UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

NUMENUS

# Ray Tracing von Volumendaten auf Basis des Augenblick-SDK

## Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von

Matthias Bohnen

| | |
|---|---|
| Erstgutachter: | Prof. Dr.-Ing. Stefan Müller<br>(Institut für Computervisualistik, AG Computergraphik) |
| Zweitgutachter: | Dr. Oliver Abert<br>Numenus GmbH |

Koblenz, im September 2009

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☐ | ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ☐ | ☐ |

.....
(Ort, Datum) (Unterschrift)

# Inhaltsverzeichnis


**1 Motivation** **2**
- 1.1 Vor- und Nachteile der CPU . . . 3
- 1.2 Das Augenblick-SDK . . . 7
- 1.3 Voxel Raytracing . . . 8
  - 1.3.1 Rasterisierung . . . 8
  - 1.3.2 Raytracing von Voxeln . . . 10
  - 1.3.3 Medizinische Volumendaten . . . 13

**2 Grundlagen** **14**
- 2.1 Raytracing . . . 14
- 2.2 Voxel . . . 19
- 2.3 Octree . . . 19
- 2.4 SIMD und SEE . . . 20
- 2.5 Augenblick-SDK . . . 22
  - 2.5.1 Plugin-API . . . 23
  - 2.5.2 Arbeitsteilung . . . 24

**3 Die Verarbeitungskette** **26**
- 3.1 Das Konverterprogramm . . . 26
  - 3.1.1 Einlesen des Polygondatensatzes . . . 27
  - 3.1.2 Erstellen einer Farbpalette . . . 28
  - 3.1.3 Erzeugen des Voxel-Octree . . . 31
  - 3.1.4 Nachbearbeitung des Octree . . . 35
  - 3.1.5 Serialisieren des Baumes in ein binäres Dateiformat . 37
- 3.2 Das Voxel-Dateiformat . . . 39
  - 3.2.1 Der Header . . . 39
  - 3.2.2 Der serialisierte Baum . . . 40
- 3.3 Das Voxel-Loader-Plugin . . . 44
- 3.4 Die Octree-Datenstruktur . . . 46
  - 3.4.1 Uniform Grid . . . 46
  - 3.4.2 Der Octree . . . 47
  - 3.4.3 kd-Trees . . . 51
  - 3.4.4 Bounding Volume Hierarchien . . . 51
  - 3.4.5 Kodierung des Octree . . . 52
  - 3.4.6 Die einzelnen Voxel . . . 56
- 3.5 Der Voxel-Raytracer . . . 60
  - 3.5.1 Das Konzept . . . 60
  - 3.5.2 Vorteile aus der Datenstruktur . . . 66
  - 3.5.3 Optimierungen . . . 67
- 3.6 Erzeugung des resultierenden Framebuffers . . . 79

**4 Fazit** **83**
4.1 Ergebnisse . . . . . 83
4.1.1 Geschwindigkeit des Raytracing . . . . . 83
4.1.2 Speicherverbrauch der Volumendaten . . . . . 87
4.1.3 Darstellungsqualität . . . . . 90
4.2 Probleme . . . . . 92
4.2.1 Architekturbedingte Probleme . . . . . 92
4.2.2 Probleme der Volumendaten . . . . . 94
4.3 Ausblick . . . . . 97
4.3.1 Weitere Beschleunigung des Raytracers . . . . . 97
4.3.2 Erhöhung der Datenkompression . . . . . 98
4.3.3 Steigerung der Darstellungsqualität . . . . . 99
4.3.4 Die mögliche Zukunft des Raytracings von Volumendaten . . . . . 101

**5 Anhang** **104**

**Zusammenfassung**

Die Welt der interaktiven Computergrafik befindet sich, wie kaum ein anderer Bereich der Informatik, seit über einer Dekade in einer sehr dynamischen Entwicklung. Angefeuert von den großen und immer noch steigenden, Umsatzvolumina der Computerspieleindustrie[1] steigt die Leistungsfähigkeit der Grafikkarten seit Jahren schneller als das Mooresche Gesetz dies erwarten ließe. Die Hersteller überbieten sich gegenseitig mit noch schnelleren und noch flexibleren Recheneinheiten. Von Anwendungsseite befeuern die Programmierer diese Entwicklung ebenfalls, in dem sie mit jeder neuen Computerspiele-Generation die verfügbare zusätzliche Leistung aufzehren. Ein Ende dieser Entwicklung ist vorerst nicht absehbar. Mit dem wachsenden Detailgrad steigen aber auch die Datenmengen, die es zu verwalten und darzustellen gilt. War die Rasterisierung von Polygondaten in der interaktiven Computergrafik bisher scheinbar ohne ernsthafte Alternative, werden die Entwickler nun verstärkt dazu gedrängt über neue Wege nachzudenken wie man der Datenmengen auch in Zukunft noch Herr werden kann.
Ein vielversprechender Ansatz zur Bewältigung der Datenmengen und weiteren Steigerung der Darstellungsqualität ist das das Raytracing von Voxel-Octrees. Allen voran John Carmack, zigfach ausgezeichneter Technologieführer und Leitfigur der Industrie und seine Firma id Software preisen in jüngster Zeit die Zukunft dieses Ansatzes in Kombination mit Rasterisierung als Hybridlösung (siehe [Shr08] und [Oli08]). Andere federführende Firmen wie Crytek und Epic Games forschen ebenfalls daran ([Yer09], [Swe09]).
In dieser Diplomarbeit wurde das Raytracing von Voxel-Octrees im Rahmen des Augenblick-SDK implementiert. Diese Ausarbeitung gibt einen Einblick in die Motivation, Grundlagen, Implementation und die Zukunft dieser Technologie.

[1] Laut der Studie [Inc08] von Price Waterhouse Coopers weltweiter Umsatz mit PC und Konsolenspielen im Jahre 2007: 28,7 Mrd. USD, mit 10% jährlichem Wachstum bis 2012

# 1 Motivation

Ziel dieser Diplomarbeit ist es, auf der Basis des Augenblick-SDK und der CPU einen hinsichtlich Geschwindigkeit und Speicherverbrauch möglichst performanten Raytracer für Volumendaten zu entwickeln.
Das Reizvolle an dieser Zielsetzung ist die Konzeption, Programmierung

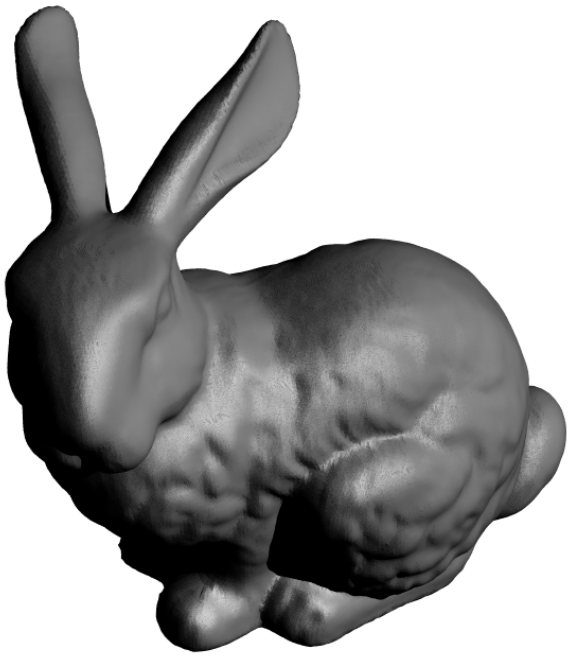

**Abbildung 1:** Ausgabe des Voxel-Octree-Raytracers

und hardwarenahe Optimierung einer kompletten Verarbeitungskette für das Raytracing von Voxeln. Bei der Bewältigung dieser Aufgaben wurden viele Teilbereiche der Computergrafik und Informatik im Allgemeinen touchiert. Das Raytracing von Voxeln ist ein derzeit von einigen Forschergruppen und Firmen forcierter Ansatz zum interaktiven Darstellen großer Datenmengen, welcher das Potential hat sich neben der herkömmlichen, Polygon-basierten Rasterisierung zu etablieren. Raytracing könnte über die Produkte der Computerspiele-Industrie endlich den Sprung in den Massenmarkt schaffen.
Abbildung 1 zeigt eine Ausgabe des in dieser Diplomarbeit entwickelten Raytracers[2]. Im Folgenden soll auf die einzelnen Aspekte der Zielsetzung näher eingegangen werden.

[2]Modell „Bunny“, 66,6 Millionen Voxel, Auflösung $4096^3$

## 1.1 Vor- und Nachteile der CPU

Zunächst soll erläutert werden, welche Motive hinter der Umsetzung des Raytracings auf der CPU stehen. Die Wahl der CPU ist keine Selbstverständlichkeit, denn mit steigender Flexibilität und Leistungsfähigkeit der Grafikkarte ist das Umsetzen rechenaufwändiger Algorithmen auf der GPU (Graphics Processing Unit) seit Beginn dieses Jahrtausends ein starker Trend in der Informatik geworden. Die CPU hat an Attraktivität verloren, da sie in Transistorzahl, Speicheranbindung und roher Rechenkraft von der GPU längst überholt worden ist.

| Typ | Modell | Transistoren | Speicherbandbreite |
|---|---|---|---|
| CPU | Intel Core i7 9xx [1] | 0,7 Mrd. | 12 Gb/s |
| CPU | AMD Athlon II X2 [2] | 0,2 Mrd. | 33 Gb/s |
| GPU | Nvidia GeForce GTX 295 [3] | 2 x 1,4 Mrd. | 223,8 Gb/s |
| GPU | AMD Radeon HD 5870 [4] | 2,1 Mrd. | 153,6 Gb/s |

**Tabelle 1:** Vergleich aktueller CPUs und GPUs

Wie Tabelle 1 zeigt, bestehen moderne GPUs aus erheblich mehr Transistoren und können einiges mehr an Daten pro Sekunde aus dem Speicher abrufen als aktuelle CPUs. Der direkte Vergleich von Gleitkommaoperationen pro Sekunde (Floating Point Operations Per Second, FLOPS) zwischen CPUs und GPUs ist nicht sinnvoll, da CPUs die zur Verfügung stehenden Recheneinheiten generell effizienter ausnutzen. Als Größenordnung sei aber darauf verwiesen, dass die in der Tabelle genannte Grafikkarte von Nvidia eine rohe Rechenleisung von etwa 1,8 Billionen Gleitkommaoperationen pro Sekunde (1,8 TFLOPS) erreicht. Die Radeon HD 5870 erzielt sogar 2,7 TFLOPS, während CPUs derzeit bei etwa 100-200 GFLOPs liegen.

Der Begriff GPGPU (General Purpose GPU) hat sich seit einem Vortrag auf der Siggraph 2004 (siehe [LHK+04]) als Oberbegriff für diese Art der Programmierung etabliert. Raytracing ist eines dieser rechenaufwändigen Verfahren welches in letzter Zeit von einigen Entwicklern auf der GPU implementiert worden ist, z.B. von [Rab08] oder [Chr05]. [PBMH02] hat schon im Jahre 2002 gezeigt, dass auf damaliger Grafikkartenhardware Raytracing umsetzbar ist, obwohl zu diesem Zeitpunkt noch keine Shader-

[1] http://www.intel.com/design/corei7/documentation.htm

[2] http://www.amd.com/us/products/desktop/processors/athlon-ii-x2/Pages/AMD-athlon-ii-x2-processor-model-numbers-feature-comparison.aspx

[3] http://www.nvidia.com/object/product_geforce_gtx_295_us.html

[4] http://www.heise.de/newsticker/meldung/145716

Hochsprache zur Verfügung stand. Wie man die Beschränkungen des Grafikkartenspeichers (kein wahlfreier Schreibzugriff auf die Texturen in den Shader) durch neue Verfahren umgehen kann zeigt [PGSS07]. Auch beim Raytracing von Voxeln auf der GPU haben „Interactive Gigavoxels“ von Crassin et al. [CNL08] sowie [GMI08] beeindruckende Ergebnisse erzielt.

Auf Seiten der CPU war die Entwicklung seit Jahrzehnten vom Mantra der Taktsteigerung geprägt. Mit schrumpfender Strukturgröße wurde die Rechenleistung der Prozessoren durch immer höhere Taktraten gesteigert. Dabei blieb die CPU jedoch auf einen einzigen Kern beschränkt. Parallelisierung gab es höchstens in Form von SIMD-Befehlssätzen, Multithreading wurde ausschließlich in Software auf Betriebssystemebene durch Timesharing umgesetzt. In jüngster Zeit jedoch hat sich der Fokus der CPU-Entwicklung, hauptsächlich aufgrund unüberwindbarer Probleme mit Leckströmen bei hohen Taktraten (wie [NC05] erläutert), auf eine Mehrkern-Philosophie ausgerichtet. Während die Taktraten auf einem Plateau von etwa 3 Ghz verweilen sind mittlerweile CPUs mit bis zu acht Kernen Stand der Technik.
Die Entwicklung der CPU hin zu mehr Parallelisierung und der GPU hin zu mehr Flexibilität lassen darauf schließen, dass sie sich in Zukunft in Aufbau und Programmierung noch ähnlicher werden. Diese Sicht wurde kürzlich unter anderem von Cervat Yerli (siehe [Yer09]), Gründer der Computerspielefirma Crytek sowie Tim Sweeney (siehe [Swe09]), Gründer und CEO von Epic Games prominent vertreten. Diese Vermutung wird ferner gestützt von in Entwicklung befindlichen Projekten namhafter Hardwarehersteller:

- So verfolgt die Firma AMD, nach Volumen zweitgrößter Hersteller von x86 CPUs[3] und drittgrößter Hersteller von Grafikkarten, mit dem Projekt „Fusion“ das Ziel, bis zum Jahre 2011 die CPU mit der GPU in einem Produkt zu verschmelzen[4].

- Intel wiederum, nach Volumen größter Hersteller von CPUs und Grafikkarten weltweit, plant mit dem „Larrabee“-Projekt nichts Geringeres, als den Markt für High-End Computergrafik mit einer aus CPUs zusammengesetzten Grafikkarte aufzurollen[5].

Auf einer Larrabee-Grafikkarte der ersten Generation werden voraussichtlich 32 und in der nächsten Generation 48 leicht modifizierte x86 CPUs der ersten Pentium Generation (P5) die Arbeit verrichten. Larrabee wird über eine frei programmierbare Grafik-Pipeline die traditionellen OpenGL

[3] http://www.heise.de/newsticker/meldung/143128
[4] http://sites.amd.com/us/fusion/Pages/index.aspx
[5] http://www.intel.com/technology/visual/microarch.htm

und DirectX Schnittstellen unterstützen. Die Fixed-Function-Pipeline wird dabei effektiv in Software auf den CPUs emuliert. Darüber hinaus wird man auch, aller Voraussicht nach, unmodifizierten x86-Assemblercode auf der Grafikkarte ausführen können. Für die noch etwas fernere Zukunft wäre es denkbar, dass die Larrabee-Grafikkarte als eine zweite, ausgelagerte CPU in das Betriebssystem integriert und somit für die Software vollends transparent wird (siehe [SCS+08]).
Je nach Blickwinkel könnte man aus den genannten Projekten schließen, dass die CPU die GPU langfristig integrieren (AMD Fusion) oder ersetzen (Intel Larrabee) wird. Für ersteren Schluss gibt es in der Geschichte der x86-Mikroprozessoren bereits Vorläufer. So war die Gleitkommaeinheit (FPU) ursprünglich ein separater Coprozessor, der mit Einführung der Intel Pentium Produktlinie Mitte der Neunziger Jahre in die CPU integriert wurde. Ferner hat AMD mit Einführung der Opteron Produktlinie im Jahre 2003 den Speichercontroller (Front Side Bus) aus der Northbridge des Mainboards in die CPU verlagert (HyperTransport) [6].

Parallel zu der Entwicklung von GPU und CPU hat es auch Bemühungen gegeben, zusätzliche Beschleunigungskarten für Spezialanwendungen wie Raytracing zu etablieren. So war die RPU (Ray Processing Unit) von Wald et al. eine wichtige Initiative in Richtung echtzeitfähiges Raytracing (siehe [WS05]). Es handelt sich hierbei um eine in einem FPGA programmierte, dedizierte Beschleunigungskarte für Raytracing. Prototypen dieser Karte konnten bei einem Bustakt von nur 66 Mhz bereits Szenen mit mit bis zu 20 Bildern pro Sekunde darstellen. Ferner bietet die Firma Caustic mit der „CausticOne“ eine, nicht für den Massenmarkt gedachte, Raytracing-Beschleunigerkarte an, welche das Raytracing im Vergleich zur Ausführung auf der CPU um den Faktor 20 beschleunigen soll[7]. Es ist nicht davon auszugehen, dass sich derartige Speziallösungen am Massenmarkt durchsetzen werden, die Entwicklung der GPU hin zu mehr Flexibilität steht dem diametral entgegen. Auch Beschleunigungskarten für Physikeffekte wie PhysX[8] konnten sich am Markt nicht durchsetzen.

Unabhängig davon was die Zukunft bringt gibt es gute Gründe, bei der Entwicklung von hochperformanter Software auf die CPU zu setzen. Rund um die CPU hat sich über die Zeit ein Ökosystem von Werkzeugen entwickelt, die das effiziente und hardwarenahe Programmieren unterstützen. Beispielsweise die GNU Toolchain mit dem GDB (GNU Debugger), oder Valgrind, ein Programm das vor allem zum Speichermonitoring, aber auch zum Code Profiling eingesetzt wird. Diese

[6] http://www.amd.com/us/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx
[7] http://www.caustic.com/caustic-rt_caustic-one.php
[8] http://www.nvidia.de/object/nvidia_physx_de.html

Programme sind im Rahmen dieser Diplomarbeit zum Einsatz gekommen und haben die Optimierung des Programmcodes unterstützt. Auf der GPU gibt es bisher kein Ökosystem vergleichbarer Werkzeuge. So hat der Grafikkartenhersteller Nvidia erst vor wenigen Monaten den weltweit ersten Grafikkarten Debugger für die CUDA-Technologie vorgestellt, dessen Schnittstelle bezeichnenderweise den GNU Debugger imitiert[9]. Ferner ist die Entwicklung auf der CPU größtenteils standardisiert. Die erwähnte GNU Toolchain, Valgrind und co. laufen auf einer Vielzahl von Betriebssystemen und allen x86-Prozessoren. Die GPU hingegen wird derzeit noch von herstellerspezifischen GPGPU-Lösungen dominiert. Die am weitesten verbreitete GPGPU-Schnittstelle/SDK ist CUDA (Compute Unified Device Architecture) von Nvidia. Mittels CUDA kann Programmcode in C-Syntax auf der Grafikkarte ausgeführt werden. Allerdings sind die zur Verfügung stehenden Befehle auf die Grafikkarte zugeschnitten, x86-Assembler kann nicht ausgeführt werden. Anders als der Name CUDA vermuten lässt steht diese Schnittstelle nur auf Nvidia Grafikkarten zur Verfügung. Als herstellerunabhängige Lösung befindet sich das kürzlich von der Khronos Group in Version 1.0 veröffentlichte OpenCL am Horizont[10]. Beim Portieren von Algorithmen von der CPU auf die GPU muss unter anderem bedacht werden, dass die GPU keinen wahlfreien Lese- und Schreibzugriff auf den Speicher erlaubt. Darüber hinaus steht der Grafikkarte in einem gewöhnlichen Setup weniger Arbeitsspeicher zur Verfügung als der CPU. Gerade Voxel-Datensätze aber, wie sie in dieser Diplomarbeit zum Einsatz kommen, sind geeignet den verfügbaren Speicher voll auszunutzen.

Zu guter Letzt ähnelt die Grafikkarte historisch bedingt einer Einbahnstraße. Als in Hardware gegossene Grafik-Pipeline empfängt sie Daten und Befehle von der CPU und stellt das Resultat auf einem Bildschirm dar. Die Anbindung an den Rest der Computers erfolgt daher üblicherweise hochgradig asynchron, das Rücksenden großer Datenmengen an die CPU ist nicht vorgesehen. Somit eignet sich die Grafikkarte mit der asychronen Anbindung und enormen parallelen Rechenkraft nicht für jede Art von Algorithmus gleich gut. Algorithmen mit einer hohen arithmetischen Dichte, d.h. Programme die viele mathematische Operationen ausführen müssen, profitieren stärker von der Ausführung auf der GPU als Programme die viele I/O Operationen durchführen müssen.

Zusammenfassend lässt sich sagen, dass die Entwicklung von CPU und GPU sich immer weiter annähern und damit die Argumente für oder gegen das Eine wie das Andere verschwimmen werden. Es scheint einiges dafür

[9] http://www.nvidia.com/object/io_1239219734947.html
[10] http://www.khronos.org/opencl

zu sprechen, dass es eine gute Entscheidung ist seine Zeit in das Programmieren auf der CPU zu investieren, da sie langfristig die GPU einverleiben oder ersetzen könnte. Ferner kann davon ausgegangen werden, dass eine gleichwertige Implementierung eines Raytracers auf der GPU hardwarebedingt aufwändiger ist und mangels guter Debugging-Werkzeuge mehr Zeit kostet.
Wie [Abe08] betont, sollte eine Umsetzung des Augenblick-SDK auf der Larrabee-Grafikkarte von Intel, sobald sie 2010 auf den Markt kommt, ohne größere Hürden möglich sein. Somit sollte es, aller Voraussicht nach, mit geringem Portierungsaufwand möglich sein den im Rahmen dieser Diplomarbeit entwickelten Raytracer, der bereits jetzt auf der CPU echtzeitfähig ist, in Zukunft zusammen mit Augenblick auf der Larrabee-Grafikkarte mit einem Vielfachen der heute verfügbaren CPU-Rechenkraft auszuführen.

## 1.2 Das Augenblick-SDK

Das Augenblick-SDK ist ein durch eine Plugin-Architektur erweiterbares Raytracing-Framework der Firma Numenus[11].
Den Kern bildet die „Augenblick Ray Tracing Engine“, die das Raytracing von Freiformflächen und Polygonen unterstützt. Durch die Verwendung von „Surface Area Heuristic Bounding Volume Hierarchy“ als Datenstruktur und eines auf SIMD-Operationen optimierten Programmcodes erreicht der Augenblick Raytracer auf modernen CPUs echtzeitfähige Bildwiederholraten [Abe08]. Mehrere Gründe sprechen für eine Umsetzung auf Basis des Augenblick-SDK.

1. Es ist durch die Plugin-Architektur leicht, Augenblick um neue Funktionalität zu erweitern.

2. Kernelemente eines Raytracers, wie der Szenengraph zur Verwaltung von Geometrie, Lichtquellen und der Kamera müssen nicht von Grund auf neu entwickelt werden.

3. Das Augenblick-SDK bietet dem Entwickler einen Satz von Klassen, die den SSE-Befehlssatz moderner Prozessoren kapseln und auf einer höheren Ebene zur Verfügung stellen. Damit wird Programmcode, der sehr hardwarenah und effizient sein soll, les- und wartbarer.

4. Der Kern von Augenblick kümmert sich durch die Verwendung von Multi-Threading um eine effiziente Ausnutzung der zur Verfügung stehenden Hardware.

[11] http://www.numenus.de

5. Das Augenblick-SDK bietet eine Benutzeroberfläche, die sich um die Darstellung im Betriebssystem und die Benutzerschnittstelle kümmert.

Auf die Details des Augenblick-SDK und die Arbeitsteilung geht Kapitel 2.5 näher ein.

## 1.3 Voxel Raytracing

Für eine grundlegende Einführung in die Begriffe „Voxel“ und „Raytracing“ siehe Kapitel 2.1 & 2.2.
Raytracing ist ein Verfahren zur Bildsynthese, also zur perspektivischen Projektion einer dreidimensionalen Szene auf eine zweidimensionale Bildebene. Der Vorgang des Erstellens des Bildes nennt sich Rendering. Wie [Bet06] feststellt, muss man in der Computergrafik zwischen dem Offline-Rendering und dem interaktiven Rendering unterscheiden. Beim Offline-Rendering ist die Darstellungsqualität wichtiger als die Berechnungszeit, beim interaktiven Rendering hingegen ist die Berechnungszeit der kritische, limitierende Faktor. Für interaktives bzw. echtzeitfähiges Rendering sollte eine Bildsynthese nicht länger als der fünfundzwanzigste Teil einer Sekunde dauern, anderenfalls kommt es für den Betrachter zu sichtbaren Störungen im Animationsfluss.
Es gibt verschiedene Bildsynthese-Verfahren. Neben dem Raytracing ist dies vor allem die Rasterisierung. Laut [Bet06] hat sich Raytracing aufgrund seiner nahezu photorealistischen Resultate schon lange im Bereich des Offline-Renderings etabliert[12]. Auf dem Sektor des interaktiven Renderings jedoch, wie es vor allem in Computerspielen zum Einsatz kommt, dominiert bisher aufgrund seiner Effizienz die Rasterisierung.

### 1.3.1 Rasterisierung

Die Rasterisierung ist ein Verfahren zur Bildsynthese welches auf Dreiecken, die zu Polygonen vernetzt eine Oberfläche annähern, basiert. Jedes Dreieck besteht dabei aus den Koordinaten der drei Eckpunkte, sowie Eigenschaften der Eckpunkte wie Farbe, Normale und Texturkoordinate. Mit der Fixed-Function-Pipeline steht eine elegante und sehr effiziente Methode zur Verfügung, die Polygonmodelle perspektivisch korrekt auf eine Bildebene zu projizieren und für die Ausgabe auf einem Bildschirm zu rasterisieren. Die Fixed-Function-Pipeline besteht aus einer Reihe von mathematischen Operationen, welche, auf die Eckpunkte der Dreiecke angewandt, eine Transformation in das Koordinatensystem der Kamera, perspektivische Projektion, Beschneidung auf den Bildausschnitt und

[12] Neben anderen Verfahren wie REYES, auf die hier nicht näher eingegangen wird. Pixars RenderMan basierte lange Zeit ausschließlich auf dem REYES-Verfahren, seit dem Film „Cars“ wird aber auch Raytracing eingesetzt (siehe [CFLB06]).

schließlich Rasterisierung ergeben. Da diese Operationen für jedes Dreieck separat ausgeführt werden können, ist das Rasterisierungsverfahren sehr gut (pro Dreieck bzw. Eckpunkt) parallelisierbar. Aus der Parallelisierung ergibt sich aber das Problem, dass einige optische Effekte bei der Rasterisierung nicht berechnet werden können. [Wal06] schreibt darüber:

> Ein noch gravierender Nachteil der Rasterisierung ist, dass sie für hochqualitative Darstellungen nur bedingt geeignet ist: Da jedes Polygon einzeln und für sich alleine gerastert wird, können indirekte Effekte wie z.B. Schattenwurf oder Reflektion – welche grundlegend auf dem direkten Zusammenspiel mehrerer Objekte basieren – nicht direkt berechnet werden. Um diese Limitierung zu umgehen wurden – speziell für Spieleanwendungen – eine Vielzahl an Algorithmen und Verfahren entwickelt, um solche Effekte wirklichkeitsähnlich vortäuschen zu können.

Verfahren die den Schattenwurf simulieren sind beispielsweise Shadow Mapping und Shadow Volumes. Spiegelungen an einer ebenen Fläche können durch Verdopplung der Geometrie hinter einer transparenten Fläche simuliert werden. Komplexere Reflexionen aber, oder etwa Kaustiken[13], können mit der Rasterisierung nicht simuliert werden. Trotz der unterlegenen Darstellungsqualität der Rasterisierung hat sie sich, und damit auch die Repräsentation von dreidimensionalen Modellen in Form von Polygonen, aufgrund der Effizienz des Verfahrens in der interaktiven Computergrafik allgemein durchgesetzt, zumal Effekte wie Kaustiken in vielen Kontexten eine untergeordnete Rolle spielen.

Mit steigender Anzahl von Dreiecken pro Szene in modernen Computerspielen drängt sich zunehmend die Frage auf, ob Polygone und Rasterisierung auch in Zukunft noch die effizienteste Form der Repräsentation und Darstellung sind. Mit steigender Anzahl an Dreiecken steigt auch die Zahl der für den Betrachter nicht sichtbaren Dreiecke. Idealerweise werden diese unsichtbaren Dreiecke von einem Verdeckungstest frühzeitig verworfen. Mittels Verdeckungstests löst die Rasterisierung das Sichtbarkeitsproblem. Bei dem Sichtbarkeitsproblem geht es um die Frage, welche Objekte auf der Bildebene sichtbar sind und welche nicht. Der entscheidende Verdeckungstest findet in der Fixed-Function-Pipeline erst am Ende statt, wenn die einzelnen Dreiecke rasterisiert werden und für jeden Pixel anhand eines Tiefenpuffertests entschieden wird, ob er näher an der Kamera ist als alles bisher an dieser Stelle gezeichnete. Da der Verdeckungstest erst am Ende geschieht, bedeutet dies dass viele

[13] Kaustiken sind Helligkeitsmuster die bei der Brechungen von Lichtstrahlen bei dem durchqueren von Materialien mit unterschiedlichen Brechungsindizes entstehen.

für den Betrachter nicht sichtbare Dreiecke unnötigerweise die Pipeline durchlaufen. Ein eindeutiger Verdeckungstest, der nur die sichtbaren Dreiecke übrig lässt, ist zu Beginn der Pipeline prinzipbedingt nicht möglich, da einzelne Dreiecke teilweise verdeckt und teilweise sichtbar sein können. Occlusion Culling nähert sich dem Problem an, löst es aber nicht vollständig. Mit steigender Zahl an Dreiecken in einer Szene verschärft sich die Problematik, die Effizienz der Rasterisierung sinkt.
Mit steigender Zahl an Dreiecken reduziert sich zudem die Fläche die jedes einzelne Dreiecke abdeckt. Wenn ein Detailgrad erreicht wird, bei dem ein Bildschirmpixel sich aus mehreren Dreiecken zusammensetzt ist es ebenfalls fraglich, ob Dreiecke noch die effizienteste Form der Repräsentation der Daten sind. Pro dargestellten Bildschirmpixel eine Vielzahl von Eckpunktfarben, Normalen und Texturkoordinaten zu laden erscheint nicht sinnvoll. Bedenkt man die mangelhafte erzielbare physikalische Korrektheit und die sinkende Effizienz der Pipeline mit steigender Zahl von Dreiecken, erscheint es lohnenswert für die interaktive Darstellung alternative Renderingverfahren in Betracht zu ziehen.

### 1.3.2 Raytracing von Voxeln

Das Raytracing-Verfahren, welches ja bereits für Offline-Rendering breiten Einsatz findet, ist eine mögliche Alternative zur Rasterisierung, wenn man es schafft seine Berechnung echtzeitfähig zu bewältigen. Das Verfahren wird in Kapitel 2.1 detaillierter erklärt.
Zusammengefasst kann man festhalten, dass die erzielbare Darstellungsqualität besser ist und der Aufwand mit steigender Anzahl von zu schneidenden Oberflächen nur logarithmisch ansteigt, während er bei der Rasterisierung linear steigt[14]. Bei sehr großen Datenmengen ist die Bilanz des Raytracing also besser als die der Rasterisierung. Der Nachweis, dass der Aufwand beim Raytracing mit wachsender Datenmenge tatsächlich nur logarithmisch steigt wird in Kapitel 4.1.1 erbracht.
Trotz des logarithmischen Aufwands ist Raytracing ein immer noch sehr rechenaufwändiges Verfahren. Um es für heutzutage gängige Datenmengen praxistauglich zu machen ist es notwendig das Verfahren weiter zu optimieren, damit es im interaktiven Rendering mittelfristig eine ernsthafte Alternative zur Rasterisierung wird.

Wenn man von Rasterisierung auf Raytracing umsteigt, entfällt der Zwang mit Polygonen zu arbeiten[15]. Da es fraglich ist, ob Polygone immer

[14] Der Aufwand steigt beim Raytracing logarithmisch, weil das Sichtbarkeitsproblem effizient gelöst wird. Für unsichtbare Objekte entsteht, sofern Datenstrukturen wie BVHs oder Octrees eingesetzt werden, kaum Mehraufwand.

