### **Crossed Rods**

- Von <http://www9.informatik.uni-erlangen.de/External/vollib/>, Direktlink <http://www9.informatik.uni-erlangen.de/External/vollib/Cross.pvm>
- $64 \times 64 \times 64$  Voxel (262 144 Voxel), 8 bit pro Voxel
- 256 KiB Speicherverbrauch, LATC1 komprimiert 128 KiB
- 292 KiB Speicherverbrauch bei 6 zusätzlichen Mipmaps, LATC1 komprimiert 146 KiB

### **Neghip**

- Von <http://www9.informatik.uni-erlangen.de/External/vollib/>, Direktlink <http://www9.informatik.uni-erlangen.de/External/vollib/Neghip.pvm>
- $64 \times 64 \times 64$  Voxel (262 144 Voxel), 8 bit pro Voxel
- 256 KiB Speicherverbrauch, LATC1 komprimiert 128 KiB
- 292 KiB Speicherverbrauch bei 6 zusätzlichen Mipmaps, LATC1 komprimiert 146 KiB

### **Hydrogen**

- Von <http://www9.informatik.uni-erlangen.de/External/vollib/>, Direktlink <http://www9.informatik.uni-erlangen.de/External/vollib/Hydrogen.pvm>
- $128 \times 128 \times 128$  Voxel (2 097 152 Voxel), 8 bit pro Voxel
- 2 MiB (2048 KiB) Speicherverbrauch, LATC1 komprimiert 1 MiB (1024 KiB)

- 2 MiB (2340 KiB) Speicherverbrauch bei 7 zusätzlichen Mipmaps, LATC1 komprimiert 1 MiB (1170 KiB)

### **Head (Visible Male)**

- Von <http://www9.informatik.uni-erlangen.de/External/vollib/>, Direktlink <http://www9.informatik.uni-erlangen.de/External/vollib/VisMale.pvm>
- $128 \times 256 \times 256$  Voxel (8 388 608 Voxel), 8 bit pro Voxel
- 8 MiB (8192 KiB) Speicherverbrauch, LATC1 komprimiert 4 MiB (4096 KiB)
- 9 MiB (9362 KiB) Speicherverbrauch bei 8 zusätzlichen Mipmaps, LATC1 komprimiert 4 MiB (4681 KiB)

### **Engine**

- Von <http://www9.informatik.uni-erlangen.de/External/vollib/>, Direktlink <http://www9.informatik.uni-erlangen.de/External/vollib/Engine.pvm>
- $256 \times 256 \times 256$  Voxel (16 777 216 Voxel), 8 bit pro Voxel
- 16 MiB (16 384 KiB) Speicherverbrauch, LATC1 komprimiert 8 MiB (8192 KiB)
- 18 MiB (18 724 KiB) Speicherverbrauch bei 8 zusätzlichen Mipmaps, LATC1 komprimiert 9 MiB (9362 KiB)

### **Thomas**

- CT-Scan eines Kommilitonen
- $512 \times 512 \times 117$  Voxel (30 670 848 Voxel), 8 bit pro Voxel
- 29 MiB (29 952 KiB) Speicherverbrauch, LATC1 komprimiert 14 MiB (14 976 KiB)
- 33 MiB (34 191 KiB) Speicherverbrauch bei 9 zusätzlichen Mipmaps, LATC1 komprimiert 16 MiB (17 095 KiB)

## Schwein

- Von Prof. Dr.-Ing. Frank Deinzer zu Testzwecken bereitgestellter Datensatz
- $512 \times 512 \times 1743$  Voxel (456 916 992 Voxel), 8 bit pro Voxel (Original: 12 bit)
- 435 MiB (446 208 KiB) Speicherverbrauch, LATC1 komprimiert 217 MiB (223 104 KiB)
- 497 MiB (509 903 KiB) Speicherverbrauch bei 10 zusätzlichen Mipmaps, LATC1 komprimiert 248 MiB (254 951 KiB)

**Frei zugängliche Datensätze im Internet** Eine kleine Auswahl an Internet Seiten mit frei zugänglichen Volumendaten:

- <http://volvis.org/> RAW-Format
- <http://openqvis.sourceforge.net/> RAW-Format mit zusätzlicher ASCII-Textdatei
- <http://www9.informatik.uni-erlangen.de/External/vollib/> im PVM-Format
- <http://pubimage.hcuge.ch:8080/> DICOM-Format



## Anhang B

# Direct3D Shadermodell vs. OpenGL-Versionen vs. GLSL-Versionen

Sofern nicht anders angegeben, wurden die Shaderbeispiele in GLSL Version 3.3 [Kes] verfasst, dies entspricht dem Direct3D Shadermodell 4 von DirectX 10. Tabelle B.1 stellt die, in der Literatur häufiger genannten, Direct3D Shadermodelle den GLSL-Versionen und diese wiederum den OpenGL-Versionen gegenüber. Aufgrund der unterschiedlichen Betrachtungsweisen ist eine exakte Zuordnung auf eine andere Versionierung nicht immer möglich.

| GLSL | OpenGL | Direct3D Shadermodell und Kommentar   |
|------|--------|---|
| 110  | 2.0    |   |
| 120  | 2.1    |   |
| 130  | 3.0    |   |
| 140  | 3.1    |   |
| 150  | 3.2    | Annähernd Feature gleich zu 4 <sup>1</sup> , <i>Geometrie-Shader</i> hinzugefügt                            |
| 330  | 3.3    | 4 (DirectX Version 10) <sup>1</sup>   |
| 400  | 4.0    | <i>Tessellation-Control</i> <sup>2</sup> und <i>Tessellation-Evaluation</i> <sup>3</sup> Shader hinzugefügt |
| 410  | 4.1    |   |
| 420  | 4.2    | 5 (DirectX Version 11) <sup>1</sup>   |

<sup>1</sup> Information entnommen aus [http://www.opengl.org/wiki/Detecting\\_the\\_Shader\\_Model](http://www.opengl.org/wiki/Detecting_the_Shader_Model)

<sup>2</sup> *Hull Shader* in DirectX 11

<sup>3</sup> *Domain Shader* in DirectX 11

Tabelle B.1: Direct3D Shadermodell vs. OpenGL-Versionen vs. GLSL-Versionen.