[15] Über den Umweg des Marching Cube Algorithmus, mit dem sich Voxel in Polygone

noch eine sinnvolle Repräsentationsform sind wenn jedes einzelne Dreieck nur noch einen Bruchteil eines Bildschirmpixels ausfüllt und beim Raytracing alternative, ggf. effizientere Speicherformen ausprobiert werden können, erscheint es sinnvoll über alternative Repräsentationsformen dreidimensionaler Daten nachzudenken.
Volumendaten sind eine solche alternative Repräsentationsform. Ein Volumendatensatz ist ein dreidimensionales Modell, welches aus vielen kleinen Volumen, Voxeln genannt, zusammengesetzt wird. Voxel sind von kubischer Form und unterscheiden sich voneinander vor allem in ihrer Farbe (und ggf. weiteren Eigenschaften). Anstatt beim Raytracing den Schnittpunkt eines Strahls mit den Dreiecken zu bestimmen, wird beim Voxel-Raytracing der Schnitt des Strahls mit den Voxeln bestimmt. Werden die Voxel in einer Baumstruktur wie dem Octree abgespeichert, bietet sich die Möglichkeit durch unterschiedlich große Voxel auf den verschiedenen Ebenen des Baums den Datensatz in unterschiedlichem Detailgrad abzuspeichern, was LOD (Level Of Detail) genannt wird. Voxel-Octrees werden in Kapitel 2.3 genauer erklärt, aber es kann festgehalten werden dass diese einige interessante Vorteile gegenüber dem Rasterisieren oder Raytracen von Polygonen haben.
Zum Einen ist es beim Raytracing eines Voxel-Octrees möglich, dank der Detailstufen den Detailgrad der Szene immer so fein aufzulösen, dass zu jeden Bildschirmpixel genau ein Voxel von passender Größe die Farbinformation liefert. Verfeinerungen der Geometrie können so beim Annähern des Betrachters kontinuierlich nachgeladen werden und beim Entfernen des Betrachters kann das Raytracing auf gröberen Detailsstufen vorzeitig abgebrochen werden. Da dieses Verfahren pixelgenau passende Detailgrade liefert, skaliert es besser als das Rasterisieren oder Raytracing von Polygondatensätzen.
Zum Zweiten ist bemerkenswert, dass die Detailstufen in einem Voxel-Octree für Geometrie und Texturen zugleich gelten, da der Voxel-Octree beides in sich vereint. In einem Polygondatensatz hingegen müssen Texturen separat gespeichert und ihr LOD in Form von Mip-Maps angelegt werden. LOD wird zwar auch für Polygondatensätze implementiert, anders als bei dem Octree bekommt man sie aber nicht umsonst. Aus weniger Dreiecken zusammengesetzte Varianten eines Modells müssen in einem Vorverarbeitungsschritt berechnet und separat abgespeichert werden. Zudem kann man den Detailgrad nicht für jeden Bildschirmpixel individuell anpassen, sondern muss für jedes Bild einen festen Detailgrad des Modells wählen. Der abrupte Wechsel des Detailgrades eines Modells zwischen zwei Bildern muss dann interpoliert werden.

---

umwandeln lassen, können zumindest indirekt auch bei der Rasterisierung alternative Repräsentationsformen wie Voxel dargestellt werden. Gleiches liefert die Tesselierung für Freiformflächen.

Aus den genannten Gründen gibt es neuerdings auch in der Unterhaltungsindustrie Bestrebungen zum produktiven Einsatz von Voxel-Octree-Raytracing. Insbesondere die „id Tech 6 Engine“ von id Software hat die Diskussion um Vor- und Nachteile des Voxel-Raytracing angefeuert. Diese Rendering Engine wird in zukünftigen Computerspielen von id Software zum Einsatz kommen und voraussichtlich Geometrie in Form von „Sparse Voxel Octrees“ raytracen. Epic Games (siehe [Swe09]) und Crytek (siehe [Yer09]) forschen ebenfalls an dieser Technologie. Raytracing von Polygonen wurde schon für ältere Spiele wie Ouake 3[16] demonstriert und wird gerade für Quake Wars umgesetzt (siehe [Poh09]).

Neben der Vereinigung von Geometrie und Texturen sowie den integrierten Detailstufen steckt in der Voxel-Octree-Datenstruktur ein weiterer Vorteil: Die Position eines Voxels wird nicht in ihm selbst abgespeichert, sondern steht implizit über seine Position im Baum fest. Das spart nicht nur Speicherplatz, sondern reduziert auch die Zahl der notwendigen Schnitttests. Denn an jedem Punkt im Raum kann es nur einen Voxel geben und der Raytracer kann die Voxel in einer festen Reihenfolge entlang des Strahls untersuchen. Nachdem ein Schnitt mit einem gefüllten Voxel gefunden wurde ist es nicht mehr notwendig weitere Voxel zu schneiden.
Bei Polygon-Raytracing müsste man theoretisch den Strahl mit jedem Dreieck in der Szene schneiden um festzustellen welcher Schnittpunkt am nächsten zum Betrachter ist. Üblicherweise grenzt man diesen Aufwand durch Datenstrukturen wie eine Bounding Volume Hierarchie ein, würde dann aber immer noch eine Priority Queue von Bounding Volumes verwalten um die notwendigen Schnitttests weiter zu reduzieren.
In der impliziten räumlichen Anordnung der Voxel in ihrer Datenstruktur steckt allerdings auch ein großer Nachteil. Es ist erheblich aufwändiger, Objekte in einem Voxel-Octree zu animieren. In einem Dreieck sind die Koordinaten seiner drei Eckpunkt abgespeichert. Will man das Dreieck verschieben, rotieren oder skalieren müssen nur die Koordinaten der Eckpunkte mithilfe von Matrixmultiplikationen manipuliert werden. Will man einen Voxel verschieben, so muss man seine Position in der Datenstruktur verändern, was in der Regel umfangreiche I/O Operationen zur Folge hat. Rotation und Skalierung eines aus Voxeln zusammengesetzten Objekts sind nicht triviale Operationen für welche die lineare Algebra nicht, oder nur über Umwege, zur Verfügung steht (siehe [GKHS98]).

[16] http://graphics.cs.uni-sb.de/ sidapohl/egoshooter/

### 1.3.3 Medizinische Volumendaten

Wenn von Volumendaten die Rede ist, sind oftmals Volumendaten aus dem medizinischen Sektor gemeint, die in der Regel in einem gleichmäßigen Raster (Uniform Grid) gespeichert werden. Hier hat sich das DICOM-Format[17] als Standard für den Austausch etabliert.
Die Verwendung solcher Datensätze kam für diese Diplomarbeit nicht in Frage, da sich die von den bildgebenden Verfahren erzeugt Datensätze inhaltlich erheblich von einem Datensatz der Unterhaltungsindustrie unterscheiden. Medizinische Volumendatensätze speichern in der Regel ein größtenteils gleichförmig mit Voxeln gefülltes Volumen, die Information ist also sehr gleichmäßig verteilt. Jeder Voxel beschreibt dabei z.B. die Dichte des untersuchten Körpers an einer bestimmten Stelle (Computertomographie) oder einen Gewebetyp (Magnetresonanztomographie). Die Visualisierung dieser Daten auf einem Bildschirm kann dabei nur durch eine Transferfunktion erfolgen, welche z.B. die Dichteverhältnisse auf eine Grauwertskala abbildet.
In einem typischer Datensatz aus der Unterhaltungsindustrie hingegen ist die Information sehr ungleichmäßig verteilt. Die Details konzentrieren sich entlang einer dünnen, meist undurchlässigen Oberfläche des Modells. Der Rest des Raumes ist homogen durchsichtig. Jedem Voxel ist direkt eine Farbe (und ggf. weitere Eigenschaften) zugewiesen, somit bedarf es lediglich eines Shaders zur Beleuchtung. Transferfunktionen werden auf diesen Datensätzen nicht benötigt.

[17] http://medical.nema.org

# 2 Grundlagen

Das Kapitel Grundlagen erläutert alle benötigten grundlegenden Begriffe für das Verständnis dieser Arbeit.

## 2.1 Raytracing

Raytracing ist ein Verfahren zur Bildsynthese, d.h. zur perspektivischen Projektion dreidimensionaler Daten auf eine virtuelle Bildebene. Das resultierende, perspektivisch korrekte Bild der virtuellen Welt kann gespeichert oder direkt auf einem Bildschirm ausgegeben werden. Werden die Bilder in Echtzeit berechnet und unmittelbar auf dem Bildschirm ausgegeben, spricht man von interaktivem Raytracing.

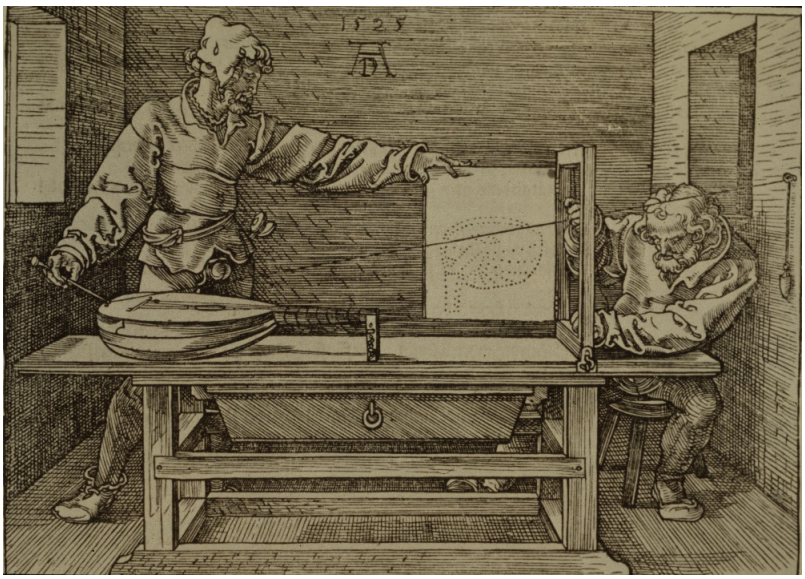

**Abbildung 2:** Albrecht Dürer: Mann, eine Laute zeichnend[18]

Das Konzept des Raytracings, mittels simulierter Lichtstrahlen perspektivisch zu projizieren, ist der Menschheit schon mindestens seit der frühen Neuzeit bekannt. Abbildung 2 zeigt eine Illustration Albrecht Dürers, entworfen im Jahre 1525 für ein Malerhandbuch. Dieser Holzschnitt soll dem Leser demonstrieren wie man Objekte perspektivisch korrekt auf eine Leinwand übertragen kann. Dem Konzept zugrunde liegt die Annahme, dass Licht in Form von Strahlen auf unser Auge trifft.

[18]Gemeinfreies Bild aus: http://commons.wikimedia.org/wiki/File:Dürer_-_Man_Drawing_a_Lute.jpg

Dieses auch Strahlenoptik genannte Konzept vernachlässigt mikroskopische Effekte die sich aus der Welleneigenschaft des Lichts ergeben.
Die Vorstellung, dass sich Licht in Form von Strahlen bewegt ist noch älter. Von dem griechischen Philosophen Archytas von Tarent (ca. 435 bis 355 v. Chr.) ist überliefert, dass er der Meinung war, Licht würde nicht in unser Auge treffen, sondern das Auge selbst würde immaterielle Strahlen in die Welt versenden (siehe [Bur05]). Interessanterweise kommt Archytas mit dieser Sicht der praktischen Umsetzung des Raytracings sehr nahe.

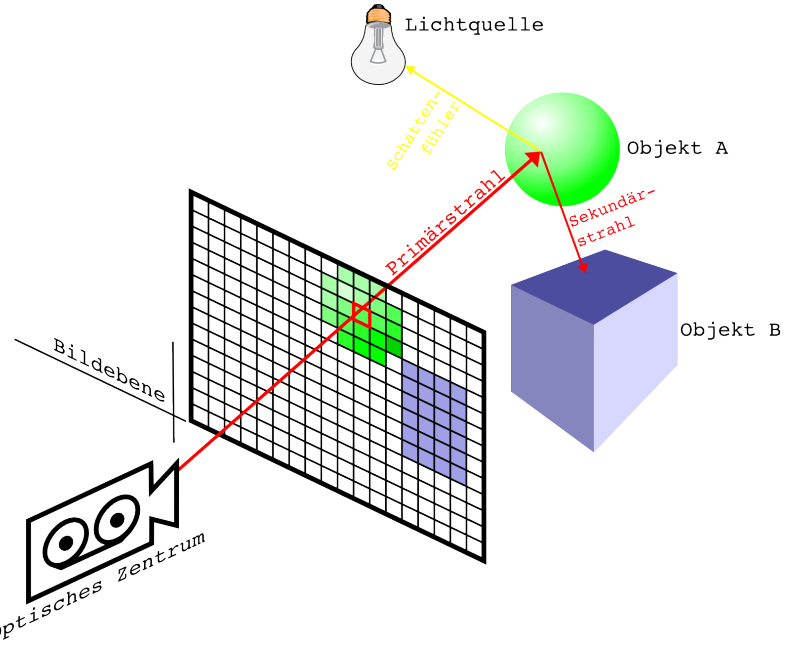


**Abbildung 3:** Schematisches Konzept des Raytracing[19]

Als möglicher Algorithmus zur Lösung des Sichtbarkeitsproblems am Computer wurde Raytracing erstmals 1968 von [App68] vorgeschlagen. Erstmals genauer ausgeführt, implementiert und um wichtige Aspekte wie die Rekursion erweitert wurde das Verfahren 1980 von Whitted (siehe [Whi80]). Abbildung 3 gibt das seit Whitted kanonische Verständnis von Raytracing schematisch wieder.

Es würde der Realität entsprechen, die Lichtstrahlen bzw. Wellen bzw. Photonen von den Lichtquellen aus in die Welt zu versenden, ggf. an Oberflächen zu reflektieren und zu schauen welche Strahlen die Linse der virtuellen Kamera treffen. Ein Verfahren welches Lichtstrahlen, oder besser

[19] Enthält eine gemeinfreie Grafik (Glühbirne) aus: http://www.openclipart.org

gesagt Photonen, auf diese Art und Weise in die Welt versendet ist Photon Mapping (siehe [Jen96]). Allerdings dient dieses Verfahren der Beleuchtung und nicht der Lösung des Sichtbarkeitsproblems.
Zur Lösung des Sichtbarkeitsproblems ist ein solches Vorgehen viel zu verschwenderisch, da nur ein Bruchteil der ausgesandten Strahlen die Linse treffen werden. Um ein Vielfaches schneller ist es daher, diejenigen Strahlen welche die Linse treffen zu ihrem Ursprungsort zurück zu verfolgen. Dass dieses umkehrte Vorgehen dennoch physikalisch korrekte Ergebnisse liefert folgt, wie [Rab08] feststellt, aus der Helmholtz-Reziprozität, die besagt dass der Pfad den das Licht beschreitet und der relative Energieverlust in beide Richtungen der Gleiche wäre.
Ausgehend vom optischen Zentrum wird also durch jeden Pixel in der Bildebene ein Strahl berechnet. Dieser nennt sich Primärstrahl. Die Formel für den Strahl ist die Geradengleichung in Parameterform:

$$\vec{r} = \vec{o} + \lambda \cdot \vec{d} \tag{1}$$

$\vec{o}$ ist der Ortsvektor (der Vektor zum optischen Zentrum) und $\vec{d}$ der Richtungsvektor des Strahls.
Nun gilt es diejenigen Oberflächen zu bestimmen, die der Strahl schneidet. Der Schnittpunkt für den wir den kleinsten Lambda-Wert erhalten liefert uns jenes Objekt, welches dem optischen Zentrum am nächsten, also sichtbar ist. Somit ist das Sichtbarkeitsproblem für diesen Pixel gelöst. In der Summe aller Pixel wird das Sichtbarkeitsproblem für die komplette Szene gelöst.

In einer naiven Implementation würde man jeden Strahl mit jeder Oberfläche schneiden. Der Aufwand steigt dabei exponentiell mit steigender Zahl von Objekten und deren Oberflächen. Erheblich effizienter ist es, die Objekte in einer Hierarchie anzuordnen. Je nach Anwendungsfall kommen dazu Baumstrukturen wie Bounding Volume Hierarchien, Octrees oder kd-Trees zum Einsatz.
Ersteres ist ein Objekt unterteilendes Verfahren, Letztere sind raumunterteilend. Ihnen allen gemein ist, dass sie die Menge aller Objekte nach dem Prinzip „Teile und Herrsche“ räumlich gruppieren. So können die Objekte die der Kamera näher sind zuerst geschnitten und die Verfolgung beim ersten erfolgreichen Schnitt abgebrochen werden. Damit reduziert sich der Aufwand auf ein logarithmisches Maß. Mehr Objekte heißt dann zwar dass ein paar mehr Schnitte gemacht werden müssen, aber je mehr Objekte da sind umso mehr Objekte sind von anderen verdeckt und müssen nie geschnitten werden.
Allerdings steigt beim Raytracing der Aufwand linear mit der Auflösung des zu berechnenden Bildes (was für die Rasterisierung, zumindest auf den ersten Stufen der Pipeline, nicht der Fall ist).

Will man berechnen ob eine Oberfläche im Schatten liegt, so muss man vom Schnittpunkt des Primärstrahls mit der Oberfläche sogenannte Schattenfühler (siehe Abbildung 3) zu jeder Lichtquelle aussenden. Gibt es einen Schnitt des Schattenfühlers mit einem Objekt vor der Lichtquelle, so liegt die Oberfläche im Schatten dieser Lichtquelle. Anderenfalls muss aus dem Winkel zu der Lichtquelle der Einfluss jener Lichtquelle auf die Farbe der Oberfläche berechnet werden. Ein gängiges Modell hierzu das von Phong ([Pho75]).
Sollen Reflektionen berücksichtigt werden, so muss der Algorithmus rekursiv vom Schnittpunkt mit der Oberfläche aus gemäß der Formel „Eintrittswinkel gleich Austrittswinkel“ für einen zweiten Strahl ausgeführt werden. Eine komplexere Funktion zur Berechnung der Winkel, Reflexion und Reflexionsanteilen liefert die BRDF (Bidirectional Reflectance Distribution Function). Der berechnete Strahl nennt sich Sekundärstrahl. Der Reflexionsvorgang kann sich theoretisch unendlich oft wiederholen. Um zu verhindern dass der Aufwand zur Berechnung einzelner Strahlen ausartet gibt es stets ein Rekursionslimit, ab dem die Strahlverfolgung abgebrochen wird.
Wenn man transparente Materialien darstellen möchte, ist es notwendig auch die hinter der transparenten Fläche liegenden Objekte mit dem Strahl zu schneiden. Sollen realistische Brechungen des Lichts beim Ein- und Austreten in einem transparentes Medium berechnet werden, ist ebenfalls ein rekursiver Aufruf mit dem durch den Brechungsindex des Mediums modifizierten Strahl notwendig.

Bezüglich der Repräsentationsform der dreidimensionalen Daten ist das Raytracing-Verfahren insofern flexibel, als grundsätzlich alles strahlverfolgt werden kann für das man den Schnittpunkt mit einem Strahl berechnen kann. Dies gilt unter anderem für Dreiecke, aber auch für Voxel. Für Freiformflächen ist dies prinzipiell nicht möglich, allerdings gibt es mit der Newton Iteration (siehe [Abe08]) und der Tesselierung zwei Verfahren um Freiformflächen annäherungsweise mit einem Strahl zu schneiden.
Da das Konzept des Raytracing im Gegensatz zur Rasterisierung an die Realität angelehnt ist können damit realistischere Darstellungen berechnet werden. Schatten, Reflexionen und Brechungen bekommt man, ohne großen programmiererischen Mehraufwand, „geschenkt“.

Die Ausführung des Raytracing-Algorithmus lässt sich leicht parallelisiern, da die Berechnung eines Strahles vollkommen unabhängig von der Berechnung des nächsten erfolgen kann. Allerdings müssen für benachbarte Strahlen, da sie sehr nah beieinander liegende Wege durch den Raum nehmen, mit hoher Wahrscheinlichkeit die gleichen Operationen ausgeführt werden. Diese Instruktionsparallelität lässt sich auf Prozessoren mit SIMD-Recheneinheit ausnutzen. Kapitel 2.4 erläutert

den Begriff SIMD im Detail. Die Instruktionsparallelität endet, wenn ein Strahl auf ein Objekt trifft und die anderen parallel, verarbeiteten Strahlen noch weiter durch die Welt verfolgt werden müssen. Aufgrund der räumlichen Nähe der Strahlen zueinander müssen oftmals nicht nur die gleichen Operationen ausgeführt, sondern auch mit den gleichen Objekten geschnitten werden. Diese Daten- oder Cache-Kohärenz kommt dem Raytracing-Verfahren automatisch zugute, da die Daten bei mehrfacher Anforderung kurz hintereinander mit hoher Wahrscheinlichkeit bereits im Cache des Prozessors vorliegen. Daten aus dem Cache zu besorgen dauert, wie [Abe08] zu entnehmen ist, im Falle des L1-Cache nur 1-4 Prozessorzyklen. Aus dem L2-Cache sind es ca. 20-30 Zyklen. Im Vergleich dazu dauert es bis zu 300 Zyklen die Daten aus dem Arbeitsspeicher abzurufen. Die Cache-Kohärenz endet meist bei den Sekundärstrahlen. Abbildung 4 illustriert diese Umstände.

Ein für das interaktive Rendering nicht zu unterschätzender Nachteil des

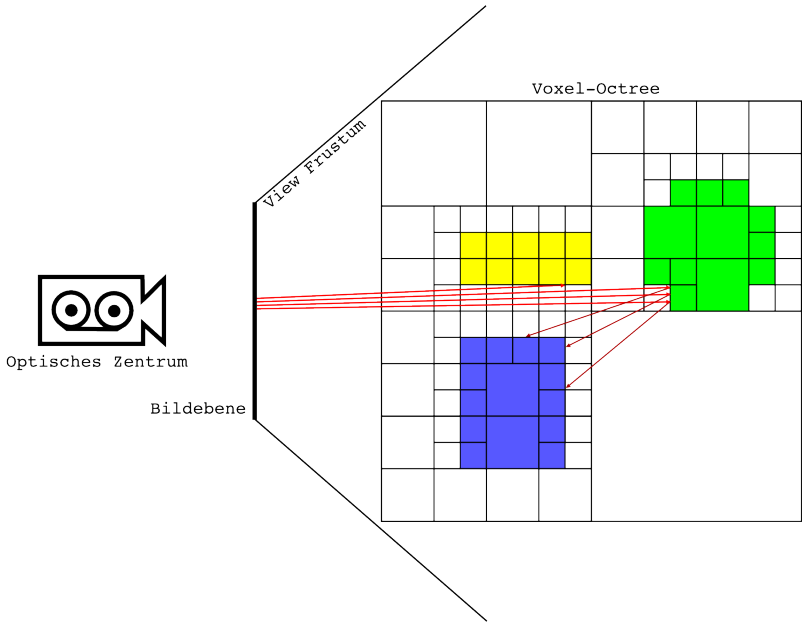


**Abbildung 4:** Illustration der Instruktionsparalellität und Datenkohärenz

Raytracings ist, dass in der Regel die komplette Welt im Arbeitsspeicher vorhanden sein muss, da man sich aufgrund der nicht vorhersehbaren Streuung der Sekundärstrahlen nicht auf die Objekte innerhalb des View Frustums beschränken kann.

## 2.2 Voxel

Ein Voxel (Volumetric Pixel) ist ein Volumenelement im dreidimensionalen Raum, in Analogie zum Pixel (Picture Element, Bildelement) im zweidimensionalen Raum. Darüber hinaus schwanken von Kontext zu Kontext und Implementation zu Implementation die weiteren Details die einen Voxel ausmachen. Im Kontext dieser Diplomarbeit besitzen Voxel eine in alle drei Dimensionen einheitliche Ausdehnung (kubische Form). Ansonsten ist jedem Voxel eine Farbe (bzw. ein Farbpalettenindex) und eine Normale zugewiesen.
Da die Voxel im Rahmen dieser Diplomarbeit in einem Octree gespeichert werden, sind nur die Voxel einer Ebene im Baum gleich groß. Zwischen den Ebenen des Baumes gilt die Beziehung:

$$d^e = \frac{d^{e-1}}{2} \tag{2}$$

D.h. die Ausdehnung $d$ jedes Voxels in Ebene $e$ entspricht der Hälfte der Ausdehnung der Voxel in Ebene $e-1$. Die Position der Voxel im Raum wird nicht explizit in jedem Voxel gespeichert, sondern ergibt sich implizit aus ihrer Position im Octree.

## 2.3 Octree

Ein *Oct-tree* ist eine Baumstruktur (ein gerichteter Graph ohne Schleifen) in der jeder Knoten keine oder acht Kinder hat. Knoten ohne Kinder werden auch als Blätter des Baumes bezeichnet. In dieser Diplomarbeit ist jeder Knoten und jedes Blatt des Octree ein Voxel. Die Begriffe *Voxel* und *Knoten/Blatt* werden daher im Folgenden austauschbar verwendet.
Abbildung 5 zeigt die Unterteilung eines Raumes durch den nebenstehend

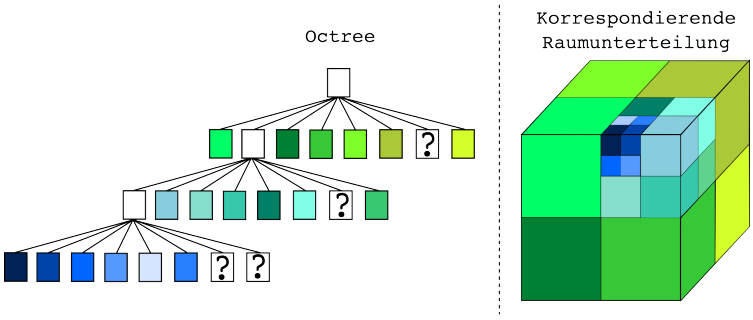


**Abbildung 5:** Ein beispielhafter Octree und die dazugehörige Raumunterteilung

abgebildeten Octree.

Die Octrees wie sie in dieser Diplomarbeit auftreten unterteilen den Raum stets uniform in acht gleichgroße Oktanten und sind nicht balanciert. Tatsächlich könnten sie hochgradig unbalanciert sein, da die Gestalt des Baums direkt mit der Schwankung der räumlichen Informationsdichte im Modell korreliert.
Abbildung 6 zeigt, in welcher Reihenfolge die Kinder räumlich angeord-

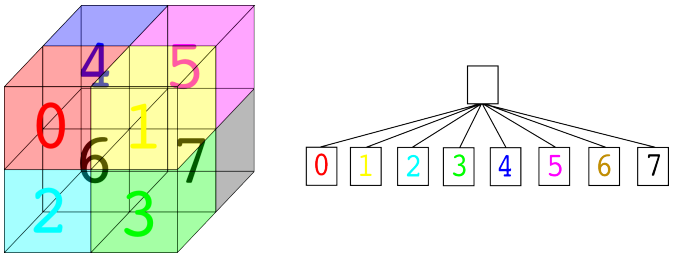


**Abbildung 6:** Anordnung der Kinder in einem Octree

net sind. Üblicherweise werden Oktanten in der Geometrie gegen den Uhrzeigersinn durchnummeriert. Aus Performancegründen wurde hier jedoch eine andere Reihenfolge gewählt. Warum diese Nummerierung die Performance steigert erläutert Kapitel 3.5.3.

Da es problematisch ist, den Inhalt eines dreidimensionalen Raumes in einer zweidimensionalen Darstellung wiederzugeben, werden im Folgenden viele Darstellungen auf Quadtrees im zweidimensionalen Raum basieren. Die Defintion eines Quadtree ist identisch zu der des Octree, mit dem Unterschied dass ein Knoten im Quadtree nur vier Kinder besitzt. Die Darstellungen des Quadtree im zweidimensionalen Raum lassen sich problemlos um eine Raumdimension erweitert auf Octrees übertragen. Abbildung 7 zeigt ein einfaches Beispiel eines Quadtree und links davon die dementsprechend unterteilte Ebene.

## 2.4 SIMD und SEE

Das Akronym SIMD steht für „Single Instruction, Multiple Data“. Es entstammt der Flynnschen Klassifikation (siehe [Fly72]), welche vier Typen von Rechnerarchitekturen unterscheidet. Diese sind in Tabelle 2 aufgeführt. Die Flynnsche Klassifikation dient dem Zweck,die in der Praxis anzutreffenden Rechnerarchitekturen anhand ihrer Befehls- und Datenströme zu unterscheiden.

- SISD entspricht dem Normalfall, bei dem ein Prozessor einen Befehl auf einen Datensatz anwendet.

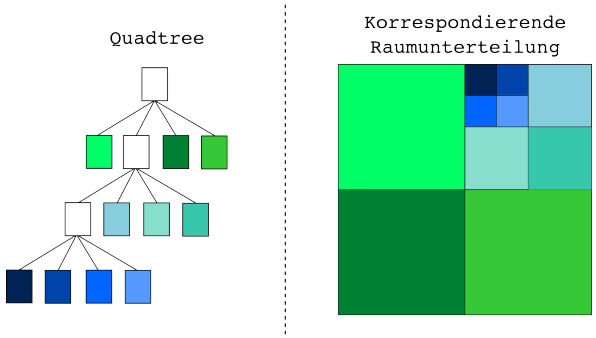


**Abbildung 7:** Ein beispielhafter Quadtree und die dazugehörige Raumunterteilung

| | Single Instruction | Multiple Instruction |
|---|---|---|
| Single Data | SISD | MISD |
| Multiple Data | SIMD | MIMD |

**Tabelle 2:** Die Flynnsche Klassifikation

- SIMD steht für den Fall dass ein Prozessor einen Befehl auf mehrere Datensätze gleichzeitig anwendet. Traditionell waren es Supercomputer in Vektorprozessorbauweise, wie die der Firma Cray, die in dieser Form konstruiert wurden. Inzwischen hat sich SIMD in Form der Streaming SIMD Extensions (SSE) von Intel als Befehlssatzerweiterung von x86-CPUs auch auf handelsüblichen PCs verbreitet. SIMD eignet sich für Algorithmen die sich gut parallelisieren lassen. Raytracing gehört zu den Verfahren die sich mit SIMD beschleunigen lassen (den Nachweis bringt Kapitel 4.1.1).

- Die Klasse MISD („Multiple Instruction, Single Data“) ist insofern umstritten, als es kein überzeugendes Beispiel einer solchen Rechnerarchitektur gibt.

- MIMD entspricht dem Fall dass mehrere Prozessoren zeitgleich verschiedene Operationen auf verschiedenen Datensätzen ausführen. Eine solche Architektur ist immer dann gegeben, wenn mehrere Prozessoren in einem Computer arbeiten oder mehrere Computer zu einem Supercomputer zusammengeschlossen werden.

SSE, eine Abkürzung für „Streaming SIMD Extensions“, ist ein erweiterter Befehlssatz für x86-Prozessoren, entworfen und 1999 eingeführt von Intel (siehe [RPK00]). In den folgenden Jahren wurde der Befehlssatz mit SSE2 bis SSE4 sukzessive um komplexere Befehle erweitert. Kern

des Befehlssatzes sind SIMD-Operationen für vier Gleitkommawerte einfacher Genauigkeit (4*32 = 128 Bit). Aufgrund eines Patenaustauschabkommens unterstützen auch Prozessoren der Firma AMD diese Befehlssatzerweiterungen. Die in Kapitel 1.1 erwähnte, zukünftige High-End Grafikkarte Larrabee von Intel wird über sechszehnfaches SIMD verfügen (im Vergleich zum vierfachen SIMD der SSE-CPUs).

## 2.5 Augenblick-SDK

Das Augenblick-SDK ist ein modulares Framework zur Entwicklung von plattformunabhängigen Raytracing-Applikationen. Mit der Augenblick-GUI steht eine ebenfalls plattformunabhängige Benutzeroberfläche auf Basis von Qt zur Verfügung.
Der Kern von Augenblick bietet einen Raytracer für Polygonmodelle und Freiformflächen, ist aber durch eine Plugin-API um weitere Funktionalitäten erweiterbar. Ferner bietet das SDK einen umfangreichen Satz von Klassen die unter anderem den SSE-Befehlssatz moderner Prozessoren kapseln und dem Programmierer in Form überladener Operatoren zur Verfügung stellen.
Üblicherweise wird der SSE-Befehlssatz über Compiler Intrinsics angesprochen, dies sind Methodenaufrufe welche einzelne Assemblerbefehle kaschieren. Der Compiler übersetzt den Methodenaufruf direkt in den dazugehörigen Assemblerbefehl. Dem Programmierer bleibt es so zwar erspart, aus seiner Hochsprache heraus auf Assemblerbefehle zugreifen zu müssen, da es sich aber um eine direkte Übertragung handelt unterscheidet sich der Umgang mit den Compiler Intrinsics kaum von der direkten Programmierung in Assembler. Augenblick bietet mit den Datentypen Float4, Int4 und Bool4 drei Klassen welche alle auf diesen Datentypen gängigen Operatoren überladen. Die Operatoren sind in den Klassen mit den SSE Compiler Intrinsics implementiert. Programmcode, welcher mit diesen Klassen arbeitet ist erheblich übersichtlicher. Befehle die sich über mehrere Zeilen Compiler Intrinsics erstrecken würden, werden auf ein intuitives Operatorzeichen verkürzt. [Abe08] konnte zeigen, dass moderne Compiler genug Wissen über den Programmcode haben um die überladenen Operatoren derart zu optimieren, dass bei ihrer Verwendung im Vergleich zu den Compiler Intrinsics kein signifikanter Geschwindigkeitsverlust zu erwarten ist.

Die interne Rendering-Pipeline von Augenblick besteht aus sogenannten „Execution Chains“, „Execution Units“ und „Execution States“. Eine Execution Chain repräsentiert eine komplette Raytracing-Pipline und besteht aus einer Reihe von Execution Units. Jede Execution Unit wiederum ist eine Kette von Execution States die nacheinander ausge-

führt werden. Gängige Execution States sind die Strahlerzeugung, das Verfolgen der Strahlen und das Shading. Das Augenblick-SDK bietet die Möglichkeit, den Execution Units an wahlfreien Stellen beliebige eigene Execution States beizufügen und/oder die normalen Execution States zu ersetzen. Für eine Execution Unit kann festgelegt werden ob sie (und damit alle darin enthaltenen Execution States) pro Frame oder pro Tile ausgeführt werden soll. Ein Tile ist ein 32x32 Pixel großer Bildausschnitt. Die Tiles werden mittels Multithreading auf die zur Verfügung stehenden Prozessorkerne verteilt. Für den Plugin-Programmierer bedeutet dies, dass sein Code automatisch die Geschwindigkeitsvorteile durch Paralellisierung auf Mehrkernprozessoren ausnutzt.

Der übliche Programmablauf in Augenblick sieht somit wie folgt aus:

- Eine Datei wird über die Augenblick-GUI zum Laden ausgewählt
- Ein für den Dateityp zuständiger Loader erzeugt Geometrieobjekte im Szenengraphen von Augenblick
- Der Renderer wird gestartet, welcher die Rendering-Pipeline pro Tile abarbeitet
  - Für jeden Pixel des Tile wird ein Strahl ausgehend vom optischen Zentrum ein Sehstrahl erzeugt
  - Ein Raytracing-Algorithmus berechnet die Schnitte der Strahlen mit den Objekten
  - Ein Shaderalgorithmus berechnet für die Auftreffpunkte der Strahlen und das dort gegebene Material eine Farbe.
- Die Augenblick-GUI stellt den fertigen Framebuffer auf dem Bildschirm dar

### 2.5.1 Plugin-API

Die Plugin-API von Augenblick ist ein Satz von abstrakten Klassen aus denen eigene Plugins abgeleitet werden können. Plugins können den Funktionsumfang von Augenblick in folgenden Bereichen erweitern:

- Execution State Plugins erweitern Augenblick um zusätzliche Ausführungsschritte in der Rendering-Pipeline
- Freeform Interface Plugins
- Geometry Plugins stellen Fähigkeiten bereit die notwendig sind um verschiedene Geometrietypen mittels Raytracing zu visualisieren
- GPU Execution Plugins

- Hierarchy Builder Plugins
- Hierarchy Traverser Plugins
- Loader Plugins ermöglichen Augenblick das Laden zusätzlicher Dateiformate
- Shader Plugins

### 2.5.2 Arbeitsteilung

Durch den Plugin-Mechanismus herrscht eine Arbeitsteilung zwischen dem Kern von Augenblick und den im Rahmen dieser Arbeit entwickelten Programmteilen.
Das im Rahmen dieser Diplomarbeit programmierte Loader-Plugin registriert sich in Augenblick als zuständig für den Datentyp „.vox“. Wird eine solche Datei ausgewählt, ruft Augenblick die Lade-Methode des Plugins auf, welche dann ein Geometrieobjekt in den Szenengraphen von Augenblick einhängt. Augenblick verwaltet den Szenengraphen für die Kamera, Lichtquellen und die Geometrie. Die normale Execution Chain von Augenblick wurde wie folgt ergänzt:
Augenblick erzeugt die von der Kamera aus zu verfolgenden Strahlen und schneidet sie mit den Bounding Volumes der Geometrieobjekte im Szenengraphen. Wenn es einen Schnitt eines Strahls mit dem Bounding Volume eines Objekts vom Typ „Voxel Geometry“ gibt, ruft Augenblick die im Rahmen dieser Arbeit entwickelte Raytracing-Methode des Voxel-Geometrieobjekts auf. Die Raytracing-Methode bekommt den schneidenden Strahl übergeben und ist dafür verantwortlich, den Strahl durch die Voxel-Geometrie zu verfolgen bis es einen Treffer gibt oder der Strahl das Geometrieobjekt wieder verlässt. Rückgabewert der Raytracing-Methode ist der Lambda-Wert zum Auftreffpunkt mit der Geometrie. Der Lambda-Wert ist das Abstandsmaß zum Schnittpunkt mit der Geometrie und entspricht einem Vielfachen des Abstandes von Kamerazentrum zur Bildebene.

Die Architektur von Augenblick sieht vor, dass jedem Geometrieobjekt genau ein Material zugewiesen ist. Da die komplette Voxel-Geometrie in einem einzigen Geometrieobjekt des Szenengraphen gespeichert wird, hat dies zur Folge dass alle Voxel für Augenblick das gleiche Material haben. Das komplette Voxel-Modell wäre also einfarbig.
Jeden Voxel in ein eigenes Geometrieobjekt im Szenengraphen umzuwandeln ist aufgrund des daraus resultierenden hohen Speicherverbrauchs[20] nicht sinnvoll. Ferner würde so jede Möglichkeit genommen, die

[20] Jede Bounding Box im Szenengraphen von Augenblick ist 32 Byte groß. Kapitel 3.4.4 geht näher darauf ein.

Schnitttests auf die Rahmenbedingungen des Voxel-Octree (achsenparallele, mittige Raumunterteilung) hin zu optimieren.
Um die Limitierung auf nur eine Farbe für alle Voxel zu umgehen war es notwendig, in der Raytracing-Methode den Inhalt des von Augenblick benutzten RGBA-Buffers direkt zu manipulieren. Der reguläre Shading Execution State von Augenblick liefert für das Standard-Material ein Graustufenbild zurück welches der Helligkeitsinformation im Bild entspricht. Um die Helligkeitsinformation beim Manipulieren des RGBA-Buffers nicht zu verlieren wird im Loader-Plugin ein eigener State in die Execution Chain eingefügt. Dieser verrechnet pro Tile die Helligkeitsinformation aus dem Shading-State mit der Farbinformation aus dem manipulierten RGBA-Buffer. Kapitel 3.6 geht näher auf die Verrechnung ein.

Der Vollständigkeit halber sei erwähnt, dass sich Augenblick ferner um die Darstellung auf dem Bildschirm, Interaktion mit dem Betriebssystem, Maussteuerung sowie die Benutzeroberfläche kümmert.

# 3 Die Verarbeitungskette

Im Rahmen der Recherche zu dieser Diplomarbeit wurde klar dass derzeit noch keine Verarbeitungskette für Octree-Voxelmodelle öffentlich verfügbar ist[21].
Eine komplette Verarbeitungskette würde aus

1. einem Modellierungswerkzeug,
2. einem Dateiformat,
3. einem Dateiformat-Parser,
4. einem Raytracer

bestehen.
Um die für diese Arbeit gesetzten Ziele zu erreichen mussten also alle Teile der Kette selbst entwickelt werden. Das Entwickeln eines Modellierungswerkzeugs lag allerdings außerhalb des im zeitlichen Rahmen Machbaren, daher wurde als Ersatz für das erste Glied der Kette ein Konverterprogramm geschrieben, welches beliebige Polygonmodelle in Voxel-Octree Datensätze überführt[22].
Dieses Vorgehen hat einerseits den Vorteil, dass die Vielzahl an vorhandenen Polygonmodellen als Datenquelle genutzt werden kann. Andererseits hat die Verwendung eines Konverterprogramms den Nachteil, dass alle Limitierungen der Polygonmodelle in das Voxelmodell übernommen werden. Die Vorteile der Voxelmodelle, insbesondere der mögliche hohe Detailgrad in feinsten Strukturen kann auf konvertierten Datensätzen nicht zur Entfaltung kommen. In den meisten erzeugten Voxelmodellen sind daher die Dreiecke aus denen sie entstanden sind noch erkennbar.

Die tatsächlich implementierten Verarbeitungskette und der Datenfluss der die Einzelprogramme verbindet wird von Abbildung 8 illustriert.

## 3.1 Das Konverterprogramm

Das grobe Vorgehen des Konverterprogramms ist wie folgt:

- Einlesen des Polygondatensatzes
- Erstellen einer Farbpalette
- Erzeugung des Voxel-Octree

[21] Es ist anzunehmen, dass vollständige Verarbeitungsketten in den Entwicklerstudios von id Software und co. existieren.
[22] Die Verwendung von DICOM-Datensätzen kam, wie in Kapitel 1.3.3 erläutert, nicht in Frage

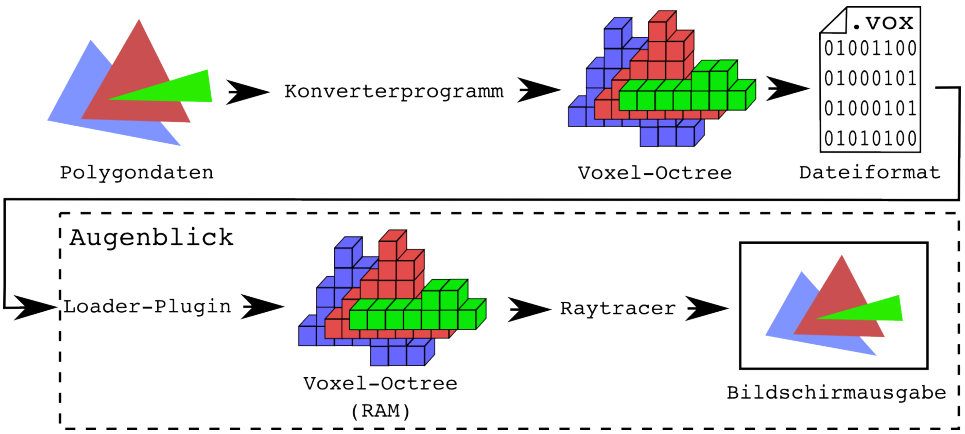


**Abbildung 8:** Die Einzelprogramme und der Datenfluss zwischen diesen

- Nachbearbeitung des Octree
- Serialisieren des Baumes in ein binäres Dateiformat

### 3.1.1 Einlesen des Polygondatensatzes

Das Konverterprogramm erwartet als Eingabe ein Polygonmodell im „Wavefront Object"-Dateiformat[23]. Die verwendeten Polygonmodelle sind frei verfügbare Beispielmodelle aus der „GLM Alias Wavefront OBJ file library"[24] von Frederic Devernay, Modelle aus dem „Tutors"-Programm von Nate Robins[25], sowie Modelle aus dem Stanford 3D Scanning Repository[26]. Die Modelle des Stanford 3D Scanning Repository liegen im „.ply"-Format vor, daher wurde zunächst das frei verfügbare Programm „ply_to_obj"[27] verwendet um die „.ply"-Dateien in „.obj"-Dateien umzuwandeln.
Die GLM Bibliothek in Version 0.3.1[28] wird dazu verwendet die „.obj"-Dateien zu parsen und in den Arbeitsspeicher einzulesen. Danach liegt das Polygonmodell mit allen Dreiecken und Materialien im Arbeitsspeicher vor und das Konverterprogramm kann mit seiner eigentlichen Arbeit beginnen.

[23] http://local.wasp.uwa.edu.au/ pbourke/dataformats/obj
[24] http://devernay.free.fr/hacks/glm/
[25] http://www.xmission.com/ nate/tutors.html
[26] http://www-graphics.stanford.edu/data/3Dscanrep
[27] http://people.sc.fsu.edu/ burkardt/c_src/ply_to_obj/ply_to_obj.html
[28] GLM geht in seinen Ursprüngen auf Nate Robins zurück. Die verbesserte Version 0.3.1 ist unter den Bedingungen der GPL 2.0 verfügbar. Gemäß den Lizenzbedingungen der GPL muss das Konverterprogramm als abgeleitetes Werk ebenfalls unter der GPL 2.0 lizenziert werden.

### 3.1.2 Erstellen einer Farbpalette

Farbpaletten sind eine Form der verlustbehafteten Kompression, bei der für jeden einzelnen Pixel, oder in diesem Fall Voxel, kein RGB-Farbwert sondern ein Index aus einer Palette abgespeichert wird. Die Palette hat einen beschränkten Umfang (meist 8 Bit = 256 Einträge). Wenn in dem Bild (in diesem Fall Voxel-Octree) mehr Farben verwendet werden als die Palette Einträge hat, müssen die Farben auf einen gemeinsamen Satz von Farben reduziert werden welcher in die Palette passt. Daher ist das Verfahren mitunter verlustbehaftet. Die Kompression kommt daher, dass alle Kombinationen der verfügbaren Bits ausgenutzt werden, während es beispielsweise bei dem RGB Farbraum mit seinen 16,7 Millionen Farben selten vorkommen wird dass in einem Bild alle diese Farben Verwendung finden. Im Fall des RGB Farbraums lässt sich der Speicherbedarf bei einer 8-Bit-Farbpalette von 24 Bit auf 8 Bit dritteln, mit je nach Bild unterschiedlich starkem oder gar keinem Verlust an Darstellungsqualität.

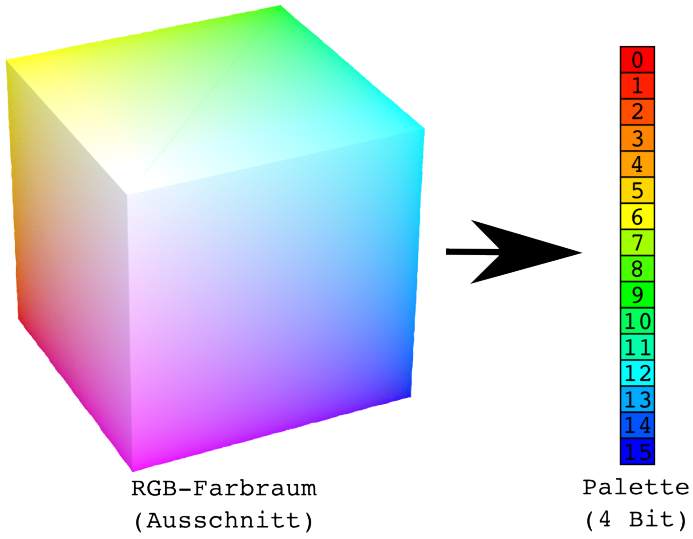


**Abbildung 9:** Reduktion eines Farbraums auf eine Palette[29]

Da ein typisches Voxelmodell Millionen von Voxeln enthält ist der Speicherverbrauch jedes einzelnen Voxels ein kritischer Faktor. Daher wurde im Konverterprogramm von vornherein die Kompression auf eine Farbpalette von zunächst 128 und im späteren Verlauf 32.768 Einträgen implementiert.

[29] Enthält eine Grafik aus: http://commons.wikimedia.org/wiki/File:RGB_color_solid_cube.png, Lizenz: Creative Commons Attribution ShareAlike 3.0

Der Vorgang des Reduzierens eines Bildes auf eine Farbpalette nennt sich Farbreduktion. Es gibt eine Reihe von Algorithmen die sich mit diesem Problem beschäftigen (siehe Median Cut von [Hec82], Local K-means von [Ver95] und [CY01]). Die Herausforderung besteht darin, die Reduktion so durchzuführen dass die Darstellungsqualität möglichst wenig beeinträchtigt wird. Dabei sind Anwendungsfälle mit einer fest vorgegebenen Palette und solche mit frei wählbarer Palette zu unterscheiden.
Bei fest vorgegebener Palette ist es das Einfachste, die Originalfarben anhand des euklidischen Abstandes auf die im Farbraum nächstgelegene Farbe der vorgegebenen Palette abzubilden[30]. Eine frei wählbare Palette hingegen hat den Vorteil, dass Teilbereiche des Farbraumes die im Originalbild nicht vorkommen durch keinen Eintrag in der Palette abgedeckt werden müssen. So können die Einträge in der Palette, bei gleicher Palettengröße, die verwendeten Bereiche des Farbraumes in feineren Nuancen abdecken. Die Verfahren zur Erzeugung einer frei wählbaren Palette sind allerdings aufwändiger.
Da die genannten Algorithmen sich nicht eins zu eins auf die hier gegebenen Rahmenbedingungen übertragen lassen, wurde ein eigener, iterativer Algorithmus zur Farbreduktion entwickelt, der entfernte Ähnlichkeiten mit dem Local K-means Algorithmus aufweist. Zu den hier gegebenen Rahmenbedingungen gehört unter anderem, dass die Qualität der generierten Farbpalette wichtig als die Laufzeit des Algorithmus ist, da das Konverterprogramm pro Modell nur einmalig ausgeführt werden muss und das Resultat dann dauerhaft auf der Festplatte gespeichert wird.

Das entwickelte Verfahren geht in zwei Schritten vor. Zunächst wird eine Liste aller im Polygonmodell verwendeten Farben erstellt. Anschließend werden in einer Schleife solange zwei Farben der Liste zu einer Mischfarbe vereint bis die Liste nur noch so viele Einträge hat wie die Farbpalette groß ist (siehe Abbildung 10).
Um die Liste aller im Polygonmodell verwendeten Farben (und deren Häufigkeit) zu erstellen wird der 3D-Scan-Konverter, auf den das nachfolgende Kapitel 3.1.3 näher eingeht, auf das komplette Modell angewandt um für jeden Voxel seine Farbe zu bestimmen. Jede verwendete Farbe wird in eine Liste eingetragen und ein Verwendungszähler (usage count) für diese Farbe inkrementiert. Die generierten Voxel werden sofort wieder verworfen, da dieser Trockenlauf ausschließlich dem Erfassen der Farben dient.
Die nachfolgende Schleife hat die Aufgabe, die Liste aller verwendeten Farben auf den Umfang der Palette zu reduzieren. Um dies zu erreichen werden pro Schleifendurchlauf diejenigen zwei Farben aus der Liste zu einer Mischfarbe vereint, deren Vereinigung die möglichst Geringste nega-

[30] Wenn man diese Abbildungen in einem zweidimensionalen Farbraum visualisiert entsteht ein Voronoi-Diagramm

tive Auswirkung auf die Darstellungsqualität hat. Mit anderen Worten: Die zwei Farben die sich am ähnlichsten sind.

Abbildung 10 zeigt wie in drei Schritten eine Liste von 19 Farben auf

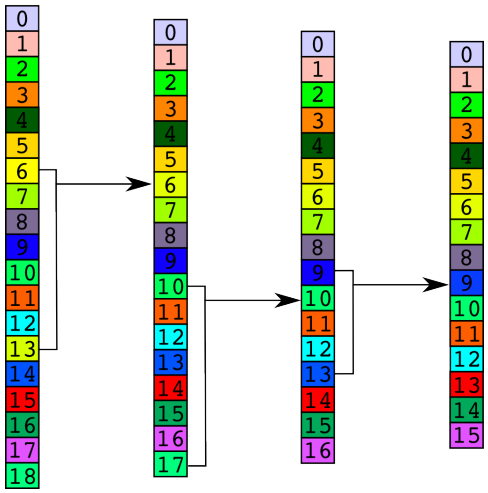


**Abbildung 10:** Iterative Reduktion eines Farbraums auf eine gewünschte Größe

eine Palette von 16 Farben schrittweise durch Auswahl zweier zu mischender Farben reduziert wird. Die Auswahl der zwei zu mischenden Farben ist allerdings kein triviales Unterfangen. Zum Einen ist die Frage, was ein geeignetes Ähnlichkeitsmaß zweier Farben ist. In dieser Implementation wird der Einfachheit halber der euklidische Abstand der Farben im RGB-Farbraum als Ähnlichkeitsmaß verwendet. Dabei werden wahrnehmungspsychologische Erkenntnisse, wie der Umstand dass Menschen Helligkeitsunterschiede stärker wahrnehmen als Farbunterschiede[31] außen vor gelassen, da dies den Umfang dieser Arbeit gesprengt hätte. Für Details siehe zum Beispiel [ABNS93]. Um das Verfahren weiter zu optimieren würde sich beispielsweise das Arbeiten im YUV-Farbraum anbieten, da dort die Helligkeit von der Farbinformation getrennt vorliegt.

Zum Zweiten werden, um die negativen Auswirkungen auf die Darstellungsqualität weiter zu reduzieren, die Farben mit einem Driftzähler als zusätzlicher Eigenschaft versehen. Der Driftzähler wirkt sich dämpfend auf die Wahrscheinlichkeit einer Farbe aus, zur Vermischung ausgewählt zu werden. Der Driftzähler wird jedes Mal wenn eine Farbe zur Mischung ausgewählt wird inkrementiert. Die Mischfarbe erhält dabei die Summe der Driftzähler der beiden Ursprungsfarben. D.h. von jeder Mischfarbe ist bekannt aus wievielen vorangegangenen Mischungen diese Farbe hervorgegangen ist. Er fungiert somit als ein Indikator dafür wie stark eine

[31] Dies hängt unmittelbar mit der Anzahl, Verteilung und Empfindlichkeit der Stäbchen (Helligkeit) und Zäpfchen (Farbe) in der Netzhaut zusammen.

Mischfarbe bereits von den Ursprungsfarben abweicht. Je stärker die Abweichung bereits ist, umso seltener sollte eine Farbe zur erneuten Mischung ausgewählt werden.
Beim Mischen der beiden Farben werden diese mit ihrer Verwendungshäufigkeit gewichtet.

$$x^m = \frac{x^1 * v^1 + x^2 * v^2}{v^1 + v^2} \tag{3}$$

Das heißt, der Farbwert $x^m$ auf Farbkanal x der Mischfarbe ergibt aus dem mit den Verwendungshäufigkeiten $v^1$ und $v^2$ gewichteten arithmetischen Mittel der Farbwerte $x^1$ und $x^2$ der beiden zu mischenden Farben.
Der Verwendungszähler jeder Farbe wird beim Erstellen der Liste aller verwendeten Farben erfasst. Hinter der Gewichtung steckt die Überlegung, dass Farben die im Modell häufiger vorkommen für die Darstellungsqualität von größerer Bedeutung sind und daher weniger von den ursprünglichen Originalfarben abweichen sollten als seltener vorkommende Farben.

### 3.1.3 Erzeugen des Voxel-Octree

Nachdem die Farbpalette für das gegebene Polygonmodell erzeugt worden ist kann die Erzeugung des Voxel-Octree beginnen.
Der hier implementierte Algorithmus ist eine auf den dreidimensionalen Raum übertragene Scan-Konvertierung. Die Scan-Konvertierung ist ein fundamentaler Bestandteil der Rasterisierung. In der Rasterisierungs-Pipeline wird der Scan-Konverter verwendet um die einzelnen, auf die Bildebene projizierten Dreiecke in diskrete Pixel umzuwandeln, d.h. sie zu rasterisieren.
Die Scan-Konvertierung funktioniert nur mit Pixeln einheitlicher Größe. Daher wird in der hier entwickelten, auf den dreidimensionalen Raum übertragenen Variante, ebenfalls mit einer festen Voxelgröße gearbeitet. Das heißt, der Octree hat eine beim Start des Programms frei wählbare, definierte maximale Tiefe. Dieser Eingabeparameter des Konverterprogramms ist insofern bedeutsam, als in Abhängigkeit dieses Wertes der resultierende Voxel-Octree aus hunderten, hunderttausenden oder hundertmillionen Voxeln bestehen wird. Ferner wird durch die maximale Baumtiefe die Unbalanciertheit des Octree in einem gemäßigten Rahmen gehalten.
Eine frühere Arbeit zu diesem Thema ist [KS87], in der ein auf den dreidimensionalen Raum übertragener Bresenham-Algorithmus zur 3D-Scan-Konvertierung in Uniform-Grids vorgeschlagen wird. [Oli08] legt sich nicht auf ein Verfahren fest, schlägt neben der 3D-Scan-Konvertierung alternativ eine ungleichmäßige Generierung der Voxel auf den Eckpunkten eines durch Subdivision mehrfach unterteilten Dreiecks vor.

Zum Erzeugen des Voxel-Octree für das ganze Polygonmodell iteriert das Programm über die Liste aller Dreiecke und ruft den 3D-Scan-Konverter für jedes Dreieck auf. Die durch die Scan-Konvertierung erzeugten Voxel werden an der passenden Stelle auf der definierten maximalen Ebene in den Baum eingehangen. Da der Octree anfangs komplett leer ist werden noch nicht vorhandene Vaterknoten und Geschwister, falls notwendig, mit angelegt. Zur Findung der „passenden Stelle“ im Baum wird der gleiche Suchalgorithmus verwendet wie später im Raytracer, bloß dass der Algorithmus in diesem Fall die Knoten und Blätter im Baum während der Suche erzeugt.

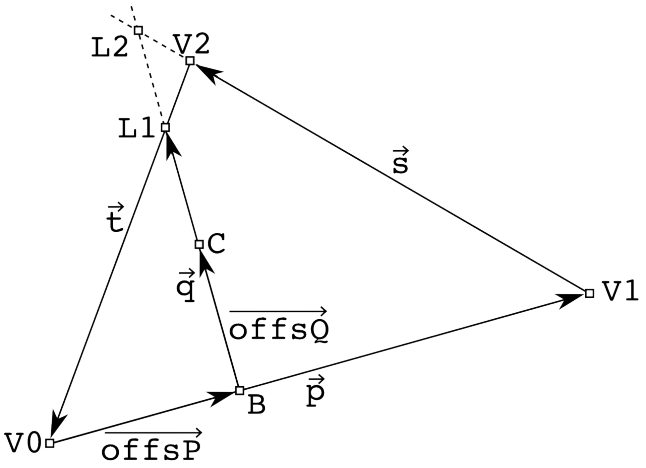


**Abbildung 11:** Schema der 3D-Scan-Konvertierung

Abbildung 11 zeigt das Schema der 3D-Scan-Konvertierung. Dargestellt ist ein Dreieck mit den Eckpunkten $V0$, $V1$ und $V2$. $\vec{p}$ ist die Hypotenuse des Dreiecks. $\vec{s}$ und $\vec{t}$ sind die Katheten des Dreiecks, gegen den Uhrzeigersinn orientiert. $\vec{q}$ ist der orthogonal zu $\vec{p}$ in der Ebene die das Dreieck aufspannt liegende Vektor mit Ursprung $B$. $off\vec{s}P$ ist der Vektor vom Ursprung der Hypotenuse ($V0$) zum Basispunkt $B$, $off\vec{s}Q$ ist der Vektor vom Basispunkt $B$ zum aktuellen Betrachtungspunkt $C$.

Der implementierte Scan-Konverter erzeugt die Voxel in zwei geschachtelten Schleifen. Die äußere Schleife inkrementiert $off\vec{s}P$ (und somit $B$) entlang der Hypotenuse $\vec{p}$. $B$ bildet den Ausgangspunkt für die innere Schleife, die $off\vec{s}Q$ (und somit $C$) entlang von $\vec{q}$ inkrementiert. Jeder Durchlauf der inneren Schleife liefert also einen neuen Punkt $C$ welcher in der Ebene liegt die das Dreieck aufspannt. Zu den Koordinaten von $C$ wird im Voxel-Octree ein neuer Voxel eingetragen.

Für die präzise Annäherung der Form des Dreiecks ist die Abbruchbedingung der inneren Schleife entscheidend. In der äußeren Schleife wird für jeden Vektor $\vec{q}$ dessen Schnitt mit $\vec{s}$ und $\vec{t}$ berechnet. Im dreidimensionalen Raum schneiden sich Geraden selten, da aber alle Vektoren in einer Ebene liegen kann die Schnittpunktberechnung auf den zweidimensionalen Fall reduziert werden. $L1$ und $L2$ sind die Schnittpunkte der Geraden entlang $\vec{q}$ mit $\vec{s}$ und $\vec{t}$. Die Lambda-Werte zu den Schnittpunkten $L1$ und $L2$ sind das Ergebnis der Schnittpunktberechnungen, der Kleinere von beiden gibt an ab welcher Länge $of\vec{f}sQ$ das Dreieck verlässt. Somit wird der kleinere Lambda-Wert als Abbruchkriterium der inneren Schleife verwendet. Eine Behandlung von negativen Lambdas ist nicht notwendig. Beide Lambdas sind immer positiv, da $\vec{q}$ senkrecht auf der Hypotenuse steht und die inneren Winkel der Hypotenuse zu den Katheten immer kleiner als $90°$ sind. Die äußere Schleife ist beendet wenn $of\vec{f}sP$ die Länge von $\vec{p}$ erreicht hat.
Die Inkrementationsschritte der Schleifen hängen von der Dimension der Voxel ab und sind genau so gewählt, dass der Nächste berechnete Punkt garantiert in einem benachbarten Voxel mit Kontaktfläche zum Vorgänger liegt. Es kommt weder dazu dass ein Voxel mehrfach angelegt wird, noch kann es zu Lücken in der Abtastung kommen. Dies wird erreicht, indem die Vektoren um die inkrementiert wird so skaliert werden, dass ihr Wert auf der „dominierenden" Achse eins beträgt. Die dominierende Achse ist diejenige Achse, auf der ein Vektor den größten Zahlwert trägt. Beispiel:

$$\begin{pmatrix} 0.5 \\ 0.3 \\ 0.1 \end{pmatrix} \div 0.5 = \begin{pmatrix} 1 \\ 0.6 \\ 0.2 \end{pmatrix} \tag{4}$$

Durch die Division durch den größten der drei Werte ist sichergestellt, dass der nächste Punkt genau eine „Voxel-Zeile" bzw. eine „Voxel-Spalte" weiter liegt. Abbildung 12 illustriert diesen Umstand für den zweidimensionalen Raum. Abbildung 13 illustriert die sukzessive Umwandlung der Fläche des Dreiecks in Voxel.

In jedem Voxel wird ein Farbpalettenindex und eine Normale abgespeichert. Der Farbpalettenindex entspricht dem Farbpalettenindex des Materials des Dreiecks. Aus interpolierten Texturkoordinaten der Eckpunkte die Farbe eines Texels aus einer Textur auszulesen ist möglich, aufgrund von anhaltenden Problemen mit dem Einlesen von Texturen in der GLM-Bibliothek in Version 0.3.1, sowie wenigen verfügbaren texturierten Modellen wurde dies jedoch nicht umgesetzt. Texturierte Modelle hätten zwar den erzielbaren hohen Detailreichtum der Volumendaten besser zur Geltung gebracht, das Erreichen einer höheren Darstellungsqualität durch detailreiche Texturen musste aber hinter den Entwicklungszielen der Optimierung des Speicherverbrauchs und der Geschwindigkeit hinten

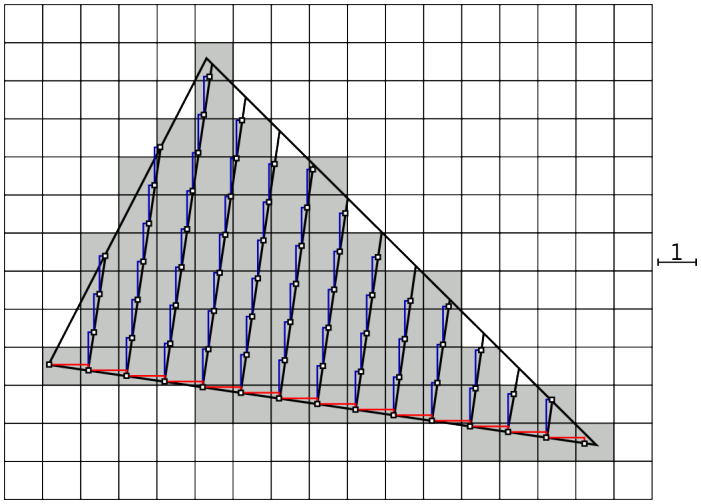


**Abbildung 12:** Flächendeckende Abtastung in einem gleichmäßigen Gitter entlang zweier orthogonaler Vektoren

an stehen.
Die Normale eines jeden Voxels ist eine aus den Eckpunktnormalen bilinear interpolierte Normale. Die Normale wird in diskreten Kugelkoordinaten mit jeweils 8 Bit für die Winkel $\theta$ und $\varphi$ kodiert (siehe Abbildung 14).