Aufgrund der großen Anzahl unterschiedlicher GLSL-Versionen ist es von entscheidender Bedeutung das ein Shader die verwendete Shaderversion definiert. Die verwendete Syntax muss hierbei Punktgenau der jeweiligen Spezifikation der gewählten Version entsprechen. Ein Miss-

## *96 ANHANG B. DIRECT3D SHADERMODELL VS. OPENGL-VERSIONEN VS. GLSL-VERSIONEN*

achten kann zur Folge haben, dass ein Shader auf einem System läuft, während der gleiche Shader auf einem anderen System zu Shadercompiler Fehlern führt.

# Literaturverzeichnis

- [Ack98] Ackerman, M.: *The Visible Human Project, Proceedings of the IEEE*, Bd. 86, Nr. 3, März 1998, S. 504–511.
- [Adva] Advanced Micro Devices Inc. ATI: *ATI OpenGL Programming and Optimization Guide*, 2007, Online: [http://developer.amd.com/media/gpu\\_assets/ATI\\_OpenGL\\_Programming\\_and\\_Optimization\\_Guide.pdf](http://developer.amd.com/media/gpu_assets/ATI_OpenGL_Programming_and_Optimization_Guide.pdf), Letzter Zugriff: 25.04.2012.
- [Advb] Advanced Micro Devices Inc. ATI: *ATI Radeon<sup>TM</sup>HD 2000 programming guide*, 2007, Online: [http://developer.amd.com/media/gpu\\_assets/ATI\\_Radeon\\_HD\\_2000\\_programming\\_guide.pdf](http://developer.amd.com/media/gpu_assets/ATI_Radeon_HD_2000_programming_guide.pdf), Letzter Zugriff: 25.04.2012.
- [Ale11] Alexiadis, D. S.; Kelly, P.; Daras, P.; O'Connor, N. E.; Boubekeur, T.; Moussa, M. B.: *Evaluating a dancer's performance using kinect-based skeleton tracking*, in *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, ACM, New York, NY, USA, 2011, S. 659–662.
- [AM08] Akenine-Möller, T.; Haines, E.; Hoffman, N.: *Real-Time Rendering 3rd Edition*, A. K. Peters, Ltd., Natick, MA, USA, 2008.
- [Bin11] Binks, D.: *Dynamic Resolution Rendering*, 2011, Online: <http://software.intel.com/en-us/articles/vcsource-samples-dynamic-resolution-rendering/>, Letzter Zugriff: 25.04.2012.
- [Bli77] Blinn, J. F.: *Models of light reflection for computer synthesized pictures*, *SIGGRAPH Comput. Graph.*, Bd. 11, Nr. 2, 1977, S. 192–198.
- [Bre09] Breidt, M.: *Be gamma correct! v1.3*, 2009, Online: [http://scripts.breidt.net/gamma\\_correct\\_v12.pdf](http://scripts.breidt.net/gamma_correct_v12.pdf), Letzter Zugriff: 25.04.2012.
- [Bro01] Bronstein, I. N.; Semendjajew, K. A.; Musiol, G.; Mühlig, H.: *Taschenbuch der Mathematik*, Harri Deutsch, 5. Ausg., 2001.

- [Bru09] Bruckner, S.; Gröller, M. E.: *Instant Volume Visualization using Maximum Intensity Difference Accumulation*, *Computer Graphics Forum*, Bd. 28, Nr. 3, 2009, S. 775–782.
- [Bus] Buschmann, S.; Ofenberg, C.: *PixelLight, free open-source 3D application framework*, Online: <http://www.pixellight.org/>, Letzter Zugriff: 25.04.2012.
- [Cab94] Cabral, B.; Cam, N.; Foran, J.: *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*, in *Proceedings of the 1994 symposium on Volume visualization*, VVS '94, ACM, New York, NY, USA, 1994, S. 91–98.
- [Con11] Congote, J.; Segura, A.; Kabongo, L.; Moreno, A.; Posada, J.; Ruiz, O.: *Interactive visualization of volumetric data with WebGL in real-time*, in *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, ACM, New York, NY, USA, 2011, S. 137–146.
- [Coo82] Cook, R. L.; Torrance, K. E.: *A Reflectance Model for Computer Graphics*, *ACM Trans. Graph.*, Bd. 1, Nr. 1, 1982, S. 7–24.
- [Cor11] Corcoran, A.; Dingliana, J.: *Time-Critical Ray-Cast Direct Volume Rendering*, Trinity College Dublin, 2011.
- [Dre88] Drebin, R. A.; Carpenter, L.; Hanrahan, P.: *Volume rendering*, *SIGGRAPH Comput. Graph.*, Bd. 22, Nr. 4, 1988, S. 65–74.
- [Eng01] Engel, K.; Kraus, M.; Ertl, T.: *High-quality pre-integrated volume rendering using hardware-accelerated pixel shading*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '01, ACM, New York, NY, USA, 2001, S. 9–16.
- [Eng04] Engel, K.; Hadwiger, M.; Kniss, J. M.; Lefohn, A. E.; Salama, C. R.; Weiskopf, D.: *Real-time volume graphics*, in *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, ACM, New York, NY, USA, 2004.
- [Fan08] Fan, Z.; Mei, X.: *Real-time Medical Image Volume Rendering Based on GPU Accelerated Method*, in *Computational Intelligence and Design, 2008. ISCID '08. International Symposium on*, Bd. 2, Oktober 2008, S. 30–33.
- [Fer03] Fernando, R.; Kilgard, M. J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