Wie Abbildung 12 zeigt, wird die Fläche des Dreiecks durch mehrere sich berührende „Zeilen“ von Voxeln angenähert. In einigen Details weicht der entwickelte 3D-Scan-Konverter vom 2D-Scan-Konverter ab. Im Original wird das Dreieck auch über zwei geschachtelte Schleifen rasterisiert, bloß sind dabei die Achsen entlang derer iteriert wird fest vorgegeben. Die äußere Schleife läuft entlang der Y-Achse, die innere entlang der X-Achse. Diese feste Anordnung der Scan-Achsen ist im Bildraum optimal, da gesichert ist dass das Dreieck senkrecht zur Z-Achse in der Bildebene liegt. Im dreidimensionalen Raum können die Dreiecke jedoch beliebig im Raum orientiert sein, eine Abtastung entlang festgelegter Achsen würde zu sehr problematischen Ergebnissen führen. So würde beispielsweise ein Dreieck welches senkrecht zur Y-Achse steht durch eine einzige Zeile von Voxeln angenähert, in der Draufsicht würden große Lücken in der Fläche des Dreiecks sichtbar. Durch die Wahl der Hypotenuse und der dazu senkrechten als Achsen kann der 3D-Scan-Konverter eine optimale Abtastung jedes Dreiecks, unabhängig von seiner Orientierung im Raum, garantieren.

[32]Gemeinfreies Bild aus http://en.wikipedia.org/wiki/File:Coord_system_SE_0.svg

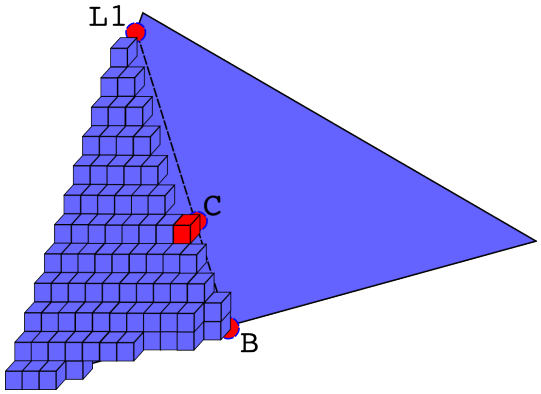


**Abbildung 13:** Diskretisierung einer Dreiecksfläche in Voxel im 3D-Scan-Konverter-Verfahren

Nachteilig an dem entwickelten Verfahren ist die etwas kompliziertere und damit anfälligere Mathematik. Der 3D-Scan-Konverter wurde auf allen zur Verfügung stehenden Polygonmodellen ausprobiert. Dabei zeigten sich Probleme mit Dreiecken die zu einer Linie zusammengefallen sind (Dreieck spannt keine Fläche auf, somit kann $\vec{q}$ nicht mehr eindeutig bestimmt werden) oder Dreiecke deren Eckpunkte so nah beieinander liegen dass die mathematischen Operationen aufgrund der begrenzten Genauigkeit von einfachen Gleitkommawerten unsinnige Ergebnisse lieferten.
Ersteres Problem konnte gelöst werden, in dem zu einer Linie zusammengefallene Dreiecke ausschließlich entlang der Hypotenuse abgetastet werden, das Verfahren verkommt in diesem Fall zu einer 3D-Linien-Rasterung. Dem Problem mit numerischer Ungenauigkeit bei zu kleinen Dreiecken konnte nur durch das Setzen und Testen bestimmter „magischer“ Grenzwerte begegnet werden. Auf Dreiecke die einen der Grenzwerte unterschreiten wird der Scan-Konverter dann gar nicht angewandt. Lediglich ein Eckpunkt des Dreiecks wird dann in einen Voxel umgewandelt. Da diese Dreiecke winzig klein sind wären sie bei den verwendeten Auflösungen des Voxelrasters sowieso nur durch einen einzigen Voxel angenähert worden. Aus der mathematischen Anfälligkeit des entwickelten Verfahrens ergibt sich als Konsequenz, dass das Konverterprogramm *zumindest theoretisch* keine einhundertprozentige Umwandlung aller Dreiecke in Voxel garantieren kann, auch wenn dies in der Praxis mit den hier getesteten Modellen keine Rolle spielte.

### 3.1.4 Nachbearbeitung des Octree

Nachdem der Voxel-Octree generiert worden ist bietet sich die Möglichkeit einer Nachbearbeitung bevor der Baum in eine Datei serialisiert wird.

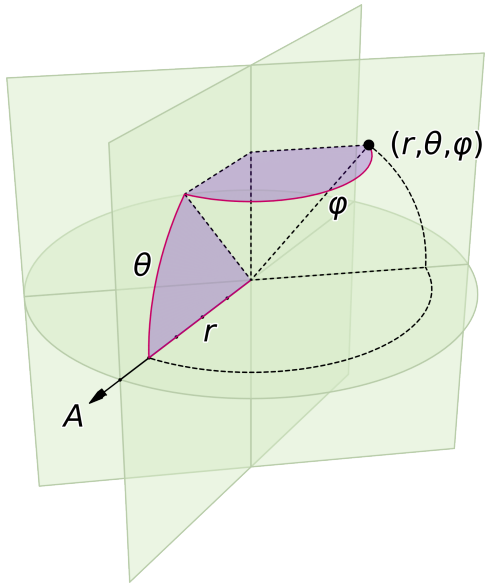


**Abbildung 14:** Bedeutung der Winkel Theta und Phi im Kugelkoordinatensystem[32]

Der erste Nachbearbeitungsschritt ist die Generierung der Detailstufen. Bis zu diesem Zeitpunkt besteht der Baum nur aus gefüllten Blättern auf der untersten Ebene des Baumes und transparenten Knoten in allen darüberliegenden Ebenen. Die bisher transparenten Ebenen des Octree können mit gröber detaillierten Versionen des Modells gefüllt werden, in dem jeder Voxel auf Ebene $x$ eine aus seinen acht Kinderknoten auf Ebene $x + 1$ verrechnete Farbinformation und Normale zugewiesen bekommt. Als Ergebnis erhält man ein und das selbe Modell in verschiedenen räumlichen Auflösungen auf den Ebenen des Baumes. Implementiert wurde der LOD mit einem rekursiven Algorithmus der in Postorder den Baum traversiert, dabei die Kinderknoten untersucht und aus den Farben und Normalen der Kinderknoten eine Farbe und Normale des Vaterknotens berechnet. Da der Farbraum auf die Palette des Modells begrenzt ist, wird für den Vaterknoten der Einfachheit halber derjenige Palettenindex gewählt den die meisten Kinderknoten verwenden. Die Normale des Vaterknotens wird aus den Normalen der Kinderknoten gemittelt.
Abbildung 15 zeigt einen Vergleich zweier Detailstufen eines Modells. Die linke Hälfte des Bildes zeigt ein Rendering des Modells mit einer Auflösung von $64^3$, d.h. Voxel-Knoten auf Ebene sieben des Baumes. In der rechten Hälfte des Bildes ist das gleiche Modell in seiner vollen Auflösung von $2048^3$ (Voxel-Blätter auf Ebene 13 des Octree) dargestellt.

Zum Zweiten ist eine Komprimierung überflüssiger Details ein

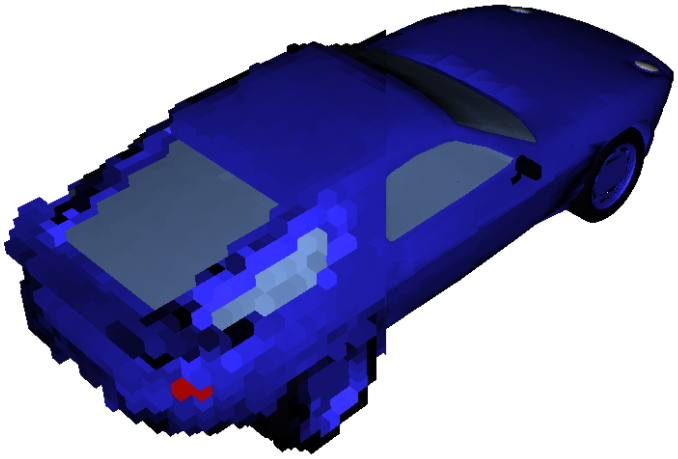

**Abbildung 15:** Vergleich zweier Detailstufen eines Modells

Nachbearbeitungsschritt der sich an dieser Stelle der Verarbeitungskette anbietet. Es ist möglich, dass beim Generieren des Baumes Teilbäume entstanden sind in denen alle Kinderknoten mit der gleichen Farbe und Normale gesetzt sind. In diesem Fall können die Kinderknoten gelöscht werden (siehe Abbildung 16). Der LOD-Algorithmus wird die fragliche Farbe und Normale bereits für den Vaterknoten gesetzt haben.

Der dritte mögliche Nachbearbeitungsschritt ist ein Füllen der Modelle. Da mit den Dreiecken als Grundlage nur die Hüllen der Modelle in Voxel umgewandelt werden, sind die resultierenden Voxelmodelle in ihrem Inneren nicht ausgefüllt. Eine Nachbearbeitung mit einem Algorithmus der den Innenraum einer geschlossenen Hülle füllt (z.B. Flood-Fill-Algorithmus) wäre denkbar und hätte auch einen kleineren, positiven Nebeneffekt auf die Darstellungsqualität (siehe Kapitel 4.2.1), wurde hier aber aus Zeitmangel und den nur marginalen Auswirkungen auf die Darstellungsqualität nicht implementiert.

### 3.1.5 Serialisieren des Baumes in ein binäres Dateiformat

Nachdem der Voxel-Octree fertiggestellt wurde kann im Dateisystem eine neue Datei angelegt und der ganze Baum von einem rekursiven Algorithmus in einer Preorder-Serialisierung abgespeichert werden. Auf den Aufbau und Inhalt der Datei geht nachfolgendes Kapitel ein.

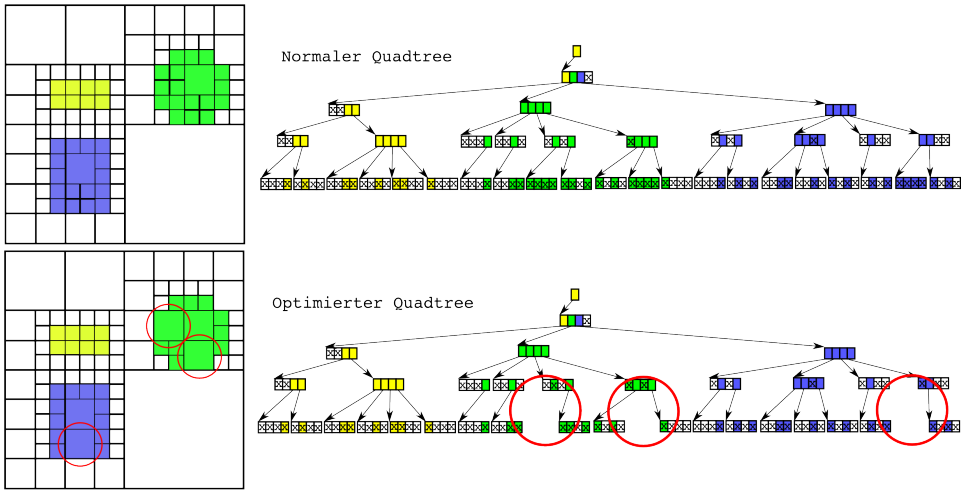


**Abbildung 16:** Kompression überflüssiger Voxel in einem Quadtree

## 3.2 Das Voxel-Dateiformat

Zur dauerhaften Speicherung der Voxel-Modelle bedarf es eines Dateiformats in dem der Voxel-Octree kodiert abgespeichert werden kann. Da es, wie in Kapitel 3 erwähnt, bisher keine öffentlich verfügbare Verarbeitungskette für Voxel-Octrees gibt, musste ein eigenes Format entwickelt werden. Dies hat den positiven Nebeneffekt, dass das Format voll auf die Anforderungen des verwendeten Voxel-Octree zugeschnitten werden konnte.
Im Laufe der Entwicklung haben sich diese Anforderungen mehrfach verändert, weshalb es insgesamt vier Versionen des Formats gibt. Im folgenden ist Version vier des Dateiformats näher beschrieben. Allen Versionen gemein ist die Dateiendung „.vox“.

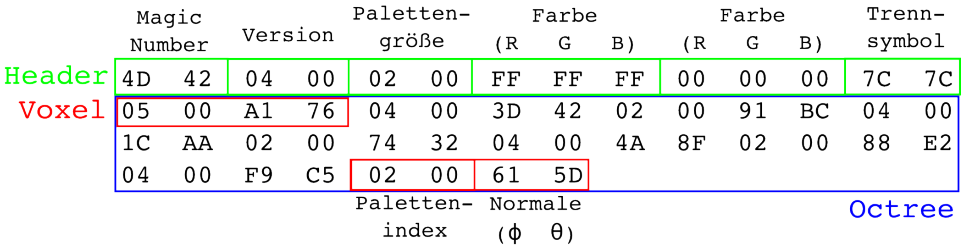


**Abbildung 17:** Beispiel einer „.vox“-Datei in hexadezimaler Schreibweise

Abbildung 17 zeigt eine simple Beispieldatei in hexadezimaler Schreibweise. Jedes Zeichenpaar repräsentiert ein Byte. Im folgenden wird auf den Header (grüne Umrandung) und den Nutzdatenbereich (blaue Umrandung) näher eingegangen.

### 3.2.1 Der Header

Der Header des Dateiformats dient zum Ersten der eindeutigen Identifikation der Datei und zum Zweiten enthält er Metadaten.

Der Inhalt im Einzelnen:

1. Magic Number.
   Die ersten zwei Byte der Datei müssen der Magic Number „0x4D42“ (ISO 8859-1 Zeichenfolge „MB“) entsprechen. Magische Zahlen sind eine häufig verwendete Methode um Dateien mit einer hohen Wahrscheinlichkeit einem bestimmten Dateityp zuzuordnen, unabhängig von der leicht manipulierbaren Dateiendung. So beginnt beispielsweise jede gültige S-DOS Executable und Windows Portable

Executable (.com, .exe, .dll) mit der Zeichenfolge „MZ“, Dateien des Archivierungsformates Zip (.zip, .jar) mit der Zeichenfolge „PK“ und Dateien des Grafikformats PNG (.png) mit der Zeichenfolge „.PNG“[33].
In diesem Fall ist die Magic Number 16 Bit lang, die Wahrscheinlichkeit dass eine beliebige Datei mit unbekanntem oder zufälligem Inhalt mit diesem Bitmuster beginnt beträgt somit 1 zu 65.536. Sollte die Datei nicht mit dieser Magic Number beginnen, bricht der Parser mit einer Fehlermeldung ab.

2. Versionsnummer.
   Dieses zwei Byte große Feld enthält die Versionsnummer. Sie dient dazu die verschiedenen Versionen des Formats zu identifizieren. Das Loader-Plugin kann anhand dieser Information die verschiedenen Versionen auseinander halten und abwärtskompatibel bleiben.

3. Palettengröße.
   Dieses zwei Byte große Feld gibt an, wie viele Einträge die nachfolgende Farbpalette besitzt.

4. Farbpalette.
   Die Farbpalette für dieses Modell. Der Umfang in Byte ergibt sich aus der Palettengröße mal drei, da jeder Eintrag aus einem 24-Bit RGB-Farbwert besteht.

5. Trennsymbol.
   Dieser zwei Byte große Bereich in unmittelbarem Anschluss an die Farbpalette markiert eindeutig das Ende des Headers. Für die Maschinenlesbarkeit des Formats spielen diese Bytes keine Rolle und können ignoriert werden, da die Länge des Headers mit $8 + 3 * Palettengroesse$ Bytes eindeutig definiert ist. Das Trennsymbol dient vielmehr dem menschlichen Betrachter zur Übersichtlichkeit bei dem Analysieren des Dateiinhalts in einem Hex-Editor.

### 3.2.2 Der serialisierte Baum

Auf den Header folgt der serialisierte Voxel-Octree. Sinn und Zweck einer Serialisierung ist es, eine Datenstruktur nach festen Regeln in einer eindimensionalen Folge abzuspeichern, so dass nachträglich die ursprüngliche Datenstruktur aus der Reihenfolge der Daten wieder erschlossen (deserialisiert) werden kann. Persistente Speichermedien wie Festplatten, Magnetbänder oder optische Medien sind eindimensional und können zu jedem Zeitpunkt nur an einer Stelle gelesen oder beschrieben werden

[33] Für eine umfangreiche Auflistung siehe http://www.garykessler.net/library/file_sigs.html. Die Magic Number 0x4D42 wird von keinem dort aufgeführten Format belegt.

(auch wenn Festplatten mitunter physisch über mehrere Schreib- und Leseköpfe verfügen). Ferner ist der sukzessive Zugriff auf nachfolgende Daten schneller als der Zugriff an beliebiger Stelle. Dieses Konzept hat sich in Form der Datei und dem Streaming-Konzept von Unix bis heute erhalten. Dateien sind eindimensionale Datenströme, die von Prozessen Byte für Byte gelesen oder beschrieben werden können.
Allerdings ist es nicht zwingend notwendig den Voxel-Octree zu serialisieren. Auch der Arbeitsspeicher mit seinem wahlfreien Zugriff ist aus Anwendungssicht eindimensional[34], da jedes Byte über *eine* Adresse referenziert wird. Daher wäre es denkbar die Datenstruktur so wie sie im Arbeitsspeicher vorliegt auf die Festplatte zu kopieren. Da die Voxel-Octree-Datenstruktur des Konverterprogramms und des Voxel-Raytracers im Arbeitsspeicher identisch sind, könnte beim Einlesen der Datei ihr Inhalt eins zu eins in den Arbeitsspeicher zurück kopiert werden. Das Laden eines Datensatzes wäre auf diese Weise schneller fertig als wenn er erst noch deserialisiert werden müsste.
Dieses Vorgehen würde allerdings voraussetzen dass der komplette Baum an einem Stück im Arbeitsspeicher vorliegt, da in der Datei ja ein kontinuierlicher Datenstrom gespeichert wird und dieser nicht fragmentiert sein kann. Je nach Umfang des Baumes mehrere Gigabyte an Arbeitsspeicher an *einem Stück* zu allozieren erweist sich in der Praxis allerdings als problematisch[35]. Neben dieser praktischen Hürde hat eine direkte Speicherkopie jedoch den erheblichen Nachteil dass sie $100 - 200\%$ mehr Speicherplatz verbraucht als die hier implementierte serialisierte Form. Ferner hat sich in der Praxis gezeigt, dass die benötigte Zeit zum Deserialisieren der Dateien vernachlässigbar ist (wenige Sekunden für Modelle mit Zehnmillionen von Voxeln).

Als Form der Serialisierung wurde Preorder-Serialisierung des Baumes gewählt (siehe Abbildung 18). Bei Preorder-Serialisierung folgen Kinderelemente unmittelbar auf ihr Vaterelement, noch vor den Geschwistern des Vaterelementes.
Preorder-Serialisierung und alternative „Depth-First"-Serialisierungen wie Inorder und Postorder oder die Breadth-First-Serialisierung sind im Hinblick auf Speicherverbrauch und Geschwindigkeit der Serialisierung gleichwertig. „Breadth-First"-Serialisierung ist allerdings ungleich schwieriger umzusetzen, da es sich anders als die Depth-First Ansätze nicht rekursiv implementieren lässt.

[34] Physisch ist der RAM zweidimensional, da die Bytes auf den Chips über Zeilen- und Spaltennummern adressiert werden

[35] Ein malloc() von einer Milliarde Byte hat gute Chancen das Programm zum Absturz zu bringen. Genauer gesagt wird malloc() einen Zeiger auf Null zurück geben wenn das Betriebssystem die geforderte Speichermenge nicht in vollem Umfang und an einem Stück bereitstellen kann.

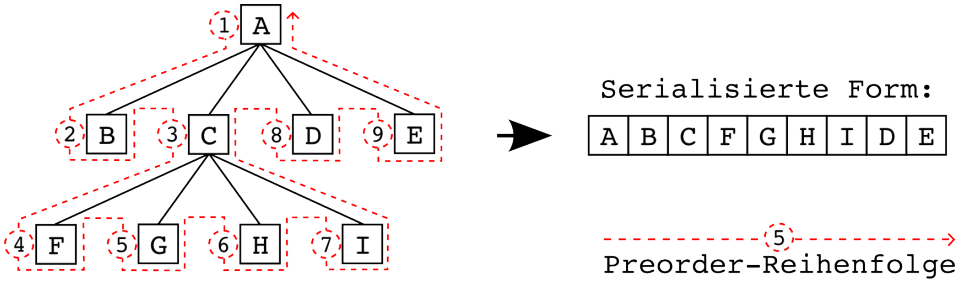


**Abbildung 18:** Preorder-Serialisierung eines Quadtree

In der konkreten Umsetzung ergibt sich die Speicherersparnis der Serialisierung gegenüber einem direkten Speicherabbild daraus, dass für die serialisierten Voxel die Zeiger auf die Kinderelemente entfallen können. Wie Kapitel 3.4.6 zeigen wird, verbrauchen die hier verwendeten Voxel auf einem 32-Bit Betriebssystem acht Byte, wobei vier Byte auf die Nutzdaten und vier Byte für einen Zeiger auf die Kinderelemente entfallen. Auf einem 64-Bit Betriebssystem sind es insgesamt zwölf Byte, vier Byte für die Nutzdaten und vier Byte für den Zeiger. In serialisierter Form nimmt jeder Voxel hingegen nur vier Byte in Anspruch (31 Bit Nutzdaten, 1 Bit Metadaten). Daraus ergibt sich, dass das direkte Speicherabbild auf einem 32 Bit-Betriebssystem $100\%$ und auf auf einen 64-Bit Betriebssystem $200\%$ größer wäre als die serialisierte Form.

Das Metadaten-Bit in der serialisierten Form ist ein Blatt-/Knoten-Marker. Wenn das Bit gesetzt ist, handelt es sich bei diesem Voxel um einen Knoten im Baum. Dieser Voxel hat also Kinder, somit ist der nachfolgende Voxel das erste Kind dieses Voxels. Wenn das Bit nicht gesetzt ist handelt es sich bei diesem Voxel um ein Blatt im Baum, daher ist der im Datenstrom nachfolgende Voxel das nächste Geschwister in der Reihe. Nach dem letzten Geschwister kommt das nächste Kind des nächsten Vorfahren dessen Kinderknoten noch nicht vollständig sind (siehe Abbildung 18: Auf das Blatt „I“ folgt das Blatt „D“).

Zusammenfassend lässt sich feststellen, dass ein Bit an Metainformation pro Voxel, in Kombination mit der Konvention dass die Voxel in Preorder-Reihenfolge abgespeichert werden, ausreichend ist um im Loader-Plugin aus dem Datenstrom den kompletten Voxel-Octree zu rekonstruieren.

Wie Abbildung 17 zeigt, setzen sich die vier Byte eines jeden Voxel aus zwei Byte für den Farbpalettenindex und zwei Byte für die Normale in

diskreten Kugelkoordinaten zusammen. Das Metadatenbit ist in der Abbildung nicht visualisiert, es ist das niederwertigste Bit des Farbpalettenindex. Der Farbpalettenindex ist also tatsächlich nur 15 statt 16 Bit breit. Da es das niederwertigste Bit ist welches als Knoten/Blatt Marker dient, gilt als Faustregel: Ungerade Palettenindizes zeigen einen Knoten an, gerade Indizes ein Blatt. Der tatsächliche Palettenindex entspricht dem gespeichertern Wert geteilt durch zwei (Rechtshift um ein Bit). Mit diesen Informationen lässt sich rekonstruieren dass in Abbildung 17 die serialisierte Form eines Voxel-Octree aus neun Voxeln (Wurzel + acht Blätter) abgebildet ist. Die Wurzel hat Palettenindex 2 (schwarz), die Kinder sind abwechselnd schwarz/weiß.

## 3.3 Das Voxel-Loader-Plugin

Wie in Kapitel 2.5.2 bereits angeschnitten, kann über den Plugin-Mechanismus von Augenblick ein Loader-Plugin eingebunden werden und dieses seine Zuständigkeit für bestimmte Dateiendungen, in diesem Fall „.vox“, annoncieren. Wenn eine Datei mit dieser Endung in der Augenblick-Benutzeroberfläche ausgewählt wird, ruft Augenblick die Lade-Methode des Loader-Plugins auf und übergibt den Pfad zu der Datei. Die Lade-Methode des Plugins öffnet dann besagte Datei und beginnt mit der Auswertung. Sofern die Magic Number korrekt ist, splittet sich die Verarbeitung des Datenstroms anschließend auf, je nachdem welche Versionsnummer im Header eingetragen ist. Im Folgenden wird die Verarbeitung von Version vier des Formats näher erläutert.

Zunächst wird die Farbpalette aus dem Header ausgelesen. Anschließend wird ein leeres Voxel-Geometrieobjekt angelegt und eine Methode aufgerufen die den Datenstrom nach dem Header rekursiv in einen Voxel-Octree deserialisiert. Depth-First Serialisierung und Deserialisierung lassen sich elegant rekursiv implementieren. So hat der Code zum Deserialisieren des kompletten Voxel-Octree aus dem „.vox“-Dateiformat einen Umfang von lediglich 19 Zeilen (siehe Listing 1).

```
1  void deserialize(Voxel* voxel, File* fileStream) {
2      // get the palette index and normal
3      fread(&(voxel->paletteIndex),2,1,fileStream);
4      fread(&(voxel->theta),1,1,fileStream);
5      fread(&(voxel->phi),1,1,fileStream);
6
7      // if the palette index is uneven the voxel has children
8      if((voxel->paletteIndex)%2) {
9          voxel->children=(Voxel*)malloc(8*voxelSize);
10         for(int i=0; i<8; i++) {
11             deserialize(&(voxel->children[i]), fileStream);
12         }
13     } else {
14         voxel->children=NULL;
15     }
16
17     // shift the palette by one bit to the right
18     voxel->paletteIndex>>=1;
19 }
```

**Listing 1:** Rekursive Methode zum Deserialisieren von „.vox“-Dateien

Die Methode deserialize() in Listing 1 erwartet als Eingabeparameter

einen leeren Voxel und eine Datei aus der gelesen werden kann. In den Zeilen drei bis fünf werden vier Byte aus der Datei gelesen und als Palettenindex und Normale des Voxels gespeichert. Wenn der Palettenindex in Zeile acht ungerade ist (niederwertigstes Bit ist gesetzt), wird Speicher für acht Kinder allokiert (Zeile neun) und die Methode rekursiv für alle acht Kinder aufgerufen (Zeilen zehn bis zwölf). Anderenfalls ist der Zeiger auf die Kinder ein Zeiger auf NULL (Zeile 14). Zu guter Letzt wird der Palettenindex noch um ein Bit nach rechts verschoben, um den im serialisierten Format benötigten Kinder-Marker zu entfernen (Zeile 18).

Der fertige Voxel-Octree wird dem Voxel-Geometrieobjekt übergeben und dieses in den Szenengraphen von Augenblick eingehangen. Anschließend wird, wie in Kapitel 2.5.2 erwähnt, noch eine eigene Execution Chain angelegt. In dieser Execution Chain wird ein neuer Execution State eingefügt der am Ende der Verarbeitungskette zwei Pixelpuffer (pro Tile) verrechnet. Der eine Puffer enthält die Farbinformation, der andere die Helligkeitsinformation. Die Farbinformation kommt direkt aus dem Raytracer, die Helligkeitsinformation aus dem Shading State von Augenblick. Dieser Umweg ist notwendig, da anderenfalls das Voxelmodell auf eine einzige Farbe beschränkt wäre. Augenblick sieht pro Geometrieobjekt nur ein Material vor mit dem der Shading State arbeitet. Jeden Voxel in einem eigenen Geometrieobjekt abzuspeichern kam nicht in Frage, da dies 32 zusätzliche Byte pro Voxel gekostet hätte. Der Augenblick Szenengraph ist eine Bounding Volume Hierarchie, Millionen von Voxeln in einer BVH abzuspeichern hätte nicht nur den Speicherverbrauch explodieren lassen, es wäre auch aus Sicht des Raytracers inperformant. Augenblick hat einen „Hierarchy Traverser“ der sehr performant darin ist die Schnitttests mit den geschachtelten BVHs durchzuführen und durch die BVH abzusteigen. Mit einer BVH einen Voxel-Octree zu simulieren hätte aber bedeutet dass viele unnötige Schnitttests gemacht würden. Kapitel 3.5.1 erklärt im Detail wie der entwickelte Octree-Suchalgorithmus den Baum ohne jegliche Schnitttests traversiert.

Der Vollständigkeit halber sei an dieser Stelle bereits erwähnt, dass das Loader-Plugin des weiteren noch ein Uniform Grid von Zeigern auf Voxel einer bestimmten Ebene des Baumes anlegt. Dieses Uniform Grid dient der Beschleunigung des Raytracers, die Motivation dahinter erläutert Kapitel 3.5.

## 3.4 Die Octree-Datenstruktur

Die Datenstruktur welche das Loader-Plugin erzeugt ist ein Voxel-Octree. In diesem Kapitel soll erläutert werden warum die Wahl auf einen Octree fiel und warum andere Datenstrukturen keine Alternative waren. Alle hier aufgeführten Berechnungen gehen, sofern nicht explizit anders erwähnt, von einem 32-Bit Betriebssystem aus.

### 3.4.1 Uniform Grid

Ein „Uniform Grid“, zu deutsch gleichmäßiges Raster, ist die gängigste Datenstruktur für Voxel. So wie Pixel in einem gleichmäßigen zweidimensionalen Raster gespeichert werden, können auch Voxel in einem gleichmäßigen dreidimensionalen Raster gespeichert werden. Im Arbeitsspeicher können die Voxel sequentiell als dreidimensionales Array gespeichert und über die Arrayindizes direkt angesprochen werden. D.h. zu jeder Raumposition kann direkt die Position des Voxels im eindimensionalen Arbeitsspeicher bestimmt werden. Abbildung 19 illustriert diesen Umstand.

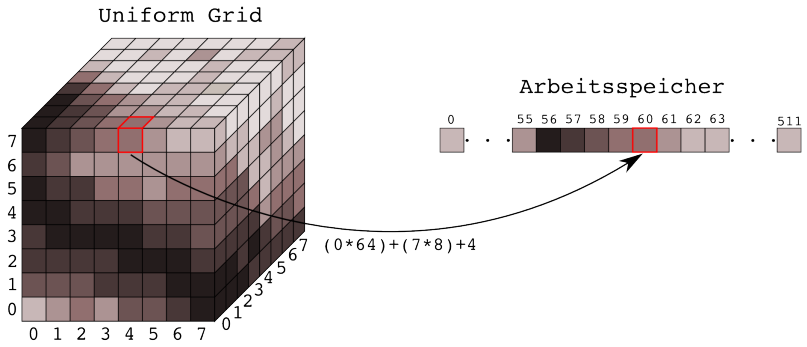


**Abbildung 19:** Ein Uniform Grid aus 8x8x8 Voxeln und der Zugriff im Speicher

Ein Uniform Grid zeichnet sich also durch sehr schnelle, direkte Zugriffe auf die einzelnen Voxel aus. Ferner müssen beim Raytracing von Voxeln in einem Uniform Grid keinerlei Schnitttests durchgeführt werden. Die Schnitttests sind bei Raytracing von Polygonen das Aufwändigste am ganzen Verfahren. Im Uniform Grid kann der Pfad des Strahls durch das gleichmäßige Raster mit einem einfachen 3D-Bresenham-Algorithmus abgetastet werden. Das Bresenham-Verfahren zeichnet sich dadurch aus, dass es ausschließlich mit Additionen und Vergleichen auskommt.
Der große Nachteil des Uniform Grid ist sein Speicherverbrauch. Mit steigender Auflösung wächst der Speicherverbrauch exponentiell. Für einen Raum mit einer Auflösung von $1024$ Voxeln pro Achse kommt man auf insgesamt $1024^3 = 1,07 * 10^9$ Voxel. Bei einer durchaus er-

strebenswerten Auflösung von $8192$ Voxeln pro Achse kommt man bereits auf $550$ Milliarden Voxel insgesamt. Bei vier Byte Nutzdaten pro Voxel würde dies einem Speicherverbrauch von $2,15$ Terabyte entsprechen, was selbst für moderne Festplatten problematisch wird, vom Arbeitsspeicher ganz abgesehen.
Wie in Kapitel 1.3.3 erwähnt, konzentrieren sich die Details in typischen Datensätzen der Computerspiele-Industrie entlang einer dünnen Oberfläche. Der Rest des Raumes ist homogen transparent. Sollen ganze Szenen in einer Hierarchie gespeichert werden muss obendrein das häufig anzutreffende „Teapot in a stadium"-Szenario berücksichtigt werden, welches die Verteilung der Information noch ungleichmäßiger macht[36]. Das Uniform Grid unternimmt nichts um diese Gleichförmigkeit zu komprimieren, in Bezug auf den Speicherverbrauch entspricht es also dem „Worst Case" für Voxeldaten. Das effiziente Kodieren und überspringen dieser Leerräume ist Sinn, Zweck und Herausforderung für jede Hierarchie und jede Voxel-Hierarchie misst sich daran wie viel kompakter sie ist als ein Uniform Grid gleicher Auflösung.

### 3.4.2 Der Octree

Der Voxel-Octree kann als eine Verbesserung des Uniform Grid betrachtet werden. Das gleichmäßige Raster wird durch einen Baum ersetzt welcher unterschiedlich große und geschachtelte Voxel enthält. Homogene Räume werden durch dem Volumen des Raumes angepasste, größere Voxel angenähert. Dieser einfache „Trick " reicht aus, um im Vergleich zum Uniform Grid bei gleicher Auflösung den Speicherverbrauch um ein Vielfaches zu senken. Abbildung 20 zeigt anhand eines Beispiels im zweidimensionalen Raum wie ein Quadtree im Vergleich zum Uniform Grid Voxel spart.

In dem Beispiel von Abbildung 20 fällt die Ersparnis in Voxeln mit $66,7\%$ moderat aus. Darüber hinaus gilt zu beachten, dass jede Hierarchie selbst Speicher verbraucht. In dem hier implementierten Octree besteht jeder Voxel aus vier Byte Nutzdaten und einem vier Byte großen Zeiger für die Datenstruktur, d.h. jeder Voxel im Octree ist doppelt so groß wie ein Voxel im Uniform Grid. In dem Beispiel aus Abbildung 20 ergäbe sich somit, in Bytes gemessen, nur noch eine Ersparnis von $(((256 * 4) - (85 * 8)) * 100)/(256 * 4) = 33,59\%$.

Im dreidimensionalen Raum ist die erzielbare Ersparnis aufgrund der zusätzlichen Dimension jedoch um einiges größer und das Beispiel aus Abbildung 20 ist recht künstlich gewählt. Tabelle 3 zeigt den Speicherverbrauch einiger Modelle im Octree und den erzielten realen Kompressionsfaktor im Vergleich zur Speicherung mit gleicher Auflösung im Uniform

[36] Die Teekanne in einem Stadium steht beispielhaft für Szenarien in denen sich kleine, komplexe Modelle in weniger komplexen, großen Umgebungen befinden.

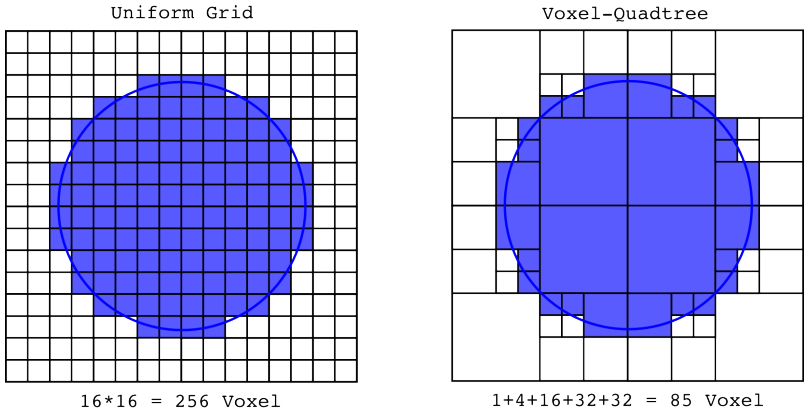


**Abbildung 20:** Vergleich der Voxelzahl in einem Uniform Grid und einem Quadtree

Grid. Wie die Tabelle zeigt sind die Modelle im Octree in Bytes um das Hundert- bis Tausendfache kleiner als im Uniform Grid. Es wird auch deutlich dass die Ersparnis mit wachsender Auflösung umso größer ausfällt.

| Modell | Tiefe | Auflösung | Voxel | Speicher | Ersparnis |
|---|---|---|---|---|---|
| Al | 13 | $4096^3$ | 131.827.905 | 0,98 Gb | 260x |
| Bunny | 13 | $4096^3$ | 69.798.129 | 532,5 Mb | 492x |
| Porsche | 11 | $1024^3$ | 4.504.929 | 34,3 Mb | 119x |
| Porsche | 12 | $2048^3$ | 18.095.137 | 138 Mb | 237x |
| Porsche | 13 | $4096^3$ | 72.457.401 | 552,8 Mb | 474x |
| Soccerball | 12 | $2048^3$ | 48.210.017 | 367,8Mb | 89x |
| World | 14 | $8192^3$ | 85.519.385 | 652,4 Mb | 3214x |

**Tabelle 3:** Speicherverbrauch und Kompressionsfaktor verschiedener Modelle im Octree

Die der Tabelle zugrunde gelegte Formel lautet $f = \frac{a^3}{v*2}$, wobei $a$ die Auflösung ist in der das Modell vorliegt, $v$ die Zahl der bei dieser Auflösung und diesem Modell insgesamt benötigten Voxel (Blätter und Knoten) und $f$ der erzielte Ersparnisfaktor ist. $v$ wird doppelt gewichtet, da, wie bereits erwähnt, die Voxel im Octree doppelt so viel Speicher belegen wie im Uniform Grid.

Tabelle 4 zeigt die Entwicklung des Speicherverbrauchs in einem Octree bei steigender Auflösung im schlimmsten Fall, d.h. wenn der Raum bis zur untersten Ebene vollkommen heterogen ist. Die Ebenen

| Ebenen | Auflösung | Blätter | Knoten | Total | Speicher |
|---|---|---|---|---|---|
| 1 | $1^3$ | 1 | 0 | 1 | 8 Byte |
| 2 | $2^3$ | 8 | 1 | 9 | 72 Byte |
| 3 | $4^3$ | 64 | 9 | 73 | 584 Byte |
| 4 | $8^3$ | 512 | 73 | 585 | 4.57 Kb |
| 5 | $16^3$ | 4.096 | 585 | 4.681 | 36,57 Kb |
| 6 | $32^3$ | 32.768 | 4.681 | 37.449 | 292,57 Kb |
| 7 | $64^3$ | 262.144 | 37.449 | 299.593 | 2,28 Mb |
| 8 | $128^3$ | 2.097.152 | 299.593 | 2.396.745 | 18,28 Mb |
| 9 | $256^3$ | 16.777.216 | 2.396.745 | 19.173.961 | 146,28 Mb |
| 10 | $512^3$ | $1,34 * 10^8$ | $1,91 * 10^7$ | $1,53 * 10^8$ | 1,14 Gb |
| 11 | $1024^3$ | $1,07 * 10^9$ | $1,53 * 10^8$ | $1,22 * 10^9$ | 9,14 Gb |
| 12 | $2048^3$ | $8,58 * 10^9$ | $1,22 * 10^9$ | $9,81 * 10^9$ | 73,14 Gb |
| 13 | $4096^3$ | $6,87 * 10^{10}$ | $9,81 * 10^9$ | $7,85 * 10^{10}$ | 585,14 Gb |
| 14 | $8192^3$ | $5,49 * 10^{11}$ | $7,85 * 10^{10}$ | $6,28 * 10^{11}$ | 4,57 Tb |

**Tabelle 4:** Entwicklung des Speicherverbrauchs eines Octree im schlimmsten Fall

des Baumes entsprechen dann jeweils einem Uniform Grid mit unterschiedlichen Auflösungen. Der Berechnung wurden acht Byte pro Voxel zugrunde gelegt. Wie Tabelle 3 zeigt, werden diese Zahlen in der Praxis niemals erreicht. So kann man der Tabelle 3 entnehmen, dass das Modell „World“ bei einer Auflösung von $8192^3$ aus $85,5$ Millionen Voxeln besteht, theoretisch wären bei dieser Auflösung im schlimmsten Fall bis zu $549,7$ Milliarden Voxel, also das $6.428$-fache, möglich.

Ein weiterer Vorteil des Octree gegenüber dem Uniform Grid ist es, dass in der Octree-Datenstruktur der LOD gespeichert werden kann. Die Knoten im Baum können benutzt werden um von den Vaterknoten der Blätter bis zur Wurzel immer gröber aufgelöste Varianten des Modells zu speichern. Würde man im Uniform Grid die Detailstufen speichern wollen, entspräche der Speicherverbrauch stets dem in Tabelle 4 aufgelisteten Worst-Case-Szenario des Octree. Die Detailstufen kostet allerdings ein wenig zusätzlichen Speicher. Statt in den Knoten nur einen Zeiger zu speichern, speichert man auch eine Farbe und Normale, macht aus den Knoten der Datenstruktur also Voxel. Da das Verhältnis von Knoten zu Blättern $1 : 7$ beträgt und die Knoten ihren Speicherverbrauch durch den LOD von vier auf acht Byte verdoppeln, kosten die Detailstufen also insgesamt lediglich ein Vierzehntel zusätzlichen Speicher. Dass das Verhältnis von Knoten zu Blättern in einem Octree gegen $1 : 7$ konvergiert ergibt sich daraus, dass es zu je acht Kindern ein Vaterelement geben muss und zu je

acht Vaterelementen wiederum ein Vaterelement usw.

$$\sum_{n=1}^{\infty} \frac{1}{8^n} = \frac{1}{7} \tag{5}$$

Die Anzahl aller Knoten im Baum entspricht bei einem Octree mit unendlicher Tiefe also einem Siebtel der Blätter. Auch bei Bäumen mit endlicher Tiefe und Blättern auf verschiedenen Ebenen entspricht das Verhältnis von Knoten zu Blättern nahezu $1 : 7$. Beispielsweise enthält das Modell „Bunny“ bei einer Auflösung von $2048^3$ 2.183.376 Knoten und 15.283.633 Blätter, was auf sieben Nachkommastellen genau einem Verhältnis von $1 : 7$ entspricht.

Je nach Implementation kann ein Octree uniform oder nicht-uniform unterteilt sein. Bei uniformer Unterteilung wird ein Voxel in acht gleich große Oktanten unterteilt, d.h. die Kinder unterteilen den Raum den der Vaterknoten einnimmt gleichmäßig in einen oberen/unteren Halbraum, linken/rechten Halbraum und einen vorderen/hinteren Halbraum. Bei nicht-uniformer Unterteilung können die drei Trennungsachsen frei gewählt werden.
Für eine nicht-uniforme Unterteilung spricht, dass die homogenen Bereiche des Raumes durch die frei wählbaren Trennungsachsen potentiell besser angenähert werden können. Allerdings müssen die Trennungsachsen dafür explizit in jedem Knoten abgespeichert werden, was der potentiellen Speicherersparnis entgegenwirkt.
Aufgrund der hier aufgeführten Vorteile des Octree wurde er als Datenstruktur für die Voxel-Modelle gewählt. Der hier implementierte Octree ist uniform und achsenparallel. Die Achsenparallelität vereinfacht im Raytracer die Formeln zur Berechnung des Schnittpunkts des Strahls mit den Seitenflächen der Voxel. Ein nicht uniformer Octree stand nie zur Diskussion, da der implementierte 3D-Scan-Konverter ausschließlich uniforme Voxel generieren kann.

An dieser Stelle sei erwähnt, dass id Software für die „id Tech 6 Engine“ ebenfalls mit Voxel-Octrees arbeitet (welche nicht uniform unterteilt sind), bei Ihnen „Sparse Voxel Octrees“ genannt (siehe [Shr08] und [Oli08]). Die Octrees sind „sparse“, weil der Octree im Speicher der Grafikkarte lückenhaft ist und bei Bedarf Teilbäume von der Festplatte nachgeladen werden.

Abschließend sei noch auf ein bedeutsamen Nachteil aller Hierarchien, auch des Octree, gegenüber den Uniform Grid hingewiesen: Wie im vorangegangenen Kapitel 3.4.1 erwähnt, kann im Uniform Grid direkt auf jeden Voxel im Speicher zugegriffen werden. Im Octree muss erst durch den Baum von der Wurzel bis zum Voxel-Blatt abgestiegen werden. In der Praxis hat zwar kein generierter Octree eine größere Tiefe als 14 Ebenen,

dennoch ist es um einiges langsamer bis zu 13 Voxelknoten zu besuchen bevor man auf das gesuchte Voxelblatt zugreifen kann als ein direkter Zugriff. Wie man durch eine Hybridlösung aus Uniform Grid und Octree diesen Nachteil größtenteils wett machen kann zeigt Kapitel 3.5.3

### 3.4.3 kd-Trees

Ein kd-Tree ist ein raumunterteilender Binärbaum der einige Charakteristika mit nicht-uniformen Octrees teilt. In kd-Trees wird pro Knoten eine Ebene definiert die den Raum an beliebige Stelle in zwei Hälften trennt. Da der hier entwickelte 3D-Scan-Konverter aber ausschließlich uniforme Voxel generieren kann, könnten die Vorteile der frei wählbaren Trennungsebenen in diesem Anwendungsfall nicht genutzt werden. Detailstufen können wie im Octree in den Knoten abgelegt werden. Im Octree konvergiert das Verhältnis von Knoten zu Blättern gegen $1 : 7$, da es sich beim kd-Tree um einen Binärbaum handelt konvergiert das Verhältnis hier bei steigender Baumtiefe gegen $1 : 1$. Bei gleicher räumlicher Auflösung ist daher mehr Speicherverbrauch als bei einem Octree zu erwarten. Tatsächlich kann ein Octree mit einem kd-Tree simuliert werden, wobei der kd-Tree dann aber die dreifache Tiefe des Octree hat. Die Unterteilung eines Octree-Knoten in acht Oktanten durch drei Trennungsebenen kann im im kd-Tree durch sieben Knoten simuliert werden, welche die drei Trennungsebenen in drei Ebenen des Binärbaumes nacheinander umsetzten. D.h. um zum gleichen Voxel zu kommen müssen im kd-Tree von der Wurzel aus dreimal mehr Knoten besucht werden als im Octree.
Aufgrund dieser Nachteile kam ein kd-Tree als Datenstruktur nicht in Frage.

### 3.4.4 Bounding Volume Hierarchien

Eine BVH (Bounding Volume Hierarchy) ist eine Datenstruktur, in der Objekte und die Objekte annähernde Volumen ineinander verschachtelt gespeichert werden. Während Octree und kd-Tree raumunterteilende Hierarchien sind, sind BVHs objektunterteilend. Eine BVH kann beispielsweise „bottom-up“ generiert werden, in dem um alle Objekte ein Volumen generiert wird. Benachbarte Volumen werden durch ein größeres Volumen zusammengefasst bis man bei einem Gesamtvolumen angekommen ist. Der Szenengraph von Augenblick ist eine BVH. Diese BVH zur Speicherung der Voxel zu benutzen kam jedoch nicht in Frage, da dies zu viel Speicherplatz gekostet hätte. Ein Bounding Volume kostet 32 Bytes Speicher. Dieser erheblich größere Speicherbedarf gegenüber dem Octree kommt daher, dass für jedes Bounding Volume seine Koordinaten im Raum gespeichert werden müssen. Für uniforme Voxel ist dies hochgradig redundant.

BVHs sind für Polygondaten oder Freiformflächen optimiert und haben keine Chance die besonderen Bedingungen achsenparalleler, uniformer Voxel auszunutzen. So ergibt sich das „Bounding Volume“der Voxel in einem Octree implizit aus ihrer Position und Ausdehnung, es ist nicht notwendig das Bounding Volume in jedem Knoten/Blatt explizit abzuspeichern wie es bei einer BVH der Fall wäre.
Ferner müssen beim Raytracing von BVHs Schnitttests des Strahls mit allen Kinderknoten eines Bounding Volume durchgeführt werden um zu den Blättern der Hierarchie zu gelangen. Der in Kapitel 3.5.1 beschriebene Suchalgorithmus für Octrees hingegen kommt ohne jegliche Schnitttests beim Absteigen durch den Baum aus.
Aufgrund des sehr viel höheren Speicherverbrauchs und der zu erwartenden niedrigeren Geschwindigkeit kam eine BVH für die Voxeldatensätze nicht in Frage.

### 3.4.5 Kodierung des Octree

Nachdem die Entscheidung für einen Octree gefallen ist stellt sich noch die Frage wie der Octree im Arbeitsspeicher möglichst kompakt kodiert werden kann.
In einem naiven Ansatz wird jeder Knoten acht Zeiger auf alle acht Kinder enthalten. Dieser Ansatz bewahrt größtmögliche Flexibilität, die Position jedes Voxels im Speicher ist vollkommen beliebig, denn er kann stets über den im Vaterknoten vermerkten Zeiger aufgefunden werden. Allerdings ist dieser Ansatz auch sehr verschwenderisch. Bei vier Byte pro Zeiger macht dies 32 Byte Speicherverbrauch pro Knoten. Bei vier Byte Nutzdaten pro Voxel macht dies ein Nutzdaten zu Metadaten-Verhältnis von eins zu acht. Auf jedes Byte Nutzdaten kommen acht Byte Metadaten.
Eine Verbesserung der Kodierung kann erreicht werden, indem jeder Knoten nur noch einen einzigen Zeiger, nämlich denjenigen auf das erste Kind, enthält. Wenn garantiert werden kann dass alle acht Kinder direkt nacheinander im Arbeitsspeicher liegen, können so über einen Zeiger und sieben feste Offsets alle acht Kinder adressiert werden. Diese Garantie ist beim Deserialisieren des Voxel-Octree im Loader-Plugin leicht zu bewerkstelligen. Das Loader-Plugin muss dafür lediglich pro Knoten einen 64 Byte großen Speicherblock für die Kinder allokieren.
In diesem Ansatz verbraucht ein Knoten vier Byte für die Datenstruktur (den Zeiger) und vier Byte für die Nutzdaten. Daraus ergibt sich ein Nutzdaten zu Metadaten-Verhältnis von $1:1$. Dabei erhält dieser Ansatz größtenteils die Flexibilität des naiven Ansatzes, die Position der Knoten im Arbeitsspeicher ist immer noch größtenteils frei wählbar, mit der Einschränkung dass Geschwister stets in einem gemeinsamen Speicherblock liegen. Abbildung 21 illustriert den Unterschied zwischen den beiden Ansätzen.

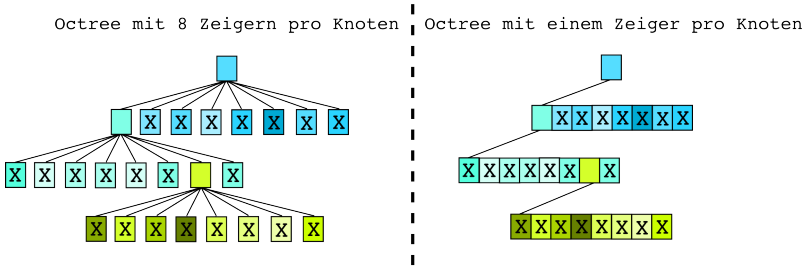


**Abbildung 21:** Naive und optimierte Kodierung des Octree im Vergleich

Damit ist das Ende der Kodiereffizienz aber noch nicht erreicht. Durch das Einführen weiterer Zwänge kann der Octree noch effizienter kodiert werden. Die effizienteste Kodierung ist die im Konverterprogramm entwickelte Preorder-Serialisierung. Wie Kapitel 3.2.2 ausführt, kommt sie mit nur einem Bit an Metainformation pro Knoten aus. Eine noch effizientere Kodierung ist unmöglich. Wenn man Preorder-Serialisierung als Kodierung des Octree zur Laufzeit des Raytracers wählen würde, könnte man sich das Deserialisieren des Baumes aus der Datenstruktur sparen und einfach den Inhalt der „.vox“-Datei in den Arbeitsspeicher kopieren. Allerdings ist diese Kodierung diejenige mit der geringsten Flexibilität, da sie eine feste Reihenfolge aller Voxel des kompletten Baumes im Speicher erfordert. Bei 31 Bit Nutzdaten[37] und ein Bit Metadaten ergibt sich ein hervorragendes Nutzdaten zu Metadaten-Verhältnis von $1 : 0,032$.
Für das Dateiformat hat die Speichereffizienz der Preorder-Serialisierung im Vordergrund gestanden. Gegen die Preorder-Serialisierung als Kodierung des Octree zur Laufzeit des Raytracers sprechen aber mehrere Gründe.
Neben der sehr unflexiblen Speicherung[38] erkauft man sich bei der Preorder-Serialisierung die hohe Kodiereffizienz mit extrem langsamen Zugriffszeiten wenn man wahlfreien Zugriff auf die einzelnen Voxel benötigt, wie es zur Laufzeit des Raytracers permanent der Fall ist. Innerhalb der Preorder-Serialisierung gibt es keinerlei Sprungadressen. Aufgrund der unbekannten Gestalt des Baumes lassen sich auch keine festen Adressen zu bestimmten Voxeln berechnen[39]. Daher muss man, um

[37] Einzelne Bits lassen sich im Speicher nicht ablegen, deshalb 31 Bit Nutzdaten + ein Bit Metadaten = vier Byte im Speicher. Das eine Bit weniger an Nutzdaten wird bei der Farbpalette eingespart, die 15 statt 16 Bit groß ist.

[38] Mehrere Gigabyte an Arbeitsspeicher an einem Stück zu allokieren erweist sich, wie bereits in Kapitel 3.2.2 erwähnt, als sehr problematisch.

[39] Die Berechenbarkeit der Adresse eines jedes Voxels ist ja der große Vorteil des Uniform Grid, wie in Kapitel 3.4.1 ausgeführt.

einen bestimmten Voxel im Baum anzusprechen, den Baum beginnend bei der Wurzel bis zum gesuchten Voxel in Preorder-Reihenfolge rekursiv ablaufen. Diese Form des Zugriffs ist offensichtlich um mehrere Magnituden langsamer als ein Zugriff über maximal 14 Zeiger (von der Wurzel bis zum Blatt). Obendrein schwankt die Zugriffszeit extrem. So kommt das erste Kind des Wurzelknotens direkt nach der Wurzel selbst, der Zugriff darauf geschieht also recht schnell. Das letzte Kind des Wurzelknotens hingegen kommt erst nach sämtlichen Teilbäumen aller anderen Kinder des Wurzelknotens, also müssen zum Ansprechen dieses Kindes mitunter Millionen von Voxeln abgelaufen werden.
Eine Breadth-First-Serialisierung hätte den Vorteil dass die Zugriffszeiten nicht so stark schwanken. Für zwei Knoten einer Ebene wäre sie in etwa gleich da die Knoten der Tiefe nach sortiert sind. Die mittlere Zugriffszeit auf einen beliebigen Voxel wäre jedoch genauso schlecht wie jene der Preorder-Serialisierung.
Die um mehrere Magnituden langsamere Zugriffszeit auf einzelne Voxel steht in keinem Verhältnis zur guten Kodiereffizienz der Preorder-Serialisierung. Daher kam dieser Ansatz nicht als Kodierung des Octree zur Laufzeit in Frage. Dass durch die Verwendung der Preorder-Sortierung das Deserialisieren im Voxel-Loader entfallen könnte spielt im übrigen eine untergeordnete Rolle. Wie in Kapitel 3.2.2 erwähnt, liegt die benötigte Zeit zum Deserialisieren von Bäumen mit Zehnmillionen von Voxeln im Bereich von wenigen Sekunden.

Grundsätzlich kann festgehalten werden, dass jede Steigerung der Kodiereffizienz mit zusätzlichen Operationen zum Zugriff auf einzelne Voxel einhergeht, die Zugriffszeiten zur Laufzeit also verlangsamt. Es stellt sich die Frage, ob es zwischen dem Ansatz mit einem Zeiger pro Knoten und dem Serialisierungs-Ansatz noch einen Mittelweg gibt, der effizienter kodiert ohne die Zugriffszeiten zu sehr zu beeinträchtigen.
Ein möglicher Mittelweg wäre es, die Zeiger von vier Byte auf zwei oder drei Byte zu verkürzen. Auf Betriebssystem-Ebene besteht diese Option nicht. Wenn man aber den Baum auf einen am Stück allokierten Speicherbereich beschränkt, könnte man statt mit absoluten Zeigern mit internen, relativen Sprungadressen arbeiten.
Dies hat zum Einen den Nachteil dass wieder der komplette Baum an einem Stück im Arbeitsspeicher liegen muss und zum Zweiten zur Adressierung eines Voxels mehrere Additionen von Basisadressen und relativen Sprungadressen zu den Voxeln ausgeführt werden müssen.
Da zwei Byte in jeder Hinsicht zu knapp erscheinen um den Speicherbereich eines Baumes mit Millionen von Voxeln abzudecken, soll im Folgenden die Adressierung mit drei Byte großen internen Sprungadressen durchgerechnet werden.
Zunächst könnte man überlegen, für alle Voxel eine drei Byte Sprung-

adresse relativ zu einer Basisadresse, derjenigen der Wurzel, abzulegen. Bedenkt man, dass bei drei Byte eine relative Sprungadresse einen Bereich von maximal $2^{24}$ Bytes ab der Basisadresse adressieren kann, ergibt sich daraus eine maximale Gesamtgröße des Baumes von 16,7 Megabyte. Dies ist für die hier vorkommenden Mengen an Voxeln nicht ausreichend. Bedenkt man weiterhin, dass alle Voxel eine feste Größe (Drei Byte Sprungadresse, vier Byte Nutzdaten) hätten, könnte man mit der Sprungadresse statt Bytes Voxel zählen. Somit ließen sich mit einer drei Byte Sprungadresse bis zu 16,7 Millionen Voxel adressieren. Auch dies ist für die hier vorkommenden Voxelmengen nicht ausreichend. Geht man noch einen Schritt weiter und interpretiert alle Sprungadressen nicht relativ zur Basisadresse der Wurzel sondern relativ zur bereits ausgerechneten Adresse des aktuellen Voxels, so erreicht man eine Limitierung auf 16,7 Millionen Voxel für jeden beliebigen Teilbaum des Octree. Man könnte meinen, dass maximal 16,7 Millionen Voxel für jeden beliebigen Teilbaum ausreichend groß ist um die hier vorkommenden Octrees zu kodieren. Die Limitierung auf maximal 16,7 Millionen Voxel gilt aber insbesondere auch für die Teilbäume direkt unterhalb der Wurzel. Die Teilbäume der ersten sieben Kinder der Wurzel dürften den Umfang von $16.777.216 + 8 - 1$ Voxeln nicht überschreiten, damit der achte Kinderknoten der Wurzel noch seinen eigenen Teilbaum adressieren kann. Somit wird durch diese Variante der maximal mögliche Umfang des Baumes nur unwesentlich (um sieben Voxel) gesteigert.
Da im Rahmen dieser Diplomarbeit Modelle mit bis zu 300 Millionen Voxeln entstanden sind ist klar, dass der Ansatz mit drei Byte Sprungadressen nicht für jedes Modell funktionieren kann. Gerade die Modelle mit den meisten Voxeln sind es aber, für die eine weitere Komprimierung der Datenstruktur relevant wäre.

Alles in allem wurde mit dem 4+4-Byte-Ansatz eine Lösung mit einem akzeptablen Nutzdaten/Metadatenverhältnis implementiert. Sie stellt, entgegen den anderen Ansätzen, keine schwer zu garantierenden Anforderungen an die Speicherung, unterstützt Bäume mit einer Größe von bis zu vier Gigabyte und bietet schnelle Zugriffszeiten bei wahlfreiem Zugriff auf einzelne Voxel[40]. Zu Gunsten der Geschwindigkeit des Raytracers kam die sehr effizient kodierte Preorder-Serialisierung für den Baum zur Laufzeit nicht in Frage. Die Reduktion der Zeiger von vier auf drei Byte schränkt den Umfang des Baumes zu stark ein um praktisch anwendbar zu sein.
Kapitel 4.3.2 gibt weitere Gedankenanstöße, wie der Speicherverbrauch der Datenstruktur trotz vier Byte großer Zeiger in den Knoten noch erheblich

[40]Die Zugriffszeiten steigen bei diesem Ansatz linear mit der Distanz des Voxels zur Wurzel. Um einen Voxel auf Ebene fünf zu erreichen müssen vorher vier weitere Voxel (die Vorfahren) besucht und der jeweils nächste Zeiger ausgelesen werden.

reduziert werden kann.

### 3.4.6 Die einzelnen Voxel

Bereits mehrfach wurde erwähnt, dass die Voxel in dieser Implementation acht Byte groß sind. Im vorangegangenen Kapitel 3.4.5 wurde bereits im Detail auf den vier Byte großen Zeiger eingegangen den jeder Voxel enthält. Im Folgenden wird auf die vier Byte Nutzdaten eines jeden Voxels eingegangen.
Die Nutzdaten des Voxels setzen sich aus zwei Byte für den Farbpalettenindex (zur Motivation dahinter siehe Kapitel 3.1.2) und zwei Byte für die Normale des Voxels zusammen. Von den zwei Byte für den Farbpalettenindex muss ein Bit leider ungenutzt bleiben, da die Farbpalette im „.vox“-Dateiformat nur einen Umfang von 15 Bit hat.
Letztlich geht es bei der Wahl des Nutzdatenumfangs pro Voxel um die Frage, wie man bei einem Speicher von beschränkter Größe die Darstellungsqualität maximieren kann. Hierbei gilt es, eine Abwägung zwischen räumlicher Auflösung des Modells und Informationsgehalt pro Voxel zu treffen. Diese Abwägung entspricht der Abwägung zwischen räumlicher Auflösung und Farbtiefe in zweidimensionalen Bildern. Je mehr Information ein einzelner Voxel enthält (mehr Bits pro Voxel), umso weniger Voxel kann das Modell insgesamt enthalten.
Einerseits möchte man die räumliche Auflösung des Modells maximieren. Durch eine höhere räumliche Auflösung werden die negativen Effekte der Diskretisierung minimiert. D.h. Treppeneffekte an den Rändern der Modelle und Darstellungsfehler wie die Selbstverschattung treten seltener auf, da die Form des Modells präziser angenähert wird. Andererseits möchte man möglichst viel Information in jedem einzelnen Voxel speichern. Mehr Bits pro Voxel können genutzt werden um die Farbtiefe zu erhöhen, die Normalen feiner abzustufen oder dem Voxel weitere Eigenschaften wie Reflexionsgrad, Transparenz etc. zu verleihen. Legt man die Repräsentation der Modelle zu stark in eine der beiden Richtungen aus, erreicht man unterm Strich eine suboptimale Darstellungsqualität. Die Frage ist also, wo der beste Kompromiss aus räumlicher Auflösung und Informationsgehalt pro Voxel liegt.
Darstellungsqualität ist ein schwer messbarer Begriff. Zum Einen ist das Empfinden der Darstellungsqualität subjektiv und kann von Betrachter zu Betrachter schwanken. Zum Anderen hängt die erzielte Darstellungsqualität auch vom Anwendungsfall und dem Kontext des Modells ab. Legt man in einer Anwendung beispielsweise großen Wert auf den Realismus der Beleuchtung, würden nicht vorhandene Reflexionseigenschaften der Voxel verstärkt als qualitätsmindernd empfunden.
Da der Fokus dieser Diplomarbeit auf der Implementation und Optimierung des Voxel-Raytracings hinsichtlich Geschwindigkeit und

Speicherverbrauch liegt, wurde die erzielte *absolute* Darstellungsqualität im Vergleich zum Speicherverbrauch der Modelle hinten an gestellt und versucht, im Rahmen dessen was mit dem limitierten, zur Verfügung stehenden Speicher möglich ist eine gute Balance aus räumlicher Auflösung und Informationsgehalt pro Voxel zu finden. Im Laufe der Entwicklung zeigt sich, dass die Modelle zunächst zu stark auf die räumliche Auflösung ausgelegt waren. Der Informationsgehalt wurde dann sukzessive auf vier Byte pro Voxel gesteigert.
So war zu Beginn der Implementation der Farbpalettenindex pro Voxel auf sieben Bit (128 Einträge) beschränkt. Es konnten in einem Modell also nur 128 verschieden Farben benutzt werden. Mit der Vergrößerung des Index auf 15 Bit konnte dieses Limit auf 32.768 verschiedene Farben gesteigert werden. Der Speicherverbrauch eines Voxels stieg dabei von fünf Byte (vier Byte Zeiger plus ein Byte Palettenindex) auf sechs Byte, was bei konstanter Obergrenze für den Speicherverbrauch einer Reduktion der möglichen räumlichen Auflösung von 16% entspricht. In der Folge wurde der Speicherverbrauch pro Voxel durch das Hinzufügen einer Normalen nochmals um zwei Byte auf insgesamt acht Byte gesteigert. Vorher wurden die Normalen zur Laufzeit berechnet und entsprachen den Seitenflächen der Voxel (siehe Abbildung 22). Die Voxelstruktur der Modelle ist durch die

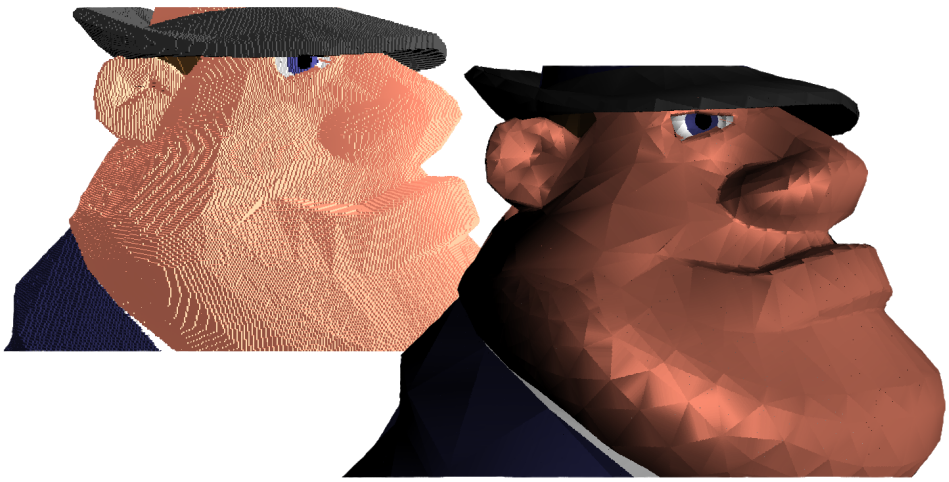

**Abbildung 22:** Gegenüberstellung von zur Laufzeit generierten Normalen auf den Seitenflächen der Voxel und den aus Eckpunktnormalen interpolierten Normalen

senkrecht auf den Seitenflächen der Voxel stehenden Normalen deutlich erkennbar. Die individuellen Normalen pro Voxel hingegen ermöglichen eine gleichmäßigere, der Form des Modells angenäherte Beleuchtung, was

die Kantigkeit der Modelle stark reduziert[41]. Durch die Normalen wurde die erzielbare maximale räumliche Auflösung um weitere $25\%$ reduziert.