- [Fer04] Fernando, R.: *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, Chapter 39. Volume Rendering Techniques*, Pearson Higher Education, 2004.
- [Gag01] Gagvani, N.; Silver, D.: *Animating volumetric models, Graph. Models*, Bd. 63, Nr. 6, 2001, S. 443–458.
- [Gho05] Ghosh, A.; Trentacoste, M.; Heidrich, W.: *Volume Rendering for High Dynamic Range Displays*, in *IN EG/IEEE VGTC WORKSHOP ON VOLUME GRAPHICS 2005*, 2005, S. 91–98.
- [Gre05] Green, S.: *Volume Rendering in Games, Talk at Game Developers Conference (GDC)*, 2005, Online: [http://http.download.nvidia.com/developer/presentations/2005/GDC/Sponsored\\_Day/GDC\\_2005\\_VolumeRenderingForGames.pdf](http://http.download.nvidia.com/developer/presentations/2005/GDC/Sponsored_Day/GDC_2005_VolumeRenderingForGames.pdf), Letzter Zugriff: 25.04.2012.
- [Guo11] Guo, H.; Mao, N.; Yuan, X.: *WYSIWYG (What You See is What You Get) Volume Visualization, IEEE Transactions on Visualization and Computer Graphics*, Bd. 17, Nr. 12, 2011, S. 2106–2114.
- [Gut02] Guthe, S.; Wand, M.; Gonser, J.; Straßer, W.: *Interactive rendering of large volume data sets*, in *Proceedings of the conference on Visualization '02*, VIS '02, IEEE Computer Society, Washington, DC, USA, 2002, S. 53–60.
- [Had06] Hadwiger, M.; Kniss, J. M.; Rezk-salama, C.; Weiskopf, D.; Engel, K.: *Real-time Volume Graphics*, A. K. Peters, Ltd., Natick, MA, USA, 2006.
- [Had09] Hadwiger, M.; Ljung, P.; Salama, C. R.; Ropinski, T.: *Advanced illumination techniques for GPU-based volume raycasting*, in *ACM SIGGRAPH 2009 Courses*, SIGGRAPH '09, ACM, New York, NY, USA, 2009, S. 2:1–2:166.
- [Har03] Harris, M. J.; Baxter, W. V.; Scheuermann, T.; Lastra, A.: *Simulation of cloud dynamics on graphics hardware*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, Eurographics Association, Aire-la-Ville, Switzerland, 2003, S. 92–101.
- [Her10] Hernell, F.; Ljung, P.; Ynnerman, A.: *Local Ambient Occlusion in Direct Volume Rendering, IEEE Transactions on Visualization and Computer Graphics*, Bd. 16, Nr. 4, 2010, S. 548–559.

- [Huc10] Hucko, M.; Vanek, M.; Sramek, M.: *The VRE volume rendering engine*, in *Proceedings of the 26th Spring Conference on Computer Graphics*, SCCG '10, ACM, New York, NY, USA, 2010, S. 61–68.
- [IR-] IR-Entertainment Ltd.: *Infinite Kopf Scan*, Online: <http://www.ir-ltd.net/> royalty-free-high-quality-scan, Letzter Zugriff: 25.04.2012.
- [Kau94] Kaufman, A. E.: *Voxels as a Computational Representation of Geometry*, in *The Computational Representation of Geometry. SIGGRAPH '94 Course Notes*, 1994, S. 45.
- [Kay86] Kay, T. L.; Kajiya, J. T.: *Ray tracing complex scenes*, *SIGGRAPH Comput. Graph.*, Bd. 20, Nr. 4, 1986, S. 269–278.
- [Kes] Kessenich, J.; Baldwin, D.; Rost, R.: *The OpenGL® Shading Language, Language Version: 3.30*, Online: <http://www.opengl.org/registry/>, Letzter Zugriff: 25.04.2012.
- [Kil05] Kilgariff, E.; Fernando, R.: *The GeForce 6 series GPU architecture*, in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, ACM, New York, NY, USA, 2005.
- [Kin02] Kindlmann, G.: *Transfer Functions in Direct Volume Rendering: Design, Interface, Interaction, Direct*, Bd. d, 2002, S. 6.
- [Kni01] Kniss, J.; Kindlmann, G.; Hansen, C.: *Multi-dimensional transfer functions and direct manipulation widgets*, in *Proceedings of the conference on Visualization '01*, VIS '01, IEEE Computer Society, Washington, DC, USA, 2001, S. 255–262.
- [Koh03] Koh, W.; McCormick, B. H.: Juli 2003.
- [Kra06] Kratz, A.; Hadwiger, M.; Fuhrmann, A.; Splechtna, R.; Bühler, K.: *GPU-Based High-Quality Volume Rendering For Virtual Environments*, 2006.
- [Kro11] Kroes, T.; Post, F. H.; Botha, C. P.: *Interactive direct volume rendering with physically-based lighting*, Preprint 2011-11, Delft University of Technology, 2011.
- [Krü03] Krüger, J.; Westermann, R.: *Acceleration techniques for GPU-based volume rendering*, in *Visualization, 2003. VIS 2003. IEEE*, Oktober 2003, S. 287–292.
- [Kub12] Kubisch, C.; Glaßer, S.; Neugebauer, M.; Preim, B.: *Vessel Visualisation with Volume Rendering*, in et al., L. L. (Hrsg.): *Visualization in Medicine and Life Sciences II, Mathematics and Visualization (Workshop VMLS 2009)*, Springer Verlag, 2012, S. 109–134.

- [Lac94] Lacroute, P.; Levoy, M.: *Fast volume rendering using a shear-warp factorization of the viewing transformation*, in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, ACM, New York, NY, USA, 1994, S. 451–458.
- [Lai10] Laine, S.; Karras, T.: *Efficient sparse voxel octrees*, in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, ACM, New York, NY, USA, 2010, S. 55–63.
- [Lee10a] Lee, B.; Yun, J.; Seo, J.; Shim, B.; Shin, Y.-G.; Kim, B.: *Fast High-Quality Volume Ray Casting with Virtual Samplings*, *IEEE Transactions on Visualization and Computer Graphics*, Bd. 16, Nr. 6, 2010, S. 1525–1532.
- [Lee10b] Lee, V. W.; Kim, C.; Chhugani, J.; Deisher, M.; Kim, D.; Nguyen, A. D.; Satish, N.; Smelyanskiy, M.; Chennupaty, S.; Hammarlund, P.; Singhal, R.; Dubey, P.: *Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU*, *SIGARCH Comput. Archit. News*, Bd. 38, Nr. 3, 2010, S. 451–460.
- [Len01] Lengyel, J.; Praun, E.; Finkelstein, A.; Hoppe, H.: *Real-time fur over arbitrary surfaces*, in *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, ACM, New York, NY, USA, 2001, S. 227–232.
- [Lev88] Levoy, M.: *Display of surfaces from volume data*, *Computer Graphics and Applications, IEEE*, Bd. 8, Nr. 3, Mai 1988, S. 29–37.
- [Lev90] Levoy, M.: *Efficient ray tracing of volume data*, *ACM Trans. Graph.*, Bd. 9, Nr. 3, 1990, S. 245–261.
- [Lor87] Lorensen, W. E.; Cline, H. E.: *Marching cubes: A high resolution 3D surface construction algorithm*, *SIGGRAPH Comput. Graph.*, Bd. 21, Nr. 4, 1987, S. 163–169.
- [Lot09] Lottes, T.: *FXAA*, 2009, Online: [http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA\\_WhitePaper.pdf](http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf), Letzter Zugriff: 25.04.2012.
- [Lum01] Lum, E. B.; Ma, K. L.; Clyne, J.: *Texture hardware assisted rendering of time-varying volume data*, in *Proceedings of the conference on Visualization '01*, VIS '01, IEEE Computer Society, Washington, DC, USA, 2001, S. 263–270.
- [Med] Medical Imaging & Technology Alliance: *DICOM - Digital Imaging and Communications in Medicine*, Online: <http://medical.nema.org/>, Letzter Zugriff: 25.04.2012.

- [Mei00] Meißner, M.; Pfister, H.; Westermann, R.; Wittenbrink, C. M.: *Volume Visualization and Volume Rendering Techniques*, 2000.
- [Mei02] Meißner, M.; Kanus, U.; Wetekam, G.; Hirche, J.; Ehlert, A.; Straßer, W.; Doggett, M.; Forthmann, P.; Proksa, R.: *VIZARD II: a reconfigurable interactive volume rendering system*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2002, S. 137–146.
- [Mih03] Mihajlovic, Z.; Budin, L.; Quid, N.: *Reconstruction of gradient in volume rendering*, in *Industrial Technology, 2003 IEEE International Conference on*, Bd. 1, Dezember 2003, S. 282–286 Vol.1.
- [Mik09] Mikkelsen, M. S.: 2009, Online: [http://jbit.net/~sparky/academic/mm\\_brdf.pdf](http://jbit.net/~sparky/academic/mm_brdf.pdf), Letzter Zugriff: 25.04.2012.
- [Mit07] Mittring, M.: *Finding next gen: CryEngine 2*, in *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, ACM, New York, NY, USA, 2007, S. 97–121.
- [Mor] Mora, B.; Evert, D. S.: *Instant Volumetric Understanding with Order-Independent Volume Rendering*, *Computer Graphics Forum*, Bd. 23, Nr. 3, S. 489–497.
- [Ngu07] Nguyen, H.: *GPU Gems 3, Chapter 24. The Importance of Being Linear*, Addison-Wesley Professional, 2007.
- [Nic65] Nicodemus, F. E.: *Directional reflectance and emissivity of an opaque surface*, *Applied Optics*, Bd. 4, Nr. 7, 1965, S. 767–775.
- [Nic77] Nicodemus, F. E.; Richmond, J. C.; Hsia, J. J.; Ginsberg, I. W.; Limperis, T.: *Geometric Considerations and Nomenclature for Reflectance*, *National Bureau of Standards*, 1977.
- [NVI] NVIDIA Corporation: *NVIDIA GPU Programming Guide, GeForce 8 and 9 Series*, December 19, 2008, Online: <http://developer.nvidia.com/nvidia-gpu-programming-guide>, Letzter Zugriff: 25.04.2012.
- [Šo10] Šoltészová, V.; Patel, D.; Bruckner, S.; Viola, I.: *A Multidirectional Occlusion Shading Model for Direct Volume Rendering*, *Computer Graphics Forum*, Bd. 29, Nr. 3, 2010, S. 883–891.