Kapitel 3.1.2 erläutert bereits ausführlich welche Gründe für den Einsatz einer Farbpalette im Vergleich zum RGB-Farbraum sprechen. Zusammengefasst kann gesagt werden, dass eine Farbpalette den verfügbaren Speicher besser ausnutzt als der 24-Bit RGB-Farbraum. In der Palette werden alle verfügbaren Kombinationen der Bits ausgereizt, während es selten vorkommen würde dass in einem Modell sämtliche 16,7 Millionen Kombinationen des RGB-Farbraums ausgenutzt werden. Die Entropie (der mittlere Informationsgehalt eines Bits) ist in der Farbpalette also höher.
Eine ähnliche Überlegung führte zur Kodierung der Normalen. Wie in Kapitel 3.1.3 bereits erwähnt, werden die Normalen in Form von Kugelkoordinaten gespeichert, dabei werden die Winkel $\theta$ und $\phi$ mit jeweils acht Bit kodiert. Üblicherweise werden Normalen in kartesischen Koordinaten in einfacher Gleitkommagenauigkeit (3*32 Bit=96 Bit) gespeichert. Mit dieser Kodierung können beliebige Vektoren im dreidimensionalen Raum beschrieben werden. Normalen sind aber in der Regel auf die Länge eins normiert, das heißt es ist überflüssig die Länge der Normalen abzuspeichern. Entscheidend ist die Orientierung der Normalen. Kugelkoordinaten liefern genau dies: Mithilfe zweier Winkel wird die Orientierung eines Vektors im Raum beschrieben, bzw. ein Punkt auf der Einheitskugel (siehe Abbildung 14 auf Seite 36). Beide Winkel werden in dieser Implementation auf 256 diskrete Werte (acht Bit) beschränkt. Insgesamt können mit den 16-Bit Kugelkoordinaten also 65.536 verschiedene Normalen kodiert werden, welche gleichmäßig auf der Einheitskugel verteilt sind. Ähnlich wie bei den Farben ergibt sich im Vergleich zu den kartesischen Koordinaten eine bessere Ausnutzung der zur Verfügung stehenden Bitkombinationen. Im Vergleich zu zwölf Byte für kartesische Normalen ergibt sich bei zwei Byte für Kugelkoordinaten eine leicht verlustbehaftete Kompression auf ein Sechstel.

Alles in allem kann geschlossen werden dass die gewählte Repräsentation der Modelle immer noch mehr in Richtung räumliche Auflösung denn Informationsgehalt pro Voxel tendiert. Dies liegt zum Einen daran, dass so besser getestet werden konnte wie viele Voxel sich auf der CPU echtzeitfähig strahlverfolgen lassen, zum Anderen hätten komplexere Materialeigenschaften der Voxel auch zusätzlichen Aufwand im Raytracer bedeutet.
Zum Vergleich sei auf die Voxelgröße bei id Software verwiesen. Jon Olick hat in [Oli08] aufgeführt, dass bei ihnen ein Voxel zur Laufzeit aus 52 Byte

[41] Mit den zur Laufzeit generierten Normalen entsteht ein „LEGO-Look“, welcher durchaus einen gewissen Reiz besitzt, dem Realismus aber nicht dienlich ist.

besteht. Dabei entfallen 45 Bytes auf die Octree-Datenstruktur, nämlich 32 Byte für Offsets zu den Kindern, 12 Bytes für die Trennungsebenen des nicht uniformen Octree und ein Byte für ein Padding. Die restlichen sieben Bytes entfallen auf die Nutzdaten, drei Byte für den diffusen RGB-Term, drei Byte für die Normale[42] und ein Byte für einen Spiegelungswert. Daraus ergibt sich ein Nutzdaten-/Metadatenverhältnis von $1 : 6,42$, was erheblich schlechter ist als das hier erzielte Verhältnis von $1 : 1$. Im Dateiformat von id Software verbrauchen die Voxel lediglich acht Byte (hier: vier Byte), die sieben Byte Nutzdaten plus ein Byte mit einer „children bit mask“[43]. Für dieses Format wird von id Software ein Nutzdaten-/Metadatenverhältnis von $1 : 0,142$ erreicht (hier: $1 : 0,032$).
Würden in dieser Implementation pro Voxel mehr Materialeigenschaften gespeichert, würde sich übrigens auch das Nutzdaten-/Metadatenverhältnis von $1 : 1$ noch weiter verbessern. Kapitel 4.3.3 zeigt Wege auf, wie die Darstellungsqualität druch zusätzliche Materialeigenschaften weiter gesteigert werden könnte.

[42] Dass für Farbe und Normale keine kompaktere Form benutzt wird, wie in dieser Implementation, ist vermutlich den geforderten Bildwiederholraten zuzuschreiben, da so die Umwandlungsschritte entfallen.

[43] Dass auch für das Dateiformat keine effizientere Kodierung gewählt wurde ist vermutlich der geforderten Geschwindigkeit des Datentransfers von Festplatte zur Grafikkarte geschuldet.

## 3.5 Der Voxel-Raytracer

Nachdem die Struktur der Volumendaten im Arbeitsspeicher geklärt ist kann auf die Details des Kernstücks dieser Diplomarbeit - den Raytracer - eingegangen werden.

### 3.5.1 Das Konzept

Das grundlegende Konzept des Raytracers ist es, nacheinander Punkte entlang eines Sehstrahls zu berechnen und für jeden Punkt heraus zu finden innerhalb von welchem Voxel-Blatt des Octree er liegt. Er bricht dann die Suche bei dem ersten gefüllten (d.h. nicht transparenten) Blatt ab und liefert die Farbe des Blattes und dem Lambda-Wert zu dem Punkt an Augenblick zurück.
Der allererste entlang des Strahls berechnete Punkt ist der Eintrittspunkt des Strahls in das Voxel-Geometrieobjekt. Dass der Strahl das Geometrieobjekt schneidet ist garantiert, denn Augenblick ruft den Raytracer für die Voxel-Geometrie überhaupt nur dann auf wenn der fragliche Strahl das Voxel-Geometrieobjekt durchquert.
Nun fragt sich, wie das Blatt im Octree bestimmt wird innerhalb dessen der Eintrittspunkt liegt (oder besser gesagt: auf dessen Außenfläche er liegt). Dies geschieht durch einen Suchalgorithmus welcher in einer Schleife implementiert ist, die, beginnend bei der Wurzel, entscheidet in welchem der acht Oktanten des Raumes den der Voxel einnimmt der untersuchte Punkt liegt. Je nach Oktant steigt die Schleife dann in das dazugehörige Kind des Knotens ab und wiederholt den Prozess, bis ein Kind erreicht wird das ein Blatt ist, d.h. selber keine Kinder mehr hat. Somit ist dasjenige Blatt gefunden, innerhalb dessen der auf dem Strahl generierte Punkt liegt. Von diesem Kind muss nun nur noch abgefragt werden, ob es ein gefüllter Voxel oder ein transparenter Voxel ist. Sollte es sich um einen gefüllten Voxel handeln ist der Algorithmus an seinem Ende angelangt. Sollte es sich um um einen transparenten Voxel handeln, muss ein neuer Punkt entlang des Sehstrahls bestimmt werden. Die optimale Position für den neuen Punkt ist der Austrittspunkt des Sehstrahls aus dem gerade gefundenen Blatt. Denn der Austrittspunkt aus dem just gefundenen Blatt ist der Eintrittspunkt in das nächsten Blatt den der Strahl durchquert. Um den Austrittspunkt aus dem aktuell gefundenen Voxel zu bestimmen, müssen Schnitttests des Strahls mit den Seitenflächen des aktuellen Voxels durchgeführt werden. Von den berechneten Schnittpunkten ist dann noch derjenige zu bestimmen welcher dem untersuchten Punkt am nächsten gelegen ist, denn dieser ist der Austrittspunkt des Sehstrahls aus dem gerade gefundenen Voxel.
Abbildung 23 illustriert das grundlegende Konzept des Raytracers. Die vom Algorithmus berechneten Schnittpunkte sind von eins bis sechs

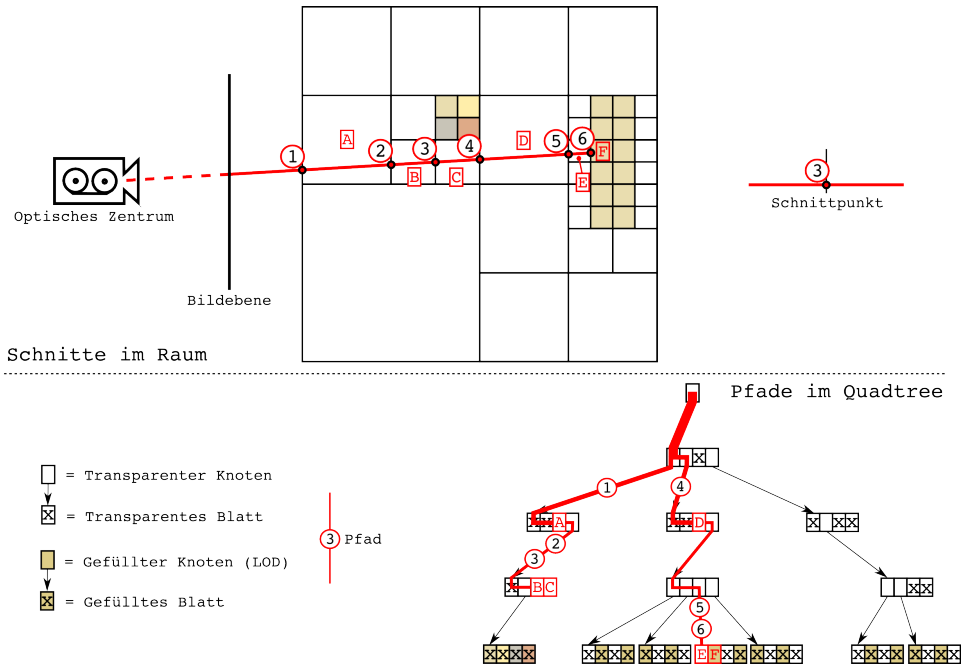


**Abbildung 23:** Die Vorgehensweise des Raytracers beim Durchqueren eines Quadtree

durchnummeriert, die gefundenen Blätter des Baumes mit den Buchstaben „A“ bis „F“ gekennzeichnet. Rot markiert sind die Pfade entlang derer der Suchalgorithmus den Baum traversiert. Je dicker die Linie, umso öfter wird entlang dieses Pfades traversiert. Blatt „F“ ist von den gefundenen Blättern das Erste welches nicht transparent ist. Damit kann die Strahlverfolgung abgebrochen werden. In nebenstehendem Baum sind die vom Algorithmus gelaufenen Wege von der Wurzel zu den Blättern eingezeichnet. Der Grafik ist zu entnehmen, dass die Anzahl der berechneten Punkte auf dem Strahl identisch ist mit der Anzahl der Blätter die der Strahl durchquert, d.h. der Algorithmus berechnet die minimal notwendige, optimale Menge an Punkten. Jedes durchquerte Blatt wird genau einmal untersucht und kein Blatt welches der Sehstrahl durchquert wird übersprungen. Daher schwanken auch die Abstände in denen der Strahl abgetastet wird. Die Abtastung passt sich der lokalen räumlichen Auflösung des Octree an. Große, gleichförmige Bereiche des Raums werden somit effizient übersprungen.

Abbildung 24 illustriert warum von den Schnittpunkten mit den Seitenflächen (grüne und blaue Linien) des Voxels derjenige Schnittpunkt mit dem geringsten Abstand zum aktuellen Punkt der Austrittspunkt

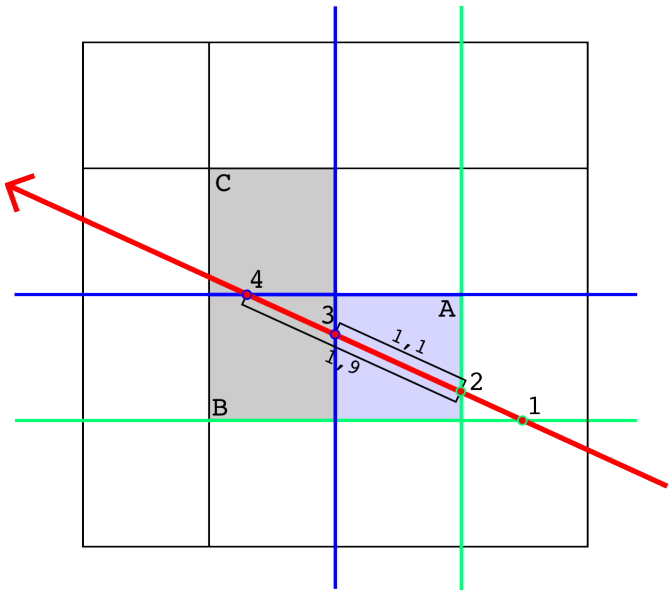


**Abbildung 24:** Die Schnittpunkte eines Strahls mit den Seitenflächen eines Voxels

aus dem gerade untersuchten Voxel ist. Der blau unterlegte Voxel A ist der derzeit untersuchte Voxel, Schnittpunkt 2 ist aktuelle Punkt (der Eintrittspunkt in den Voxel A) und kommt nicht als neuer Schnittpunkt in Betracht. Schnittpunkt 1 kommt ebenfalls nicht in Betracht und wird auch gar nicht berechnet. Die Schnittpunkte 3 und 4 kommen als potentielle Austrittspunkte aus Voxel A in Frage. Schnittpunkt 3 hat von beiden den geringeren Abstand zum optischen Zentrum und ist daher die korrekte Wahl. Schnittpunkt 4 darf nicht als nächster Punkt ausgewählt werden, denn dann würde Voxel B vom Algorithmus übersprungen werden.

Ein Problem mit den bisher berechneten Schnittpunkten ist, dass sie sowohl Austrittspunkt aus dem derzeit untersuchten, als auch Eintrittspunkt in den nächsten Voxel sind. Daher wird von dem Algorithmus ein kleiner Versatz auf die Austrittspunkte addiert, damit aus den Schnittpunkten Punkte werden die eindeutig innerhalb des nächsten Voxels liegen. Abbildung 25 illustriert die Notwendigkeit der Versatzpunkte. Jeder der Versatzpunkte liegt eindeutig innerhalb eines der Voxel die der Strahl auf seinem Weg durch den Raum durchquert. Wie in Kapitel 3.4.1 erwähnt, kann im Uniform Grid ein 3D-Bresenham-Algorithmus verwendet werden um zu bestimmen welche Voxel der Strahl durchquert. Im Octree ist dies nicht möglich, da der Bresenham-Algorithmus nur auf Voxeln von einheitlicher Größe funktioniert.

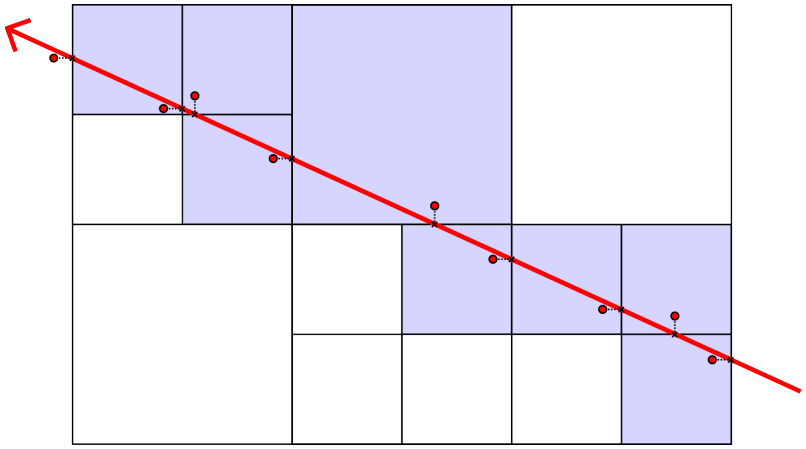

**Abbildung 25:** Schnitt- und Versatzpunkte eines Strahls.

Listing 5 auf Seite 82 zeigt die Umsetzung des beschriebenen Algorithmus in Pseudocode. Abgesehen von der anfänglichen Berechnung des Eintrittspunktes in das Geometrieobjekt wird der Algorithmus mit zwei geschachtelten Schleifen umgesetzt. Die äußere Schleife berechnet die Punkte entlang des Strahls, die innere Schleife führt den Suchalgorithmus, also den Abstieg durch den Baum bis zu dem Blatt in dem der Punkt liegt, durch. Der Code der inneren Schleife ist daher der Programmteil welcher mit Abstand am häufigsten ausgeführt wird. Diesen Suchalgorithmus effizient zu implementieren, bzw. die Zahl der inneren Schleifendurchläufe zu reduzieren, hat die größten Auswirkungen auf die Geschwindigkeit des Raytracers.
Für das Raytracing von Polygonen gilt, dass die Schnitttests in der Regel das Teuerste am ganzen Verfahren sind. Beim Raytracing von Voxeln, so wie es hier implementiert wurde, ist die Suche nach dem passenden Blatt im Octree das Teuerste. Schnittpunktberechnungen müssen relativ wenige durchgeführt werden und diese sich auch noch sehr einfach, da die Seitenflächen der Voxel achsenparallel sind. Die Formel zur Berechnung des Schnittes eine Geraden mit einer Ebene verkürzt sich bei achsenparallelen Seitenflächen auf eine Subtraktion und eine Multiplikation. Denn aus der Ebenengleichung in Parameterform:

$$Ax + By + Cz + D = 0 \tag{6}$$

und der Geradengleichung in der Form:

$$\vec{x} = \begin{pmatrix} x_o \\ y_o \\ z_o \end{pmatrix} + \lambda \cdot \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} \tag{7}$$

folgt durch Einsetzen, dass

$$\lambda = \frac{-(Ax_o + By_o + Cz_o + D)}{Ax_d + By_d + Cz_d} \tag{8}$$

ist. Da die Seitenflächen achsenparallel sind, sind die Normalen welche senkrecht auf den Seitenflächen stehen $(1,0,0)^T$, $(0,1,0)^T$ oder $(0,0,1)^T$. Damit sind immer zwei der Summanden der Ebenengleichung gleich Null. Die Formel zur Berechnung von Lambda reduziert sich damit auf:

$$\lambda = \frac{-D - x_o}{x_d} \tag{9}$$

für Seitenflächen die parallel zur YZ-Ebene liegen,

$$\lambda = \frac{-D - y_o}{y_d} \tag{10}$$

für Seitenflächen die parallel zur XZ-Ebene liegen und

$$\lambda = \frac{-D - z_o}{z_d} \tag{11}$$

für Seitenflächen die parallel zur XY-Ebene liegen.

$D$ ist dabei die Inverse der Verschiebung $V$ der Ebene entlang der Achse auf der die Ebene senkrecht steht. Statt $-D$ kann also $V$ eingesetzt werden. Statt durch die Komponenten des Richtungsvektors zu teilen bietet es sich an, einmalig die Inverse des Richtungsvektors zu bilden und dann in den Schnittpunktberechungen mit den Inversen zu multiplizieren, da eine Multiplikation schneller ausgeführt werden kann als eine Division.

In Zeile 18 des Pseudocode-Listings befindet sich das Programmende. Dieses tritt ein wenn ein gefülltes Blatt gefunden wurde[44]. In Zeile 20 des Listings befindet sich die Abbruchbedingung des Suchalgorithmus. Die Abbruchbedingung tritt ein wenn ein Blatt gefunden wurde. In Zeile 36 befindet sich die Abbruchbedingung der äußeren Schleife, sie ist gegeben wenn der nächste berechnete Punkt außerhalb des Geometrieobjekts liegt. Dieser Fall liegt dann vor, wenn ein Strahl auf seinem kompletten Weg durch das Volumen kein gefülltes Blatt getroffen hat. Auf die in den Zeilen 24-27 gegebene Logik der Berechnung des Index des Oktanten in den abgestiegen werden soll geht Kapitel 3.5.3 näher ein.

[44]Wenn von gefüllten und transparenten Voxeln die Rede ist, stellt sich die Frage wie ein Voxel überhaupt transparent sein kann wenn alle Voxel einen Farbpalettenindex besitzen. Tatsächlich ist für transparente Voxel der Farbpalettenindex 0 reserviert. Daraus ergibt sich, dass der Umfang der Farbpalette in Wirklichkeit nicht $2^{15}$ Farben, sondern nur $2^{15} - 1$ Farben beträgt.

Was aus dem Pseudocode-Listing u.a. nicht hervorgeht ist, dass für jeden Schnitt mit einem gefüllten Voxel neben des Abstands zum optischen Zentrum noch die Farbe und die Normale an Augenblick zurück gegeben werden muss. Da die Farbe in Form eines Palettenindex vorliegt, muss der RGB-Wert erst in der Palette nachgeschlagen werden, die das Loader-Plugin aus dem Header der „.vox"-Datei ausgelesen hat. Dieses Nachschlagen, ein einfacher Array-Zugriff, kostet ein paar zusätzliche Prozessorzyklen und ist ein notwendiger Tribut an die Speicherplatzersparnis die durch das Verwenden der Farbpalette erzielt wurde. Die Normale liegt auch nicht in der Form vor wie sie Augenblick erwartet, sie muss von Kugelkoordinaten in kartesische Koordinaten umgerechnet werden. Die Kugelkoordinaten ($\theta$, $\phi$) können in die kartesischen Koordinaten $(x, y, z)^T$ mit der Formel:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} sin\theta \cdot cos\phi \\ sin\theta \cdot sin\phi \\ cos\theta \end{pmatrix} \tag{12}$$

umgerechnet werden. Listing 2 zeigt, wie diese Umrechnung effizient mit

```
1  theta = (theta*_1_256th)*PI;
2  phi = (phi*_1_128th)*PI;
3
4  __m128 sinTheta, cosTheta, sinPhi, cosPhi;
5  sincos_ps(theta.getVector(),&sinTheta,&cosTheta);
6  sincos_ps(phi.getVector(),&sinPhi,&cosPhi);
7
8  VecPack3 normals;
9  normals[0] = _mm_mul_ps(sinTheta,cosPhi);
10 normals[1] = _mm_mul_ps(sinTheta,sinPhi);
11 normals[2] = cosTheta;
```

**Listing 2:** Umwandlung der Normalen von Kugelkoordinaten in kartesische Koordinaten

der SIMD-Einheit der x86-Prozessoren umgesetzt werden kann. In dem Listing werden vier Normalen parallel mit SSE-Befehlen in kartesische Koordinaten umgewandelt. In den ersten beiden Zeilen werden die diskreten Winkel aus dem Wertebereich [0, 255] bei $\theta$ und [-128,128] bei $\phi$ in Bogenmaß umgerechnet. In Zeile vier werden vier Variablen angelegt, welche die Sinus- und Kosinus-Werte der Winkel speichern werden. In den Zeilen fünf und sechs werden diese dann von der Methode sincos_ps() ausgerechnet. sincos_ps() ist eine Hilfsfunktion aus dem Augenblick-SDK welche die Sinus- und Kosinusberechnung effizient mit SEE-Befehlen implementiert. In den Zeilen neun bis elf werden dann aus den Sinus- und Kosinustermen mit dem Compiler Intrinsic _mm_ml_ps(), welcher eine SIMD-Multiplikation durchführt, die kartesischen Koordinaten ermittelt.

Trotz dieser effizienten Implementation kostet diese Umwandlung einige Prozessorzyklen. Auch dies ist ein notwendiger Tribut an die effiziente Kodierung der Voxel. Wie bei der Farbpalette galt es hierbei, zwischen Speicherverbrauch der Daten und Geschwindigkeit der Ausführung abzuwägen. Die Kugelkoordinaten reduzieren im Vergleich zu den kartesischen Koordinaten den Speicherverbrauch auf ein Sechstel (zwei Byte statt zwölf Byte pro Voxel), während die Geschwindigkeit des Raytracers durch den Code in Listing 2 nur um etwa ein $\frac{1}{25}$-stel reduziert wird.

Auch die Seitenflächen der Voxel mit denen der Raytracer die Schnittpunkte berechnet liegen nicht in der Datenstruktur vor, da dies hochgradig redundant wäre und viel Speicherplatz pro Voxel kosten würde[45]. Die Seitenflächen werden daher zur Laufzeit in der inneren Schleife bestimmt. Die Seitenflächen des Wurzelelements sind bekannt, sie entsprechen der Bounding Box des Voxel-Geometrieobjekts. Bei jedem Abstieg in einen Kinderknoten werden dann von der inneren Schleife die Seitenflächen des Kindes aus den Seitenflächen des Vaterelements berechnet.

Da die verwendeten Voxel keinen spiegelnden Term enthalten, besteht für den Algorithmus keine Notwendigkeit vom Auftreffpunkt des Strahls aus Sekundärstrahlen in die Geometrie zu versenden. Das Gesamtsystem ist dennoch als Raytracer und nicht als „Raycaster“[46] zu klassifizieren, da der Algorithmus optional auch zur Versendung von Schattenfühlern in der Verschattungsberechnung (siehe Abbildung 3 auf Seite 15 für eine schematische Zeichnung von Primär- und Sekundärstrahlen) eingesetzt wird.

### 3.5.2 Vorteile aus der Datenstruktur

Aus der Verwendung eines Octree zur Strukturierung der räumlichen Daten entstehen dem Raytracer einige Vorteile.

- Der Strahl kann vom Raytracer dank des Octree adaptiv abgetastet werden. Wenn der Strahl über große Distanzen einen transparenten Raum durchquert entsteht auch keinerlei Mehraufwand. Bei einer Abtastung des Strahls in konstanten Intervallen würden viele redundante Berechnungen durchgeführt.
- Die Abbruchbedingung des Raytracers ist sehr einfach. Da der Octree raumunterteilend ist gibt es die Sicherheit dass das erste gefun-

[45] Die Redundanz steckt darin, dass sich viele Voxel Seitenflächen teilen. Würde jede Seitenfläche in einer zwei Byte großen Ganzzahl gespeichert, ergäbe dies $2 * 6 = 12$ Byte pro Voxel an zusätzlichem Speicherverbrauch.

[46] Die Definition eines Raycasters schwankt von Kontext zu Kontext. Generell sind damit Programme gemeint die ausschließlich Primärstrahlen durch den Raum verfolgen.

dene, gefüllte Blatt garantiert aus Sicht der Kamera der vorderste, also sichtbare, Voxel ist. Es sind keine weiteren Schnitte mit anderen Voxeln notwendig um das Sichtbarkeitsproblem zu lösen. Daraus folgt auch, dass die Geschwindigkeit des Raytracers mit steigender Datenmenge nur logarithmisch abnimmt (siehe Kapitel 4.1.1).

- Da die Voxel im Octree allesamt achsenparallel sind, vereinfacht sich die Berechnung der Schnittpunkte von Strahlen und den Seitenflächen.

- Die im Octree enthaltenen Detailstufen ermöglichen eine Beschleunigung der Darstellung. Durch die Wahl größerer Detailstufen wird die räumliche Auflösung reduziert und somit der Aufwand des Raytracings reduziert. Die Zahl der notwendigen inneren Schleifen sinkt exponentiell bei reduzierter Auflösung. Denn zum Einen wird durch die gröbere räumliche Auflösung die Zahl der notwendigen Abtastung des Strahls, d.h. äußere Schleifendurchläufe, reduziert. Zum Anderen reduziert sich durch die Wahl einer gröberen Detailstufe auch die Zahl der pro berechnetem Punkt notwendigen inneren Schleifendurchläufe, da der Suchalgorithmus auf höheren Ebenen des Baumes beendet wird. Der Suchalgorithmus ist also pro berechnetem Punkt schneller beendet und er muss insgesamt für weniger Punkte entlang des Strahls ausgeführt werden.

### 3.5.3 Optimierungen

Im Laufe der Entwicklung des Raytracing-Algorithmus wurden eine Vielzahl größerer und kleinerer Verbesserungen vorgenommen, welche die Geschwindigkeit des Verfahrens steigern.
So wird im Begleittext zu Abbildung 24 behauptet, dass die Schnittpunkte 1 und 2 als Austrittspunkte aus dem Voxel nicht in Frage kommen und gar nicht berechnet werden. Dies ist intuitiv verständlich, da Schnittpunkt 1 und 2 aus Sicht des optischen Zentrums *vor* dem aktuellen Voxel und nicht *hinter* ihm liegen. Die Berechnung der Schnittpunkte 1 und 2 kann also ausgelassen werden. Dies gilt äquivalent für den dreidimensionalen Raum, in welchem jeder Voxel sechs Seitenflächen hat. Von diesen müssen nur mit dreien die Schnittpunkte berechnet werden. Mit welchen der drei Seitenflächen die Schnittpunkte berechnet werden müssen hängt von der Orientierung des Strahls ab. Vereinfacht ausgedrückt sind es die dem Betrachter abgewandten Seiten (in Abbildung 24 blau markiert), mit denen geschnitten werden muss. Für einen Strahl entlang der positiven X- und Y-Achse wären es beispielsweise die rechte und obere Seitenfläche mit denen geschnitten werden muss. Da die Orientierung des Strahls konstant ist, muss nur einmalig zu Beginn des Algorithmus entschieden werden welche Seiten dem Strahl abgewandt sind. Die in der äußeren Schleife

notwendigen Schnittpunktberechnungen können so von sechs auf drei halbiert werden.