- [One07] Oneppo, M.: *HLSL shader model 4.0*, in *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, ACM, New York, NY, USA, 2007, S. 112–152.
- [onla] *impactscan.org - A brief history of CT*, Online: <http://www.impactscan.org/CThistory.htm>, Letzter Zugriff: 25.04.2012.
- [onlb] *The Programming Language Lua*, Online: <http://www.lua.org/>, Letzter Zugriff: 25.04.2012.
- [onlc] *Qt - A cross-platform application and UI framework*, Online: <http://qt.nokia.com/products/>, Letzter Zugriff: 25.04.2012.
- [Per02] Perlin, K.: *Improving noise*, *ACM Trans. Graph.*, Bd. 21, Nr. 3, 2002, S. 681–682.
- [Pfi99] Pfister, H.; Harderbergh, J.; Knittel, J.; Lauer, H.; Seiler, L.: *The VolumePro real-time ray-casting system*, in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999, S. 251–260.
- [Pha05] Pharr, M.; Fernando, R.: *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, Addison-Wesley Professional, 2005.
- [Pho75] Phong, B. T.: *Illumination for computer generated pictures*, *Commun. ACM*, Bd. 18, Nr. 6, 1975, S. 311–317.
- [Pre00] Preim, B.; Spindler, W.; Peitgen, H.-O.: *Interaktive medizinische Volumenvisualisierung - ein Überblick*, in *SimVis*, 2000, S. 69–88.
- [Pur02] Purcell, T. J.; Buck, I.; Mark, W. R.; Hanrahan, P.: *Ray tracing on programmable graphics hardware*, in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, ACM, New York, NY, USA, 2002, S. 703–712.
- [Ras08] Raspe, M.; Müller, S.: *Controlling GPU-based Volume Rendering using Ray Textures*, *The 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision, (WSCG)*, Februar 2008, S. 277–283.
- [Ray99] Ray, H.; Pfister, H.; Silver, D.; Cook, T. A.: *Ray Casting Architectures for Volume Visualization*, *IEEE Transactions on Visualization and Computer Graphics*, Bd. 5, Nr. 3, 1999, S. 210–223.

- [Rei02] Reinhard, E.; Stark, M.; Shirley, P.; Ferwerda, J.: *Photographic tone reproduction for digital images*, *ACM Trans. Graph.*, Bd. 21, Nr. 3, 2002, S. 267–276.
- [Ril06] Riley, K.; Song, Y.; Kraus, M.; Ebert, D. S.; Levit, J. J.: *Visualization of Structured Nonuniform Grids*, *IEEE Comput. Graph. Appl.*, Bd. 26, Nr. 1, 2006, S. 46–55.
- [Rod99] Rodler, F. F.: *Wavelet Based 3D Compression with Fast Random Access for Very Large Volume Data*, in *Proceedings of the 7th Pacific Conference on Computer Graphics and Applications*, PG '99, IEEE Computer Society, Washington, DC, USA, 1999, S. 108–117.
- [Roe03] Roettger, S.; Guthe, S.; Weiskopf, D.; Ertl, T.; Strasser, W.: *Smart hardware-accelerated volume rendering*, in *Proceedings of the symposium on Data visualisation 2003*, VISSYM '03, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003, S. 231–238.
- [Roe08] Roessler, F.; Botchen, R. P.; Ertl, T.: *Dynamic Shader Generation for Flexible Multi-Volume Visualization*, in *Proceedings of IEEE Pacific Visualization Symposium*, 2008, S. 17–24.
- [Rop10a] Ropinski, T.; Döring, C.; Rezk-Salama, C.: *Advanced Volume Illumination with Unconstrained Light Source Positioning*, *IEEE Computer Graphics and Applications*, 2010, accepted for publication.
- [Rop10b] Ropinski, T.; Döring, C.; Rezk-Salama, C.: *Interactive volumetric lighting simulating scattering and shadowing*, in *PacificVis*, 2010, S. 169–176.
- [RS00] Rezk-Salama, C.; Engel, K.; Bauer, M.; Greiner, G.; Ertl, T.: *Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization*, in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '00, ACM, New York, NY, USA, 2000, S. 109–118.
- [RS05] Rezk-Salama, C.; Kolb, A.: *A Vertex Program for Efficient Box-Plane Intersection*, in *Proc. Vision, Modeling and Visualization (VMV)*, 2005, S. 115–122.
- [RS06] Rezk-Salama, C.; Kolb, A.: *Opacity Peeling for Direct Volume Rendering*, *Computer Graphics Forum (Proc. Eurographics)*, Bd. 25, Nr. 3, 2006, S. 597–606.
- [Sab88] Sabella, P.: *A rendering algorithm for visualizing 3D scalar fields*, *SIGGRAPH Comput. Graph.*, Bd. 22, Nr. 4, 1988, S. 51–58.

- [Sch02] Schmalstieg, D.; Fuhrmann, A.; Hesina, G.; Szalavári, Z.; Encarnaçäo, L. M.; Ger-vautz, M.; Purgathofer, W.: *The studierstube augmented reality project, Presence: Teleoper. Virtual Environ.*, Bd. 11, Nr. 1, 2002, S. 33–54.
- [Sch05] Scharsach, H.: *Advanced GPU Raycasting*, in *In Proceedings of CESCG 2005*, 2005, S. 69–76.
- [Sch06] Scharsach, H.; Hadwiger, M.; Neubauer, A.; Wolfsberger, S.; Bühler, K.: *Perspec-tive Isosurface and Direct Volume Rendering for Virtual Endoscopy Applications*, in *EuroVis*, 2006, S. 315–322.
- [Sch10] Schwarz, M.; Seide, H.-P.: *Fast parallel surface and solid voxelization on GPUs*, *ACM Trans. Graph.*, Bd. 29, Nr. 6, 2010, S. 179:1–179:10.
- [Sha49] Shannon, C. E.: *Communication in the Presence of Noise*, *Proceedings of the IRE*, Bd. 37, Nr. 1, 1949, S. 10–21.
- [Sha99] Shareef, N.; Wang, D. L.; Yagel, R.: *Segmentation of Medical Images Using LEGION*, *IEEE Trans. Med. Imag*, Bd. 18, 1999, S. 74–91.
- [Sie09] Siemens AG, Healthcare Sector: *Trusted Performance Without Compromise - SO-MATOM Sensation*, 2009, Online: [http://www.medical.siemens.com/siemens/de\\_DE/gg\\_ct\\_FBAs/files/brochures/SOMATOM\\_Sensation\\_EN\\_2009.pdf](http://www.medical.siemens.com/siemens/de_DE/gg_ct_FBAs/files/brochures/SOMATOM_Sensation_EN_2009.pdf), Letzter Zugriff: 25.04.2012.
- [Sme09] Smelyanskiy, M.; Holmes, D.; Chhugani, J.; Larson, A.; Carmean, D.; Hanson, D.; Dubey, P.; Augustine, K.; Kim, D.; Kyker, A.; Lee, V.; Nguyen, A.; Seiler, L.; Robb, R.: *Mapping High-Fidelity Volume Rendering for Medical Imaging to CPU, GPU and Many-Core Architectures*, *Visualization and Computer Graphics, IEEE Transactions on*, Bd. 15, Nr. 6, November-Dezember 2009, S. 1563–1570.
- [Sva05] Svakhine, N.; Ebert, D. S.; Stredney, D.: *Illustration Motifs for Effective Medical Volume Illustration*, *IEEE Comput. Graph. Appl.*, Bd. 25, Nr. 3, 2005, S. 31–39.
- [Tor67] Torrance, K.; Sparrow, E.: *Theory for Off-Specular Reflection from Roughened Surfaces*, *Journal of Optical Society of America*, Bd. 57, Nr. 9, 1967, S. 1105–1114.
- [Uns00] Unser, M.: *Sampling-50 years after Shannon*, *Proceedings of the IEEE*, Bd. 88, Nr. 4, April 2000, S. 569–587.

- [Ups88] Upson, C.; Keeler, M.: *V-buffer: visible volume rendering*, *SIGGRAPH Comput. Graph.*, Bd. 22, Nr. 4, 1988, S. 59–64.
- [Wal89] Wallis, J.; Miller, T.; Lerner, C.; Kleerup, E.: *Three-dimensional display in nuclear medicine*, *Medical Imaging, IEEE Transactions on*, Bd. 8, Nr. 4, Dezember 1989, S. 297–230.
- [Wei03] Weiskopf, D.; Engel, K.; Ertl, T.: *Interactive clipping techniques for texture-based volume visualization and volume shading*, *Visualization and Computer Graphics, IEEE Transactions on*, Bd. 9, Nr. 3, Juli-September 2003, S. 298–312.
- [Wes90] Westover, L.: *Footprint evaluation for volume rendering*, *SIGGRAPH Comput. Graph.*, Bd. 24, Nr. 4, 1990, S. 367–376.
- [Wil94] Wilson, O.; Gelder, A. V.; Wilhelms, J.: *Direct Volume Rendering via 3D-textures*, Santa Cruz, CA, USA, 1994.
- [Wit98] Wittenbrink, C. M.; Malzbender, T.; Goss, M. E.: *Opacity-weighted color interpolation, for volume sampling*, in *Proceedings of the 1998 IEEE symposium on Volume visualization*, VVS '98, ACM, New York, NY, USA, 1998, S. 135–142.
- [Wre10] Wrenninge, M.; Zafar, N. B.; Clifford, J.; Graham, G.; Penney, D.; Kontkanen, J.; Tessendorf, J.; Clinton, A.: *Volumetric Methods in Visual Effects*, in *SIGGRAPH Course Notes*, 2010.
- [Wyn98] Wyner, A.; Shamai, S.: *Introduction To "Communication In The Presence Of Noise"*, *Proceedings of the IEEE*, Bd. 86, Nr. 2, Februar 1998, S. 442–446.