Die größte Maßnahme zur Beschleunigung des Raytracers war die Umstellung der Implementation auf vierfache Parallelverarbeitung durch die SSE-Einheit der CPU. Der Algorithmus wurde zunächst für einzelne Strahlen entwickelt und nachdem dies zuverlässig funktionierte auf SSE umgestellt. Theoretisch wäre bei einhundertprozentiger Kohärenz eine Beschleunigung um den Faktor vier im Vergleich zur Einzelstrahl-Implementation möglich. In der Praxis ist die Kohärenz aber nicht immer vollkommen, da es immer wieder vorkommen wird, dass der Algorithmus für einzelne Strahlen schon fertig wäre, für andere Strahlen des Bündels aber noch weiter ausgeführt werden muss. Dies reduziert die erzielbare Beschleunigung gegenüber einer Einzelstrahl-Implementation. Ferner wird der Algorithmus durch die Verwendung von SSE-Befehlen auch ein Stück weit aufwändiger, da statt eines Strahles vier Strahlen verwaltet werden müssen. Insbesondere beim Zugriff auf die Octree-Datenstruktur sind teure, horizontale Zugriffe auf die SIMD-Vektoren notwendig. Eine horizontale Operation liegt dann vor, wenn die vier Einzelwerte eines SIMD-Vektors verrechnet werden müssen. Unterm Strich ist bei der Umstellung auf SIMD eine Beschleunigung um den Faktor zwei bis drei realistisch. Die tatsächlich durch die Umstellung auf SSE erzielte Beschleunigung beträgt ziemlich genau Faktor $3,0$. Tabelle 8 auf Seite 86 führt die erzielten Geschwindigkeiten der Einzelstrahl- und der SIMD-Implementation auf.
Dank der in Kapitel 2.5 erwähnten Klassen Float4, Int4 und Bool4 von Augenblick, welche die SSE Compiler Intrinsics kapseln, war die Umstellung des Programmcodes auf vierfache Parallelverarbeitung gut zu bewältigen und die Lesbarkeit des Programmcodes hat im Vergleich zur Einzelstrahl-Implementation kaum gelitten.

Ein signifikanter Unterschied zwischen der Einzelstrahl- und der SIMD-Implementation ist allerdings der Umgang mit Verzweigungen und Masken. Generell will man heutzutage in hochoptimiertem Programmcode aufgrund der Länge der Befehls-Pipeline moderner CPUs Verzweigungen nach Möglichkeit vermeiden. Verzweigungen sind problematisch weil sie so genannte „Pipeline Misses“ verursachen können. Die Pipeline moderner Prozessoren ist auf eine Länge von bis zu 31 Befehlen (in der NetBurst-Architektur des Pentium 4 Prozessors) gewachsen.
Zweck der Befehls-Pipeline ist es, die zukünftig voraussichtlich anstehenden Befehle im Voraus in eine Warteschlange zu laden. Die Befehle in der Warteschlange können dann mehr oder weniger parallel abgearbeitet werden, so dass sich der Gesamtdurchsatz der CPU erhöht, weil weniger Wartezeiten entstehen in denen die CPU nichts zu rechnen hat. Die

Pipeline kann aber nur *einen* möglichen Pfad durch den Programmfluss im Voraus laden. Bei einer Verzweigung kann der Programmfluss zwei mögliche Wege einschlagen. Wenn die „Branch Prediction“ der CPU die falsche Befehlskette in die Pipeline gelegt hat, kommt es bei der Verzweigung zu einem Pipeline Miss. Die Pipeline muss dann geleert und mit den Befehlen des anderen Pfades gefüllt werden. Dies kostet viele Prozessorzyklen. Die Branch Predicion ist, wie [Abe08] schreibt, ein in die CPU integrierter, in Hardware gegossener Mechanismus welcher versucht, aus dem Programmablauf den korrekten Pfad vorherzusagen, was oft, aber nicht immer, erfolgreich ist. Daher ist es auf modernen CPUs oftmals schneller, einen Algorithmus so zu formulieren dass man Verzweigungen vermeidet und durch aufwändigere mathematische Berechnungen ersetzt. In SSE-Code kann man Verzweigungen durch die Verwendung von SIMD-Masken vermeiden. Eine SIMD-Maske kann man als eine Liste von vier Boolean-Variablen verstehen. Mit dem select()-Befehl der Augenblick-Klassen (entspricht dem Compiler Intrinsic _mm_blendv_ps in SSE4) und einer Maske kann ein zu einer Verzweigung äquivalenter Befehl formuliert werden, welcher den Vorteil hat, dass er keine Verzweigung im Programmablauf verursacht. Listing 3 demonstriert die beiden Varianten,

```
1 // make sure the X component of the direction vector does
      not contain zero
2
3 // branch variant, single ray only
4 if(dir[0]==0) dir[0]=Augenblick->epsilon();
5
6 // mask select variant, four rays in parallel and no branch
      required
7 dir[0] = dir[0].select(Float4::epsilon(), (dir[0]==Float4::
      zero()));
```

**Listing 3:** Beispiel einer äquivalenten Verzweigungen und eines select()-Befehls

eine Verzweigung auszudrücken. In dem Beispiel geht es darum, Werte die Null enthalten durch eine Zahl nahe Null zu ersetzen. In Zeile vier ist die herkömmliche Verzweigung mit einer if()-Anweisung aufgeführt. In Zeile sieben ist der zu Zeile vier äquivalente select()-Aufruf gegeben. Der Ausdruck „dir[0]==Float4::zero()“ liefert eine SIMD-Maske zurück. Dort wo die Maske „true“ enthält wird der Eintrag in dir[0] durch einen Epsilon-Wert nahe Null ersetzt.

Bei der Umstellung der Implementation auf SIMD konnten durch den Einsatz von SIMD-Masken und den select()-Befehl die meisten Verzweigungen im Programmfluss eliminiert werden. Für die Abbruchbedingungen der inneren und äußeren Schleife waren Verzweigungen aber

unvermeidbar. So darf die äußere Schleife erst dann abgebrochen werden, wenn kein einziger Strahl mehr verfolgt wird. Hierzu ist eine Verzweigung notwendig, welche die Schleifenausführung abbricht wenn alle Einträge einer bestimmten Maske null sind. Dies impliziert eine horizontale Operation auf dem SIMD-Vektor, welcher durch einen Vergleich mit dem Compiler Intrinsic _mm_movemask_ps, von Augenblick in den testAny()-, testAll()- und testNone()-Aufrufen implementiert, umgesetzt werden kann.

Neben den Verzweigungen gibt es noch andere Befehle die in der Umsetzung auf der CPU teuer sind. Dazu gehören insbesondere das Runden und Umwandeln einer Gleitkomma- in eine Ganzzahl. Dies wurde in der Einzelstrahl-Implementation genutzt um mit der beschränkten Genauigkeit der Gleitkommawerte zurechtzukommen. Wie [Abe08] schreibt, kostet eine normale Typumwandlung von Float nach Integer etwa 40 Prozessorzyklen. Mit dem SSE-Befehl _mm_cvtss_si32 kann die Typumwandlung um etwa den Faktor drei beschleunigt werden. Dennoch stellt die Rundung und Typumwandlung eine Geschwindigkeitseinbuße dar, die durch Anpassungen am Algorithmus schließlich ganz eliminiert werden konnte.

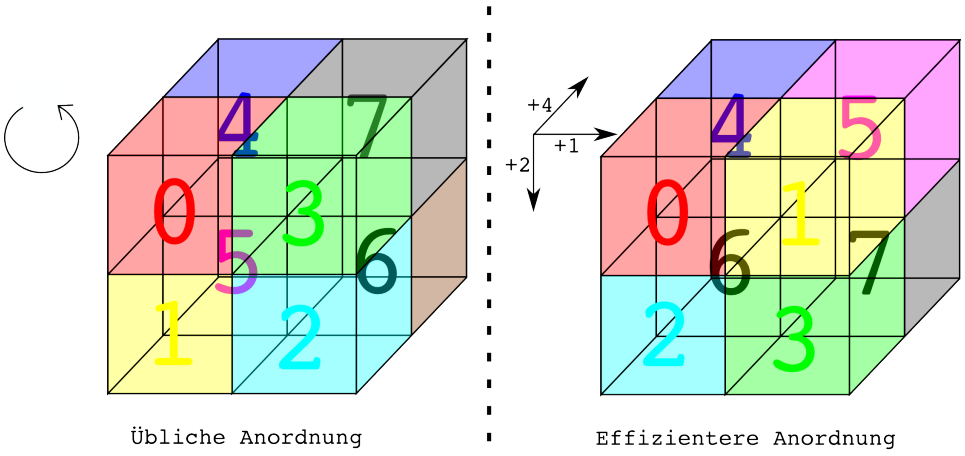


**Abbildung 26:** Vergleich der Arten, Oktanten zu indizieren

Eine weitere Optimierung des Raytracers betrifft die Nummerierung der Kinderknoten. Wie bereits in Kapitel 2.3 erwähnt, werden Oktanten üblicherweise gegen den Uhrzeigersinn durchnummeriert (siehe Abbildung 26). In jedem inneren Schleifendurchlauf muss der Index desjenigen Kindes bestimmt werden in welches der Suchalgorithmus absteigen muss. Bei der in Abbildung 26 auf der linken Seite gezeigten, üblichen Anord-

nung der Oktanten ließe sich diese Index-Bestimmung nur mithilfe einer achtelementigen Fallunterscheidung (Switch-Anweisung) implementieren. Wählt man jedoch die auf der rechten Seite der Abbildung gezeigte Anordnung der Kinder, kann die Bestimmung der Indizes durch nur drei Fallunterscheidungen und bis zu drei assoziative Additionen erfolgen. Sollte sich der gesuchte Punkt im rechten Halbraum des Voxel-Knotens befinden, wird der Index um eins inkrementiert. Sollte sich der derzeit untersuchte Punkt im unteren Halbraum befinden, wird der Index um zwei und für den hinteren Halbraum um vier inkrementiert. Listing 4

```
1  index = Int4::zero();
2  bisectors = vMin+(vMax-vMin)*Float4::half();
3
4  position = (p[0]>=bisectors[0]);
5  index = index.select(Int4::one(),position);
6  vMin[0] = vMin[0].select(bisectors[0],position);
7  vMax[0] = vMax[0].select(bisectors[0],position);
8
9  position = (p[1]>=bisectors[1]);
10 index = index.select(index+two,position);
11 vMin[1] = vMin[1].select(bisectors[1],position);
12 vMax[1] = vMax[1].select(bisectors[1],position);
13
14 position = (p[2]>=bisectors[2]);
15 index = index.select(index+four,position);
16 vMin[2] = vMin[2].select(bisectors[2],position);
17 vMax[2] = vMax[2].select(bisectors[2],position);
18
19 // descend into child voxels for those points that are
       still flagged descend
20 ((descend&&first ).testAny())?v1=&(v1->c[index[0]]):v1=v1;
21 ((descend&&second).testAny())?v2=&(v2->c[index[1]]):v2=v2;
22 ((descend&&third ).testAny())?v3=&(v3->c[index[2]]):v3=v3;
23 ((descend&&fourth).testAny())?v4=&(v4->c[index[3]]):v4=v4;
```

**Listing 4:** Der Programmcode zur Berechnung der Kinderknoten

gibt den Kern der inneren Schleife wieder, dies ist der am häufigsten ausgeführte Codeabschnitt des Raytracers.
In diesem Codeabschnitt geschehen zweierlei Dinge. Zum Einen werden für vier Voxel die Indizes der Kinder bestimmt in die abgestiegen werden soll und zum Zweiten werden die Seitenflächen dieser Kinder bestimmt. Der in Zeile 1 initialisierte SIMD-Vektor „index“ ist ein Vektor von vier Ganzzahlen, welcher zunächst vier Nullen enthält und später die Indizes der Kinder in die abgestiegen werden soll enthalten wird. In Zeile zwei werden die Trennungsebenen (bisectors) berechnet, welche den Raum des

Voxels in die acht Oktanten unterteilen. In den Zeilen 4,9 und 14 wird eine Positionsmaske berechnet, die angibt ob sich die vier Punkte „p“ auf den drei Raumachsen vor oder hinter einer Trennungsebene befinden. In den Zeilen 5, 10 und 5 des Listings werden, in Abhängigkeit von den ermittelten Positionen der Punkte, auf die Indizes die Werte eins, zwei und/oder vier addiert, je nachdem in welchen Halbräumen sich die vier Punkte befinden. In den Zeilen 6-7,11-12 und 16-17 werden die Seitenflächen der Kinder, „vMin“ und „vMax“ genannt, bestimmt. In den Zeilen 20-24 werden schließlich die vom Suchalgorithmus derzeit besuchten Voxel „v1“ bis „v4“ durch ihre jeweils ausgewählten Kinder ersetzt, d.h. es wird im Baum abgestiegen. Die dabei zunächst abgefragte Maske „descend“ gibt an, für welche der vier Punkte überhaupt noch Blätter im Baum gesucht werden.

Eine weitere Beschleunigung konnte erzielt werden in dem möglichst viel Programmcode aus den Schleifen ans Ende des Algorithmus verlagert wurde. So fand beispielsweise ursprünglich die Berechnung der Normalen und das Setzen der Farben im RGBA-Puffer in einer Verzweigung innerhalb der inneren Schleife statt. Um diese Verzweigung und den darin enthaltenen Programmcode ans Ende des Algorithmus verlagern zu können, mussten innerhalb der Schleifen zusätzliche Auflagen erfüllt werden. Damit das Setzen der Normalen und Farbe auch am Ende des Algorithmus noch funktioniert, darf nach dem Schnitt eines Strahls mit einem Voxel die den Voxel enthaltende Variable nicht mehr verändert werden. Dies könnte passieren wenn der Algorithmus für andere Strahlen aus dem Stahlenpaket weiter ausgeführt werden muss. Durch zusätzliche Abfragen der „trace“-Maske in allen relevanten Operationen konnten die zusätzlichen Auflagen erfüllt und der Code aus der inneren Schleife ans Ende des Algorithmus verlagert werden.

Wenn alle Möglichkeiten zur Beschleunigung des Programmcodes selbst ausgereizt sind, bleibt noch die Möglichkeit, die Anzahl der Ausführungen des Programmcodes selbst zu reduzieren. Die Anzahl der äußeren Schleifendurchläufe ist bereits minimal, sie entspricht der minimal notwendigen Anzahl an Abtastungen des Strahls durch Punkte. Hier kann die Ausführungshäufigkeit des Programmcodes nicht reduziert werden. Beim Suchalgorithmus aber, also der inneren Schleife - dem am häufigsten ausgeführten Teil des Codes - gibt es noch Möglichkeiten die Anzahl der Durchläufe zu minimieren.
Abbildung 27 zeigt anhand des Modells Porsche in Auflösung $1024^3$, 4,3 Millionen Voxel, wie der Aufwand zum Berechnen einzelner Strahlen im Bild schwankt. Je röter die Pixel umso mehr äußere Schleifen, also Punkte auf den Sehstrahlen, waren zur Berechnung des Pixels notwendig. Abbildung 28 zeigt beispielhaft anhand des gleichen Modells aus einem

**Abbildung 27:** Kolorierung des ungleichmäßig verteilten Berechnungsaufwands im Bild

anderen Blickwinkel auf einer Skala von grün bis rot, wo im Bild die Strahlen mit den meisten inneren Schleifendurchläufen liegen. Die aufwändigsten, rot markierten Strahlen befinden sich in einem schmalen Band entlang der Ränder des Modells. Dort streifen Strahlen Bereiche mit vielen Details, treffen aber keinen gefüllten Voxel. In dieser Darstellung sind all jene Pixel rot markiert, bei deren Berechnung insgesamt mehr als 120 innere Schleifendurchläufe notwendig sind. Bei orangefarbenen Pixeln sind es über 80, 40 bei gelb und über 15 bei grünen Pixeln. Dass sich die aufwändigsten Strahlen auf einen so schmalen Bereich des Bildes beschränken unterstreicht die Effizienz des Octree beim Kodieren des Modells. Diese Darstellung bezieht sich auf die Endfassung des Raytracers, inklusive des im Folgenden erläuterten Uniform Grid Hybrid-Ansatzes auf Ebene sieben des Baumes. Ohne den Hybridansatz wäre die Anzahl der inneren Schleifendurchläufe in dieser Darstellung um einiges höher ausgefallen.

Um die Anzahl der inneren Schleifendurchläufe pro Strahl zu reduzieren ergeben sich zwei Herangehensweisen:

- Die räumliche Auflösung, also die Tiefe des Baumes, reduzieren.
- Die Zahl der besuchten Voxel bei der Suche im Baum reduzieren.

Die räumliche Auflösung zu reduzieren, zum Beispiel über ein aggressiveres LOD, kommt nicht in Betracht da Auswirkungen auf die Darstellungsqualität bei der Optimierung nicht gewünscht sind. Somit bleibt noch die zweite Herangehensweise, also die Zahl der besuchten

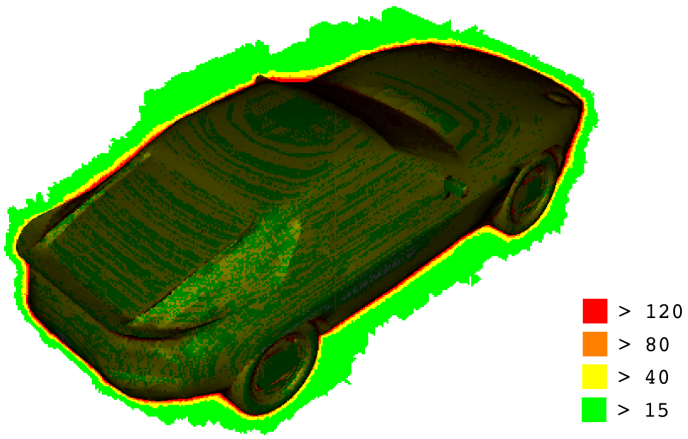


**Abbildung 28:** Hervorhebung der Strahlen mit den meisten inneren Schleifendurchläufen in einem Bild

Voxel bei der Suche im Baum zu reduzieren.
Bei näherer Betrachtung des Baumes in Abbildung 23 auf Seite 61 fällt auf, dass der Algorithmus jede Suche bei der Wurzel des Baumes beginnt, die Wurzel und ihre direkten Kinderknoten also für jeden berechneten Punkt erneut durchquert werden müssen. Die Suche bei der Wurzel des Baumes zu beginnen ist intuitiv richtig und zunächst auch der einzige mögliche Weg, da der Raytracer von der Datenstruktur nur die Speicheradresse der Wurzel kennt. Alle Zugriffe auf Elemente des Baumes müssen also mit einem Zugriff auf die Wurzel beginnen.
Dennoch gibt es Verfahren, welche die Suche an anderen Stellen als der Wurzel beginnen lassen können. Zwei Ansätze wurden im Rahmen der Optimierung implementiert.

Der erste Ansatz ist ein Bottom-Up-Verfahren, eine Nachbarschaftssuche. Dahinter steckt die Überlegung, dass aufeinander folgende Punkte räumlich relativ nah beieinander liegen und somit die Wahrscheinlichkeit hoch ist, dass auch die dazugehörigen Blätter im Octree nah beeinander liegen. Es könnte daher lohnenswert sein, die Suche nicht bei der Wurzel sondern beim letzten besuchten Voxel-Blatt zu beginnen. Von diesem Blatt im Baum ausgehend würde der Such-Algorithmus zunächst in einer zusätzlichen, zweiten inneren Schleife im Baum aufsteigen, bis ein Vorfahre gefunden ist innerhalb dessen Volumen der neue zu suchende Punkt liegt. Anschließend würde die bisherige, „absteigende“ innere Schleife von diesem Vorfahren aus die Suche abwärts durch den Baum vollenden. Abbildung 29 stellt in rot die Pfade dar, entlang derer die Nachbarschaftssuche für einen beispielhaften Strahl den Baum traver-

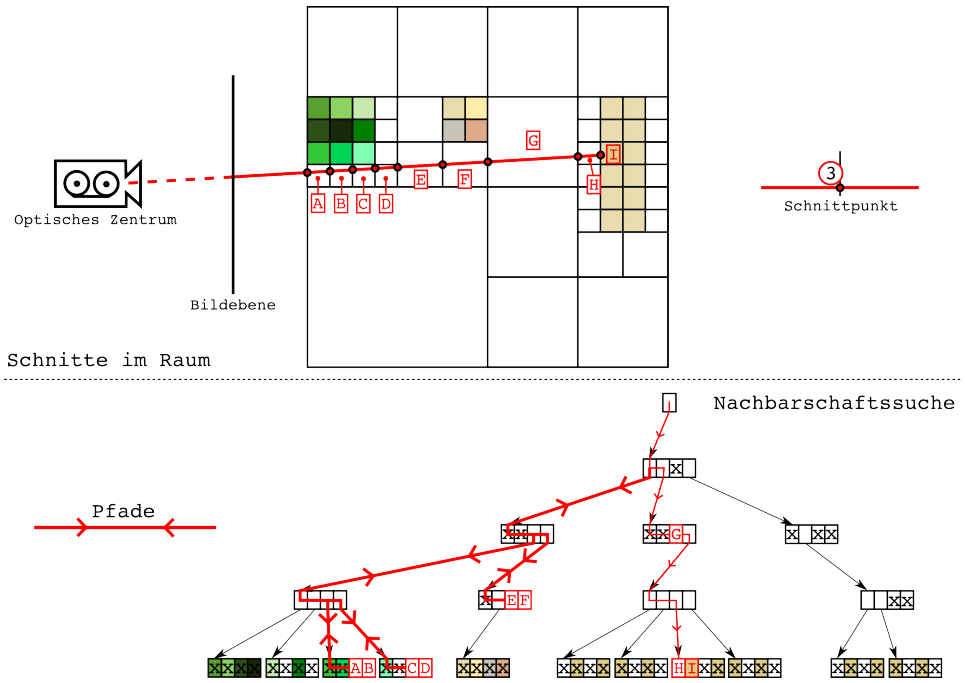


**Abbildung 29:** Pfade einer Nachbarschaftssuche

sieren würde. Im Vergleich zu Abbildung 23 auf Seite 61 fällt auf, dass in diesem Beispiel scheinbar weniger Pfade traversiert werden müssen. Denn im Optimalfall ist das gesuchte Blatt ein Geschwister-Blatt des zuletzt gefundenen Blattes (vergleiche Voxel A und B in Abbildung 29). Dann ist die Nachbarschaftssuche schon nach einem Aufstieg und einem Abstieg im Baum beendet. Im schlimmsten Fall jedoch liegt das zu findende Blatt in einem anderen Teilbaum der Wurzel (siehe den Pfad von Voxel F nach G in der Abbildung). Dann muss die aufsteigende Schleife im Baum bis zur Wurzel aufsteigen und dann von der Wurzel aus mit der absteigenden Schleife in einen anderen Teilbaum absteigen bis das Blatt gefunden ist. Dies ist offensichtlich erheblich aufwändiger als die Suche direkt bei der Wurzel zu starten. Die Nachbarschaftssuche kann also in manchen Fällen erheblich schneller, in anderen Fällen erheblich langsamer sein als der Top-Down-Ansatz. Es gibt auch keine Metrik anhand derer abzuschätzen wäre ob für ein bestimmtes Punktepaar die Nachbarschaftssuche effizienter ist als der Top-Down-Ansatz. Der räumliche Abstand der Punkte ist keine geeignete Metrik, da unabhängig vom räumlichen Abstand zweier Punkte die dazugehörigen Blätter in ganz anderen Teilbäumen liegen können[47].

[47] Beispielsweise befinden sich zwei Voxel die sich an der Grenze des linken und rechten Halbraums des Gesamtvolumens berühren in zwei verschiedenen Teilbäumen der Wurzel.

Auch wenn es keine Garantie gibt, dass zwei nah beieinanderliegende Punkte zu im Baum nah beieinanderliegenden Voxeln gehören, gibt es tendenziell sehr wohl eine Korrelation zwischen räumlicher Nähe und Nähe im Baum und es ist anzunehmen, dass die Nachbarschaftssuche unterm Strich einige absteigende innere Schleifendurchläufe erspart.
Die Nachbarschaftssuche wurde daher im Rahmen der Optimierung des Raytracers implementiert und es konnte gezeigt werden, dass durch die Nachbarschaftssuche die Zahl der inneren, im Baum absteigenden Schleifendurchläufe im Schnitt in etwa halbiert wird. Allerdings werden dafür ungefähr genauso viele Durchläufe der zusätzlichen zweiten, im Baum aufsteigenden inneren Schleife benötigt. Die Anzahl der aufsteigenden Schleifendurchläufe und die Anzahl der absteigenden inneren Schleifendurchläufe zusammengerechnet kam die Nachbarschaftssuche auf nahezu identisch viele Schleifendurchläufe wie der Top-Down-Ansatz. Damit allein wäre die Nachbarschaftssuche aber trotzdem schneller, denn die aufsteigende innere Schleife ist nicht so komplex wie die absteigende Schleife. In der absteigenden Schleife muss eines von acht Kindern bestimmt werden in das abgestiegen wird. In der aufsteigenden Schleife hingegen muss nur in das Vaterelement gesprungen und getestet werden, ob der neue zu suchende Punkt innerhalb des Vaterelementes liegt. Die Voxel enthalten jedoch aus Gründen der Speicherersparnis keinen Zeiger auf ihr Vaterelement. Um den Aufstieg im Baum zu ermöglichen muss daher ein Stapel aller Vorfahren angelegt und verwaltet werden. Beim Abstieg durch den Baum wird jeder Vorfahre auf den Stapel gelegt und beim Aufstieg wieder entfernt.
Alles in allem hat die Nachbarschaftssuche *keine* Beschleunigung der Suche im Baum erzielt. Die Ergebnisse sind mit denen des Top-Down-Ansatzes nahezu identisch, da zwar die Zahl der Abstiegs-Schleifen halbiert werden konnten, die Aufstiegs-Schleifen und die Verwaltung des Stapels diese Einsparung jedoch egalisiert haben.

Der zweite, simplere und viel effektivere Ansatz zur Reduktion der inneren Schleifendurchläufe ist es, die Top-Down-Suche nicht jedes Mal bei der Wurzel, sondern auf einer tieferen Ebene des Baumes zu beginnen. Die Vorteile sind offensichtlich. Wenn die Top-Down-Suche auf halber Höhe des Baumes gestartet werden könnte, würde sich die Zahl der inneren Schleifendurchläufe mindestens halbieren[48]. Das Problem dabei ist, wie der Algorithmus die Suche auf halber Strecke beginnen soll, wenn von der Octree-Datenstruktur nur die Adresse der Wurzel bekannt ist.
Die Lösung ist eine hybride Datenstruktur, welche die Vorteile eines Octree mit der eines Uniform Grid vereint. Der Octree bleibt dabei unverän-

[48] Sie würde sich *mindestens* halbieren, da nicht alle Suchläufe erst auf der untersten Ebene des Baumes enden würden.

dert, wird aber durch ein Uniform Grid von Zeigern auf Knoten im Baum ergänzt. Seine Stärken hat das Uniform Grid, wie Kapitel 3.4.1 ausführt, beim Zugriff auf einzelne Voxel des Rasters. Abbildung 19 auf Seite 46 illustriert, dass die Speicherposition eines Voxels im Uniform Grid direkt aus seiner Raumposition hergeleitet werden kann. Der große Nachteil des Uniform Grid ist sein explodierender Speicherverbrauch bei hohen Auflösungen. Bei niedriger Auflösung des Uniform Grid fällt sein Speicherverbrauch im Vergleich zum Octree aber kaum ins Gewicht und bietet trotzdem die Möglichkeit, die Suche etwas tiefer als bei der Wurzel zu starten.
Ausgehend von vier Byte großen Zeigern auf einem 32-Bit Betriebssystem würde beispielsweise ein Uniform Grid von $8^3$ Zeigern auf Knoten in Ebene vier des Baumes lediglich 2 Kb Arbeitsspeicher verbrauchen. Auf Ebene sieben ($64^3$ Voxel) beträgt der zusätzlicher Speicherverbrauch durch das Uniform Grid bereits ein Megabyte, auf Ebene acht wären es acht Megabyte, auf Ebene neun 64 Megabyte, usw. Wie in vielen Fällen zuvor stehen hier Speicherverbrauch und Geschwindigkeit im Konflikt. Um die Geschwindigkeit zu steigern ist mehr Speicherverbrauch notwendig. Die Frage ist, wie viel an zusätzlichem Speicherverbrauch ist akzeptabel und wie sehr lässt sich damit die Geschwindigkeit steigern?
Ein Vorteil des hier implementierten Hybridansatzes ist, dass zusätzlicher Speicherverbrauch und der daraus resultierende Geschwindigkeitszuwachs frei wählbar sind. Abhängig davon wie viel zusätzlichen Speicher man bereit ist für das Uniform Grid auszugeben, kann die Geschwindigkeit gesteigert werden.

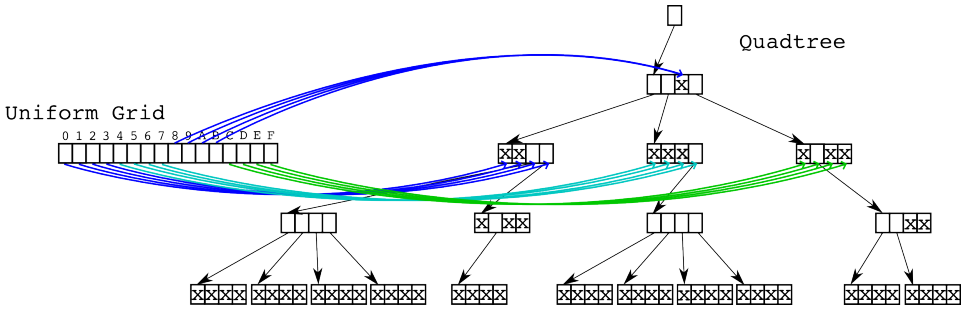


**Abbildung 30:** Hybride Datenstruktur aus Uniform Grid und einem Quadtree

Abbildung 30 illustriert ein Uniform Grid von 16 Zeigern auf Ebene drei eines Quadtree. An den Stellen, wo der Quadtree auf Ebene drei keine Voxel enthält verweisen die Zeiger auf das nächsthöhere Blatt welches diesen Raum abdeckt (Zeiger 8 bis B im Uniform Grid von Abbildung 30). Die Zeiger im Uniform Grid kann man sich als eine Art Shortcut auf tiefere Ebenen des Baumes vorstellen. Vor Beginn des Suchalgorithmus lässt sich mit geringem Aufwand berechnen, welcher Zeiger im Uniform Grid den

passenden Voxel für den Start der Suche nach einem Punkt liefert. Die innere Schleife kann dann über den Zeiger aus dem Uniform Grid direkt bei diesem Voxel die Suche beginnen. Der Geschwindigkeitsvorteil ergibt sich daraus, dass die anderenfalls notwendigen Schleifendurchläufe von der Wurzel bis zu dem Voxel auf den der Zeiger des Uniform Grid verwies entfallen können. Abbildung 31 illustriert in rot die Pfade entlang derer

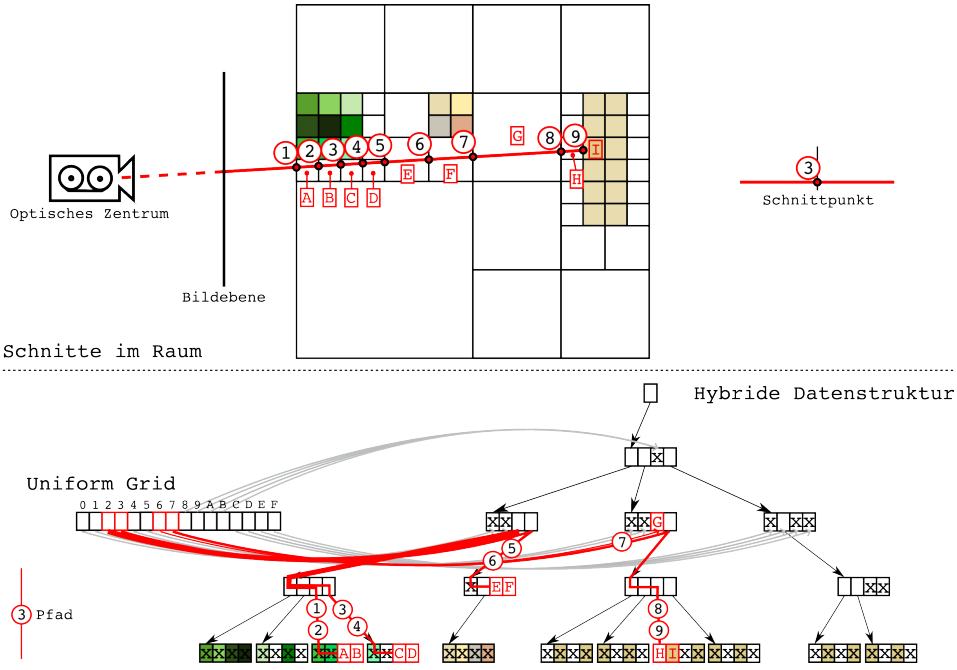


**Abbildung 31:** Pfade in einer hybriden Datenstruktur

für einen beispielhaften Strahl der Suchalgorithmus durch die hybride Datenstruktur traversieren würde. Man beachte, dass keine Pfade von der Wurzel aus traversiert werden müssen. Bemerkenswert ist ferner der Pfad zu Schnittpunkt Nummer 7. Für diesen Punkt ist die Suche bereits bei dem Voxel beendet auf den der Zeiger im Uniform Grid verweist (Voxel „G“). Für diesen Pfad muss der eigentliche Suchalgorithmus also nie gestartet werden. Abbildung 32 zeigt analog zu Abbildung 28 auf einer Skala von grün bis rot, wo in einem Rendering die Strahlen mit den meisten inneren Schleifendurchläufen liegen. In diesem Rendering wurde das Uniform Grid deaktiviert. Wie im Vergleich zu Abbildung 28 deutlich zu erkennen ist, müssen in der Variante ohne das Uniform Grid erheblich mehr innere Schleifendurchläufe getätigt werden.