# Bilderverzeichnis

|      |   |    |
|------|---|----|
| 1.1  | Volumen-Rendering eines medizinischen Datensatzes . . . . .   | 2  |
| 2.1  | Polygone vs. Voxels . . . . .                                 | 7  |
| 2.2  | Volumen-Rendering Anwendungsfälle . . . . .                   | 8  |
| 2.3  | CT-Datensatz aus einer realen Patientenbehandlung . . . . .   | 10 |
| 2.4  | Volumen-Raycasting . . . . .                                  | 13 |
| 3.1  | Mipmapping und Texturfilterung . . . . .                      | 18 |
| 3.2  | Mehrere Volumen gleichzeitig in einer Szene . . . . .         | 21 |
| 3.3  | Kantenglättung . . . . .                                      | 22 |
| 3.4  | Strahlgenerierung . . . . .                                   | 28 |
| 3.5  | Schnitt durch Ebene nahe der Bildebene . . . . .              | 34 |
| 3.6  | Tiefenpuffer Probleme bei der Strahlgenerierung . . . . .     | 34 |
| 4.1  | Schnittebene . . . . .  | 38 |
| 4.2  | Tiefentextur . . . . .  | 41 |
| 4.3  | Auswirkung der Abtastrate und künstlichen Rauschen . . . . .  | 42 |
| 4.4  | Realisierungsmöglichkeiten für künstliches Rauschen . . . . . | 43 |
| 4.5  | Strahlverfolgung . . . . .                                    | 44 |
| 4.6  | MIP-Variationen . . . . .                                     | 47 |
| 4.7  | Schnitt während der Strahlverfolgung . . . . .                | 49 |
| 4.8  | Drei Rekonstruktionsfilter im Vergleich . . . . .             | 54 |
| 4.9  | Shading Eingabe . . . . .                                     | 55 |
| 4.10 | Klassifikation . . . . .                                      | 59 |
| 4.11 | Transferfunktion GUI mit Histogramm . . . . .                 | 62 |
| 4.12 | Gradient . . . . .  | 65 |
| 4.13 | Beleuchtung . . . . .   | 67 |
| 5.1  | Volumen-Rendering Integration in eine 3-D-Engine . . . . .    | 72 |

|     |  |    |
|-----|--|----|
| 5.2 | Blickwinkel der Experimente . . . . .                      | 74 |
| 5.3 | Blinn-Phong und Cook-Torrance Beleuchtungsmodell . . . . . | 82 |
| 5.4 | VTC ohne Anpassung der Kompressionsblöcke . . . . .        | 84 |

# Tabellenverzeichnis

|      |  |    |
|------|--|----|
| 5.1  | Verwendete Hard- und Software . . . . .  | 71 |
| 5.2  | Minimum, Medium und Maximum beim Volumendatensatz <i>Head (Visible Male)</i> . . . . .   | 75 |
| 5.3  | Minimum, Medium und Maximum beim Volumendatensatz <i>Schwein</i> . . . . .               | 75 |
| 5.4  | Volumendatensatz <i>Head (Visible Male)</i> ohne Kompression . . . . .                   | 76 |
| 5.5  | Volumendatensatz <i>Schwein</i> ohne Kompression . . . . .                               | 76 |
| 5.6  | Variieren der Bildauflösung, Volumendatensatz <i>Head (Visible Male)</i> . . . . .       | 77 |
| 5.7  | Variieren der Bildauflösung, Volumendatensatz <i>Schwein</i> . . . . .                   | 77 |
| 5.8  | Variieren der Abtastrate, Volumendatensatz <i>Head (Visible Male)</i> . . . . .          | 77 |
| 5.9  | Variieren der Abtastrate, Volumendatensatz <i>Schwein</i> . . . . .                      | 78 |
| 5.10 | Variieren der Volumengröße, Volumendatensatz <i>Schwein</i> . . . . .                    | 78 |
| 5.11 | Variieren der Volumengröße mit Zwischenstufen, Volumendatensatz <i>Schwein</i> . . . . . | 79 |
| 5.12 | Variieren der Rekonstruktion, Volumendatensatz <i>Head (Visible Male)</i> . . . . .      | 79 |
| 5.13 | Variieren der Rekonstruktion, Volumendatensatz <i>Schwein</i> . . . . .                  | 79 |
| 5.14 | Variieren der Gradientenschätzung, Volumendatensatz <i>Head (Visible Male)</i> . . . . . | 81 |
| 5.15 | Variieren der Gradientenschätzung, Volumendatensatz <i>Schwein</i> . . . . .             | 81 |
| 5.16 | Variieren der Beleuchtung, Volumendatensatz <i>Head (Visible Male)</i> . . . . .         | 81 |
| 5.17 | Variieren der Beleuchtung, Volumendatensatz <i>Schwein</i> . . . . .                     | 82 |
| 5.18 | Back-To-Front vs. Front-To-Back, Volumendatensatz <i>Head (Visible Male)</i> . . . . .   | 83 |
| 5.19 | Back-To-Front vs. Front-To-Back, Volumendatensatz <i>Schwein</i> . . . . .               | 83 |
| B.1  | Direct3D Shadermodell vs. OpenGL-Versionen vs. GLSL-Versionen . . . . .                  | 95 |



# Quellcodeverzeichnis

|      |   |    |
|------|---|----|
| 3.1  | Einstiegspunkt des Volumen-Rendering Fragmentshaders . . . . .                  | 28 |
| 3.2  | Vertexshader zur Berechnung des Austrittspunktes des Strahls . . . . .          | 30 |
| 3.3  | Fragments shader für Strahlgenerierung . . . . .                                | 30 |
| 4.1  | Funktionssignatur für den Strahlschnitt . . . . .                               | 38 |
| 4.2  | Fragments shader für den Schnitt eines Strahls mit einer Ebene . . . . .        | 40 |
| 4.3  | Funktionssignatur für künstliches Rauschen . . . . .                            | 42 |
| 4.4  | Realisierung des künstlichen Rauschens über trigonometrische Funktionen . .     | 43 |
| 4.5  | Funktionssignatur für die Strahlverfolgung . . . . .                            | 44 |
| 4.6  | Fragments shader für MIP-Strahlverfolgung . . . . .                             | 46 |
| 4.7  | Funktionssignatur für einen Schnitt innerhalb des Volumens . . . . .            | 49 |
| 4.8  | Fragments shader für Schnitt innerhalb des Volumens . . . . .                   | 51 |
| 4.9  | Funktionssignaturen für die Rekonstruktion der Volumendaten . . . . .           | 52 |
| 4.10 | Fragments shader Implementierung für die <i>FetchScalar</i> -Funktion . . . . . | 53 |
| 4.11 | Funktionssignatur für Shading . . . . .   | 55 |
| 4.12 | Fragments shader für Shading mit lokaler Beleuchtung . . . . .                  | 56 |
| 4.13 | Funktionssignatur für die Klassifikation . . . . .                              | 57 |
| 4.14 | Fragments shader für die Klassifikation über eine 1-D-Transferfunktion . . . .  | 59 |
| 4.15 | Funktionssignatur für die Gradientenschätzung . . . . .                         | 63 |
| 4.16 | Fragments shader für die Gradientenschätzung über Vorwärtsdifferenzen . . . .   | 64 |
| 4.17 | Fragments shader für die Gradientenschätzung auf klassifizierten Skalarwerten . | 64 |
| 4.18 | Funktionssignatur für die lokale Beleuchtung . . . . .                          | 67 |
| 4.19 | Fragments shader für lokale Beleuchtung über das Lambert-Beleuchtungsmodell     | 68 |



# Hilfsmittelverzeichnis

Schriftliche Ausarbeitung:

- Von Prof. Dr.-Ing. Frank Deinzer bereitgestelltes L<sup>A</sup>T<sub>E</sub>X-Template (*BA-MT-Vorlage\_Ver.8.zip*)
- L<sup>A</sup>T<sub>E</sub>X für Windows: MiK<sub>T</sub>E<sub>X</sub> 2.9 (<http://miktex.org/>)
- Grafischer L<sup>A</sup>T<sub>E</sub>X-Editor: TeXstudio 2.3 (<http://texstudio.sourceforge.net/>)
- Notepad++ v5.9.8 (<http://notepad-plus.sourceforge.net/de/site.htm>)

Grafiken:

- Bullzip PDF Printer 7.2.0.1338 (<http://www.bullzip.com/>)
- Inkscape 0.48.2 (<http://www.inkscape.org/>)
- Paint.NET v3.5.10 (<http://www.getpaint.net/index.html>)
- GIMP 2.6.11 (<http://www.gimp.org/>)

Implementierung:

- Entwicklungsumgebung: Visual Studio 2010 Ultimate
- PixelLight 3-D-Engine (<http://www.pixellight.org/>)

Quellcode Dokumentation:

- Doxygen 1.7.6.1 (<http://www.doxygen.org/>)
- Graphviz 2.28 (<http://www.graphviz.org/>)
- Microsoft HTML Help Compiler



# Abkürzungsverzeichnis

- DVR** Direct Volume Rendering, Deutsch: Direktes Volume-Rendering
- IVR** Indirect Volume Rendering, Deutsch: Indirektes Volume-Rendering
- MIP** Maximum Intensity Projection, Deutsch: Maximumintensitätsprojektion
- GMIP** Gradient Maximum Intensity Projection, Deutsch: Gradient-Maximumintensitätsprojektion
- MIDA** Maximum Intensity Differences Accumulation
- CT** Computed Tomography, Deutsch: Computertomographie
- MRI** Magnetic Resonance Imaging, Deutsch: Magnetresonanztomographie
- PET** Positron Emission Tomography, Deutsch: Positronen-Emissions-Tomographie
- DSA** Digital Subtraction Angiography, Deutsch: Digitale Subtraktionsangiographie
- WYSIWYG** What You See is What You Get
- DICOM** Digital Imaging and Communications in Medicine
- VTC** Volume Texture Compression
- BRDF** Bidirectional Reflectance Distribution Function, Deutsch: Bidirektionale Reflektanzverteilungsfunktion
- HDR** High Dynamic Range Rendering
- MSAA** Multisample Anti-Aliasing
- FXAA** Fast Approximate Anti-Aliasing
- WebGL** Web Graphics Library
- OS** Operating System, Deutsch: Betriebssystem
- RAM** Random-Access Memory, Deutsch: Speicher mit wahlfreiem/direktem Zugriff
- PC** Personal Computer, Deutsch: Einzelplatzrechner
- WHQL** Windows Hardware Quality Labs
- CPU** Central Processing Unit, Deutsch: Hauptprozessor
- GPU** Graphics Processing Unit, Deutsch: Grafikprozessor
- GPGPU** General-Purpose Computing on Graphics Processing Units, Deutsch: Allgemeine Berechnungen auf Grafikprozessoren
- OpenCL** Open Computing Language

**OpenGL** Open Graphics Library

**OpenGL ES** OpenGL for Embedded Systems

**GLSL** OpenGL Shading Language

**NVIDIA** NVIDIA Corporation

**CUDA** Compute Unified Device Architecture

**Cg** C for Graphics, Shader-Hochsprache von NVIDIA

**FPS** Frames Per Second, Deutsch: Bildwiederholfrequenz

**MVC** Model-View-Control, Deutsch: Modell-Präsentation-Steuerung

**GUI** Graphical User Interface, Deutsch: Grafische Benutzeroberfläche

**RTTI** Runtime Type Information, Deutsch: Typinformation zur Laufzeit

**NaN** Not a Number, Deutsch: Keine Zahl