In der Praxis hat es sich für die hier verwendeten Modelle gezeigt, dass ein Uniform Grid auf Ebene sieben der Bäume eine ordentliche

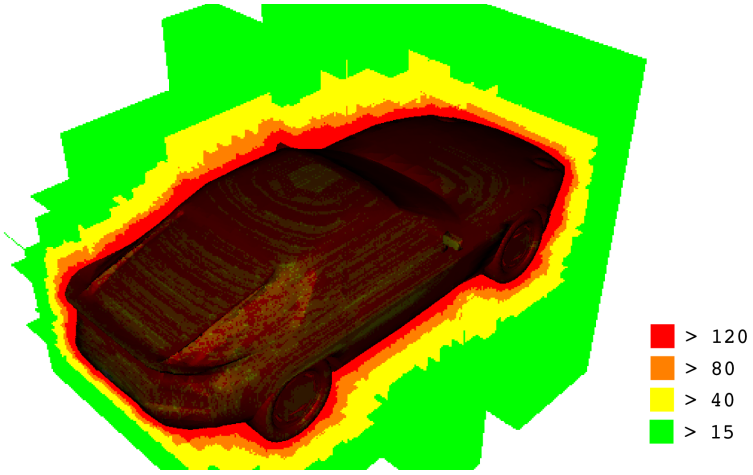


**Abbildung 32:** Hervorhebung der Strahlen mit den meisten inneren Schleifendurchläufen in einem Bild, ohne Uniform Grid

Geschwindigkeitssteigerung bei gerade noch vernachlässigbarem zusätzlichem Speicherverbrauch bietet (Auf Ebene sieben verbraucht das Uniform Grid 1 Mb zusätzlichen Speicher). Tabelle 9 auf Seite 89 führt die erzielten Bildwiederholraten für unterschiedliche Auflösungen des Uniform Grid auf. Das Uniform Grid wird im Loader-Plugin nach erfolgreichem Deserialisieren des Octree aus der „.vox“-Datei erzeugt und dem Voxel-Geometrieobjekt zusätzlich zum Octree übergeben.

Zusammenfassend lässt sich sagen, dass die hybride Datenstruktur den schnellen Zugriff auf einzelne Voxel welchen ein Uniform Grid bietet mit der Speichereffizienz des Octrees vereint. Das zusätzlich über die Welt gelegte, grobe Uniform Grid beschleunigt den Zugriff auf die Blätter im Baum, indem es die Suche im Baum verkürzt. Aufgrund der relativ geringen räumlichen Auflösung des Uniform Grid hält sich dabei sein Speicherverbrauch in akzeptablen Grenzen.

## 3.6 Erzeugung des resultierenden Framebuffers

Nachdem der Raytracer für alle Pixel des Framebuffers ausgeführt wurde ist die Verarbeitungskette noch nicht vollständig. Vor der Darstellung auf dem Bildschirm müssen noch die beiden von Augenblick benutzten Ausgabepuffer, zum Einen der RGBA-Puffer und zum Anderen der Bild-Puffer zu einem Gesamtbild verrechnet werden. Wie in Kapitel 2.5.2 bereits angeschnitten, ist es nur durch eine direkte Manipulation des RGBA-Puffers möglich die Volumendaten zu kolorieren, da Augenblick in seiner Architektur nur ein Material und damit eine Farbe pro Geometrie-

objekt vorsieht. Der reguläre Shading Execution State berechnet auf Grundlage der Schnittpunkte, der Normalen und der Materialeigenschaften des Geometrieobjekts die Helligkeit eines jeden Pixels. Sofern dies aktiviert ist wird durch diesen State auch die Verschattung (durch Aufrufe des Raytracers zur Versendung von Schattenfühlern) berechnet. Für das Voxel-Geometrieobjekt ist das Standardmaterial von Augenblick gesetzt, welches als Materialfarbe einen mittleren Grauwert besitzt und das Phong-Beleuchtungsmodell verwendet. Der Shading Execution State erzeugt also einen Bild-Puffer mit Gleitkommawerten, welcher die aus Material und Beleuchtung resultierende Farbe eines jeden Pixels enthält. Da das Standardmaterial grau ist entsprechen diese Werte in etwa der reinen Helligkeitsinformation[49].
Ohne den eingeführten eigenen Execution State würde der anschließende „Normalization Execution State“ abschließend die Information aus dem Bild-Puffer auf den RGBA-Farbraum übertragen und in den RGBA-Puffer von Augenblick schreiben, welcher dann auf dem Bildschirm dargestellt wird. Dabei würde der Normalization Execution State die vom Raytracer im RGBA-Puffer gespeicherte Farbinformation überschreiben.
Daher muss, um diesen Verlust zu vermeiden und stattdessen die Farbinformation mit der Helligkeitsinformation zu verrechnen, ein eigener Execution State nach dem Shading Execution State eingefügt werden. Dieser verrechnet pro Tile die Helligkeitsinformation aus dem Bild-Puffer mit der Farbinformation aus dem RGBA-Buffer. Dabei wird die Farbe aus dem RGBA-Puffer pro Pixel und pro Farbkanal mit der Helligkeit aus dem Bild-Puffer gewichtet, auf den RGB-Farbraum übertragen und das Resultat in den RGBA-Puffer geschrieben. Die Aufgabe des Normalization Execution State wird dabei übernommen, dieser kann also aus der Execution Unit entfernt werden.
Den regulären Shading Execution State durch einen eigenen zu ersetzen macht keinen Sinn, da in diesem Fall seine ganze Funktionalität inklusive der Schattenberechnung reimplementiert werden müsste. Abbildung 33 zeigt eine modifizierte Ausgabe des Framebuffers, wobei das linke Drittel des Bildes die reine Farbinformation und das rechte Drittel die reine Helligkeitsinformation ausgeben. In der Mitte wird die hier beschriebene Verrechnung ausgegeben.

[49] Für neutral beleuchtete Modelle wird die Helligkeitsinformation im Bild-Puffer in allen Farbkanälen identisch sein, für farbige Lichtquellen jedoch unterschiedliche Intensitäten in den Farbkanälen eines jeden Pixels enthalten.

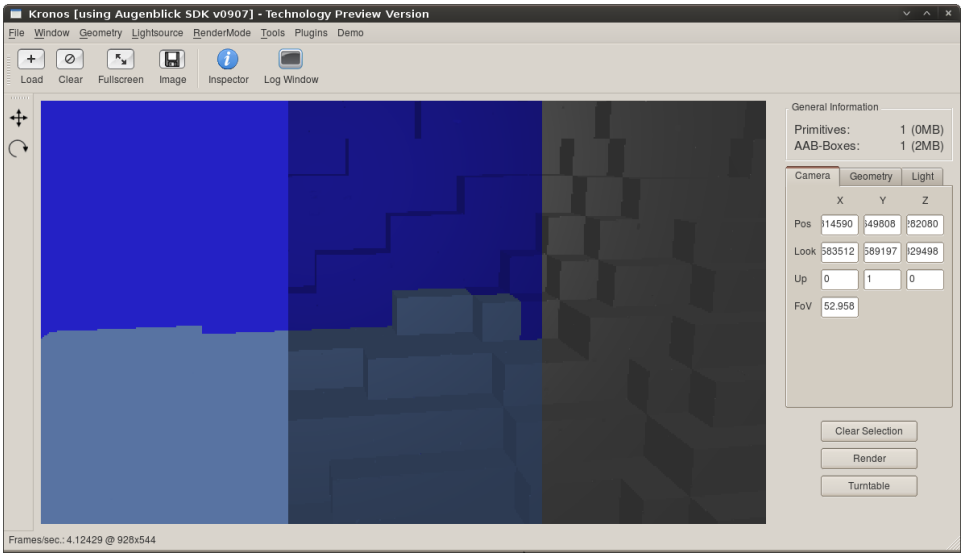


**Abbildung 33:** Verrechnung von Farbinformation (links) und Helligkeitsinformation (rechts)

```
1  /* 1. Eintrittspunkt in die Geometrie berechnen */
2  Lambda = Geometrieobjekt->Seitenflaeche - Ortsvektor /
       Richtungsvektor;
3  Punkt = Lambda*Richtungsvektor + Ortsvektor;
4  Punkt += Versatz;
5
6  /* 2. Den Strahl durch das Geometrieobjekt verfolgen */
7  Voxel = Geometrieobjekt->Wurzel;
8
9  Wiederhole endlos {
10
11    /* 2.1 Das Blatt in dem der Punkt liegt suchen */
12    Wiederhole endlos {
13
14       /* 2.1.1 Pruefen ob der gerade untersuchte Voxel ein
              gefuelltes Blatt ist */
15       Wenn(Voxel hat keine Kinder) {
16          Wenn(Voxel gefuellt) {
17             Gebe Lambda an Augenblick zurueck;
18             Programmende;
19          }
20          Beende Schleife;
21       }
22
23       /* 2.1.2 Berechne in welches Kind abgestiegen wird */
24       Index=0;
25       Wenn(Punkt im rechtem Halbraum) Index++;
26       Wenn(Punkt im unteren Halbraum) Index+=2;
27       Wenn(Punkt im hinterem Halbraum) Index+=4;
28       Voxel = Voxel->Kind[Index];
29    }
30
31    /* 2.2 Austrittspunkt aus dem gerade untersuchten Voxel
           berechnen */
32    Lambda = Voxel->Seitenflaeche - Ortsvektor /
           Richtungsvektor;
33    Punkt = Lambda*Richtungsvektor + Ortsvektor;
34    Punkt += Versatz;
35
36    Wenn(Punkt ausserhalb des Geometrieobjekts) Beende
           Schleife;
37 }
```

**Listing 5:** Der Raytracer in Pseudocode

# 4 Fazit

Dieses Kapitel gibt einen Einblick in die erzielten Ergebnisse, die Probleme auf dem Weg dorthin und einen Ausblick auf die mögliche Zukunft dieser Technologie.

## 4.1 Ergebnisse

Im Rahmen dieser Diplomarbeit konnte exemplarisch eine komplette Verarbeitungskette für interaktives Raytracing von Volumendaten umgesetzt werden. Der entwickelte Raytracer ist in der Lage auf einem handelsüblichen PC echtzeitfähige Bildwiederholraten zu liefern. Die erzeugten und dargestellten Modelle bestehen aus bis zu 300 Millionen Voxeln.

Die entwickelten Programme sind plattformunabhängig auf 32- und 64-Bit „unixoiden“ Betriebssystemen wie Linux und Mac OS X lauffähig. Da das Augenblick-SDK sowie die Augenblick-Benutzeroberfläche auch auf Microsoft Windows portiert worden sind und die entwickelten Algorithmen keine Besonderheiten von unixoiden Betriebssystemen ausnutzen, müssten die entwickelten Programme auch in diesem Betriebssystem funktionstüchtig sein, dies wurde jedoch nicht explizit getestet.

Im Folgenden sind die erzielten Ergebnisse hinsichtlich Geschwindigkeit, Speicherverbrauch und Darstellungsqualität ausgeführt. Die verwendeten Testsysteme führt Tabelle 5 auf.

### 4.1.1 Geschwindigkeit des Raytracing

Die erzielte Geschwindigkeit des Raytracings lag bei allen getesteten Modellen und allen getesteten räumlichen Auflösungen im interaktiven oder echtzeitfähigen Bereich. Die Auflösung im Bildraum entspricht, sofern nicht anders angegeben, 1312x640 Pixeln. Dies bedeutet, dass pro Bild 839.680 Strahlen in 209.920 Viererbündeln versendeten werden. Um eine leichte Reproduzierbarkeit zu garantieren wurden alle Bildwiederholraten der Modelle aus der voreingestellten Kameraposition gemessen. Ferner beziehen sich alle Werte, sofern nicht explizit anders angegeben, auf die finale Version des Raytracers mit SIMD und der hybriden Datenstruktur mit dem Uniform Grid auf Ebene sieben der Octrees.

Tabelle 6 zeigt die erzielten Bildwiederholraten ausgewählter Modelle und Auflösungen auf beiden Testsystemen in der letzten Ausbaustufe des Raytracers. Der Tabelle ist zu entnehmen, dass das Testsystem MPRO um den Faktor $6,0$ schneller ist als das Testsystem MBP. Dies liegt zum Einen daran, dass Ersteres viermal so viele Kerne hat wie Letzteres und zum Anderen daran, dass die einzelnen Kerne schneller getaktet sind, über

| Komponente | MacBook Pro 17″ |
|---|---|
| Schlüssel | MBP |
| Offizieller Name | Apple MacBook Pro (1.2) |
| Alter | > 3 Jahre |
| Prozessor (Anzahl) | Intel CoreDuo 2.16 GHz (1) |
| Architektur | 32-Bit x86 |
| Level 1 Cache | 32 KB Daten, 32 KB Anweisungen |
| Level 2 Cache | 2 MB |
| Bus Geschwindigkeit | 667 MHz |
| Arbeitsspeicher | 2.0 GB DDR2 SO-DIMM, 667 MHz |
| Grafikkarte | ATI Radeon X1600M, 256 MB |

| Komponente | Mac Pro |
|---|---|
| Schlüssel | MPRO |
| Offizieller Name | Apple Mac Pro (3.1) |
| Alter | < 1 Jahr |
| Prozessor (Anzahl) | Quad-Core Intel Xeon 2.8 GHz (2) |
| Architektur | 64-Bit x86 |
| Level 1 Cache | 32 KB Daten, 32 KB Anweisungen |
| Level 2 Cache | 3 MB pro Kern (24 MB insgesamt) |
| Bus Geschwindigkeit | 1.6 GHz |
| Arbeitsspeicher | 6.0 GB DDR2 FB-DIMM, 800 MHz |
| Grafikkarte | NVIDIA GeForce 8800 GT, 512 MB |

**Tabelle 5:** Die Hardware der Testsysteme

das Dreifache an Level 2 Cache verfügen und die SSE-Einheit der Kerne mit echten, 128 Bit breiten Vektoren arbeitet statt diese auf zwei 64 Bit Datenwörter aufzuteilen, wie es auf der im Testsystem MBP verwendeten CPU noch der Fall ist. Tabelle 7 zeigt die erzielten Bildwiederholraten ausgewählter Modelle in allen Auflösungen auf dem Testsystem MBP. Abbildung 34 auf Seite 87 stellt die erzielten Bildwiederholraten aus Tabelle 7 dar. Dem Diagramm und der Tabelle ist zu entnehmen, dass der Aufwand beim Raytracing von Octree-Volumendaten tatsächlich logarithmisch und nicht etwa linear oder exponentiell steigt. Modelle mit einer Auflösung von $4096^3$ und ca. 100 Millionen Voxeln können mit einem Drittel der Bildwiederholrate dargestellt werden wie Modelle welche aus einem einzigen Voxel bestehen. Die konkrete Gestalt der einzelnen Modelle führt dabei lediglich zu Varianzen in den Bildwiederholraten verschiedener Modelle bei gleicher Auflösung von bis zu $10\%$. Abbildung 35 auf Seite 88 stellt die Voxelmengen (logarithmisch skaliert) direkt den Bildwiederholraten gegenüber.

Tabelle 8 gibt Bildwiederholraten der vier Standard-Modelle in verschiede-

| Modell | Auflösung | Voxel | Bildrate MPRO | Bildrate MBP |
|---|---|---|---|---|
| Al | $2048^3$ | 32,8 Mio. | 12,6 | 2,1 |
| Al | $4096^3$ | 131,8 Mio. | 10,8 | 1,8 |
| Bunny | $2048^3$ | 17,4 Mio. | 13,3 | 2,2 |
| Bunny | $4096^3$ | 69,7 Mio. | 11,0 | 1,9 |
| Porsche | $4096^3$ | 72,4 Mio. | 12,4 | 2,1 |
| Soccerball | $2048^3$ | 48,2 Mio. | 8,5 | 1,4 |
| Teapot | $2048^3$ | 18,4 Mio. | 12,5 | 2,1 |
| World | $1024^3$ | 1,3 Mio. | 16,8 | 2,9 |
| World | $2048^3$ | 5,4 Mio. | 15,5 | 2,7 |
| World | $4096^3$ | 21,4 Mio. | 14,2 | 2,5 |
| World | $8192^3$ | 85,5 Mio. | 13,0 | 2,3 |

**Tabelle 6:** Bildwiederholraten pro Sekunde für ausgewählte Modelle

nen Varianten des Raytracers wieder. Die Modelle haben in allen Tests eine Auflösung von $1024^3$. Die erste Zeile gibt die Bildwiederholraten im Leerlauf wieder, d.h. wenn die Raytracing-Methode leer ist und auch sonst keinerlei eigener Code ausgeführt wird. Die in dieser Zeile erzielten Bildwiederholraten von etwa zehn Bildern pro Sekunde sind somit vollkommen unabhängig von dem geladenen Modell und die Schwankungen auf statistisches Rauschen zurückzuführen. Die zweite Zeile zeigt, dass die Bildwiederholrate um ca. $25\%$ fällt sobald die eigene Verarbeitungskette eingeschaltet wird, welche die Verrechnung des RGBA- und Bild-Puffers durchführt. Aus der dritten Zeile wird ersichtlich, dass die Geschwindigkeit durch die Aktivierung der Vorverarbeitung jedes Strahls, aber ohne die eigentliche Strahlverfolgung auszuführen, erneut um etwa $10\%$ sinkt. Zeile vier zeigt die Bildwiederholraten bei aktiver Strahlverfolgung, wobei aber durch einen auf Ebene 1 des Baumes festgelegten Detailgrad nur das Wurzelelement strahlverfolgt wird. Auch hier sind die Schwankungen der Bildwiederholraten zwischen den Modellen noch auf Rauschen zurück zu führen. Die fünfte Zeile gibt die erzielten Bildwiederholraten bei voller Auflösung des Modells in der finalen Version des Raytracers wieder, die Zahlen entsprechen denen aus Tabelle 7. Die letzte Zeile gibt die erzielten Bildwiederholraten in der Einzelstrahl-Implementation wieder, also ohne SIMD. Aus diesen Zahlen wird deutlich, dass durch die Umstellung auf vierfache Parallelverarbeitung mit dem SSE-Befehlssatz in der Praxis eine Beschleunigung um den Faktor 3 erreicht wurde. Tabelle 9 auf Seite 89 zeigt die erzielten Bildwiederholraten bei unterschiedlichen Auflösungen des Uniform Grid in der hybriden Datenstruktur. Die Bildwiederholraten aller anderen Tabellen beziehen sich auf eine hybride Datenstruktur mit dem Uniform Grid auf Ebene sieben des Baumes, also

| Auflösung | Al | Bunny | Porsche | World |
|---|---|---|---|---|
| $1^3$ | 6,6 | 6,5 | 6,7 | 6,5 |
| $2^3$ | 5,8 | 5,8 | 5,8 | 5,9 |
| $4^3$ | 5,2 | 5,4 | 5,0 | 5,0 |
| $8^3$ | 4,9 | 4,5 | 4,8 | 4,7 |
| $16^3$ | 4,4 | 4,1 | 4,5 | 4,2 |
| $32^3$ | 4,0 | 3,8 | 4,2 | 3,9 |
| $64^3$ | 3,7 | 3,6 | 3,9 | 3,7 |
| $128^3$ | 3,4 | 3,3 | 3,6 | 3,5 |
| $256^3$ | 3,1 | 3,0 | 3,3 | 3,2 |
| $512^3$ | 2,7 | 2,8 | 3,0 | 2,9 |
| $1024^3$ | 2,4 | 2,5 | 2,7 | 2,7 |
| $2048^3$ | 2,1 | 2,2 | 2,4 | 2,5 |
| $4096^3$ | 1,8 | 1,9 | 2,1 | 2,3 |

**Tabelle 7:** Bildwiederholraten ausgewählter Modelle und Auflösungen auf Testsystem MBP

| Modell (Auflösung überall $1024^3$) | Al | Bunny | Porsche | World |
|---|---|---|---|---|
| Leerlauf (leerer Methodenaufruf) | 10,0 | 9,8 | 9,9 | 10,1 |
| Mit Custom Execution Chain | 7,7 | 7,6 | 7,5 | 7,7 |
| Mit Vorverarbeitung der Strahlen | 7,1 | 7,0 | 7,2 | 7,1 |
| Level Of Detail auf Ebene 1 | 6,6 | 6,5 | 6,7 | 6,5 |
| Finale Version | 2,4 | 2,5 | 2,7 | 2,7 |
| Einzelstrahl-Implementation | 0,8 | 0,9 | 0,9 | 0,9 |

**Tabelle 8:** Bildwiederholraten ausgewählter Modelle auf Testsystem MBP in verschiedenen Entwicklungsstufen des Raytracers

einer Auflösung von $64^3$ und einem zusätzlichen Speicherverbrauch von 1 Mb.
Zunächst sind in der Tabelle als Ausgangswerte die Bildwiederholraten bei ausgeschaltetem Uniform Grid angegeben. In der zweiten Zeile finden sich die Bildwiederholraten bei eingeschaltetem Uniform Grid mit einer Auflösung von $1^3$. Ein Uniform Grid aus nur einem Voxel ergibt natürlich keinen Sinn, denn sein Zeiger wird auf die Wurzel zeigen und somit nur einen unnötigen Umweg verursachen. An den dabei erzielten, geringeren Bildwiederholraten erkennt man aber in etwa den zusätzlichen Aufwand, den die Verwaltung des Uniform Grids verursacht. Mit steigender Auflösung des Uniform Grid steigen dann auch erwartungsgemäß die Bildwiederholraten. *Allerdings erreichen sie bei einer Auflösung von $64^3$ ein Plateau (im Modell „World“ sogar bei $32^3$) und sinken danach wieder*. Dies ist nicht er-

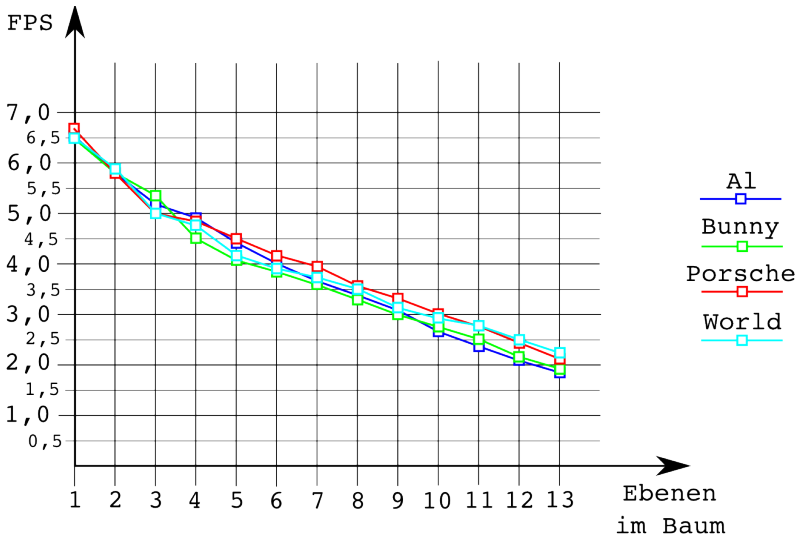


**Abbildung 34:** Darstellung der Bildwiederholraten von ausgewählten Modellen in verschiedenen Auflösungen

wartungskonform, da mit weiter steigender Auflösung des Uniform Grid die Laufzeiten des Suchalgorithmus sinken. Bei der Verwaltung des Uniform Grid entstehen allerdings auch Kosten. Dies ist im Wesentlichen das Ausrechnen der Koordinaten des Zeigers im Uniform Grid von dem aus in den Baum eingestiegen werden soll. Dazu gehört unter anderem eine, wie in Kapitel 3.5.3 beschrieben recht teure, Typumwandlung von Gleitkommazahl nach Ganzzahl. Doch sollten die Kosten zur Verwaltung des Uniform Grids eigentlich konstant sein, unabhängig von seiner Auflösung. Warum dennoch die Bildwiederholraten ab einer gewissen Größe des Uniform Grid wieder sinken konnte nicht endgültig geklärt werden.

### 4.1.2 Speicherverbrauch der Volumendaten

Der Speicherverbrauch der Datenstruktur wurde von vorneherein klein gehalten. Durch die Verwendung eines Octree konnte der Speicherverbrauch im Vergleich zu einem Uniform Grid gleicher Auflösung um mehrere Magnituden gesenkt werden. Zahlreiche Faktoren haben zu dem geringen Speicherverbrauch der hier implementierten Datenstruktur beigetragen und sind Grundlage dafür, dass einzelne Modelle aus bis zu 300 Millionen Voxeln generiert, geladen und dargestellt werden konnten:

- Die Position der Voxel im Raum wird implizit über ihre Position in der Datenstruktur definiert.
- Die Oktanten werden uniform unterteilt und daher können die Sei-

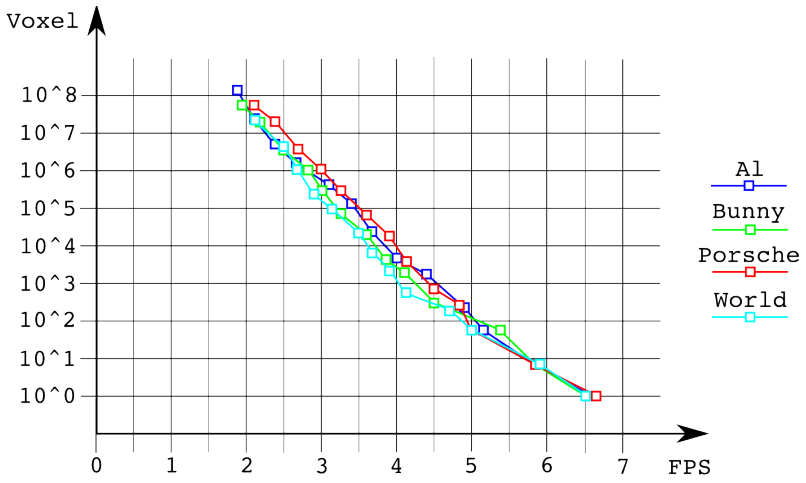


**Abbildung 35:** Bildwiederholraten im Vergleich zu Voxelmengen, logarithmisch skaliert

tenflächen der Voxel zur Laufzeit berechnet werden.

- Die Farben werden effizient in einer Farbpalette gespeichert.
- Die Normalen werden effizient in 16-bittigen Kugelkoordinaten gespeichert.
- Es werden keine weiteren Materialeigenschaften gespeichert.
- Es wurde ein hinreichend kompakte und dennoch flexible Speicherung der Datenstruktur im Arbeitsspeicher gefunden.

Bezüglich des Speicherverbrauchs des Dateiformates auf der Festplatte wurde mit der Preorder-Serialisierung eine Kodierung gefunden deren Speicherverbrauch mit nur einem Bit pro Voxel für die Datenstruktur so kompakt wie nur möglich ist.
Abbildung 47 auf Seite 106 zeigt in drei Einzelbildern das Modell „World“ in einer Auflösung von $8192^3$ Voxeln. Erst auf der letzten Zoomstufe werden einzelne Voxel erkennbar.
Tabelle 10 auf Seite 90 listet für vier ausgewählte Modelle die Gesamtzahl der Voxel des Modells bei unterschiedlichen Auflösungen. So bestehen noch alle Modelle, wenn sie mit eine Auflösung von $3^3$ erzeugt werden, aus insgesamt 73 Voxeln $(1 + 8 + 64)$. Danach beginnen die Zuwachsraten der Modelle jedoch zu schwanken. Abbildung 36 auf Seite 91 stellt die Voxel-Zuwachsraten von Ebene $x$ zu Ebene $x + 1$ für die vier Modelle dar. Aus dem Diagramm wird ersichtlich, dass bei allen Modellen zunächst die Zahl der Voxel um die Faktoren $9$ und $8,11$ steigt, anschließend je nach

| Modell (Auflösung überall $1024^3$) | Al | Bunny | Porsche | World |
|---|---|---|---|---|
| Kein Uniform Grid | 1,6 | 1,6 | 1,8 | 1,9 |
| Uniform Grid $1^3$, 4 Byte | 1,4 | 1,5 | 1,7 | 1,8 |
| Uniform Grid $2^3$, 32 Byte | 1,6 | 1,7 | 2,1 | 2,3 |
| Uniform Grid $4^3$, 256 Byte | 1,8 | 1,9 | 2,1 | 2,3 |
| Uniform Grid $8^3$, 2 Kb | 2,0 | 2,1 | 2,3 | 2,5 |
| Uniform Grid $16^3$, 16 Kb | 2,2 | 2,3 | 2,5 | 2,7 |
| Uniform Grid $32^3$, 128 Kb | 2,3 | 2,4 | 2,6 | 2,8 |
| Uniform Grid $64^3$, 1 Mb | 2,4 | 2,5 | 2,7 | 2,7 |
| Uniform Grid $128^3$, 8 Mb | 2,3 | 2,4 | 2,6 | 2,6 |
| Uniform Grid $256^3$, 64 Mb | 2,0 | 2,0 | 2,3 | 2,2 |

**Tabelle 9:** Bildwiederholraten ausgewählter Modelle auf Testsystem MBP bei verschiedenen Auflösungen des Uniform Grids in der hybriden Datenstruktur

Modell schwankt und spätestens ab einer Auflösung von $2048^3$ gegen $4,0$ konvergiert. *Dies ist eine signifikante Beobachtung, die sich mit allen im Rahmen dieser Diplomarbeit verfügbaren Modellen reproduzieren lässt*. Scheinbar ist es so, dass für alle Modelle gilt dass sich die Zahl der Voxel langfristig pro Auflösungsverdopplung vervierfacht. Zum Vergleich sei noch einmal erwähnt, dass sie sich in einem Uniform Grid pro Auflösungsverdopplung konstant verachtfacht.
Für die Konvergenz hin zu einer Vervierfachung der Voxel pro zusätzlicher Ebene im Baum muss es eine Erklärung geben. Es ist anzunehmen, dass dieser Faktor nur für Modelle gilt die durch eine infinitesimal dünne Oberfläche angenähert werden. Offenbar beträgt das Verhältnis gefüllter Blätter zu transparenter Blätter in etwa $1:1$. Bei verdoppelter Auflösung wird ein gefülltes Voxel-Blatt dann stets durch vier neue gefüllte Voxel-Blätter und vier transparente Voxel-Blätter ersetzt.
Aus dieser Beobachtung kann mit ziemlicher hoher Gewissheit auf die Größe von Modellen in Auflösungen jenseits dessen was heute ein Computer zu speichern vermag geschlossen werden. So wird, wenn man von einer weiteren konstanten Vervierfachung ausgeht, beispielsweise die „.vox"-Datei des Modells „Al" bei einer maximalen Tiefe des Octree von 20 (das entspricht einer Auflösung von $524288^3$) aus 2,1 Billionen Voxeln bestehen und 8,4 Terabyte groß sein. Das ist zwar viel Speicher, aber gemäß der Formel zu Tabelle 3 auf Seite 48 immerhin 33.362 mal kleiner als das äquivalente Uniform Grid.

Wie bereits in Kapitel 3.4.6 ausgeführt, sind die umgesetzten Datenstrukturen bezüglich ihres Nutzdaten-/Metadatenverhältnisses um einiges

| Auflösung | Al | Bunny | Porsche | World |
|---|---|---|---|---|
| $1^3$ | 1 | 1 | 1 | 1 |
| $2^3$ | 9 | 9 | 9 | 9 |
| $4^3$ | 73 | 73 | 73 | 73 |
| $8^3$ | 265 | 329 | 201 | 193 |
| $16^3$ | 1.265 | 1.129 | 841 | 657 |
| $32^3$ | 5.473 | 4.553 | 3.937 | 2.217 |
| $64^3$ | 24.385 | 17.825 | 16.553 | 7.489 |
| $128^3$ | 110.257 | 70.377 | 68.633 | 26.361 |
| $256^3$ | 474.561 | 275.697 | 277.113 | 95.145 |
| $512^3$ | 1.979.673 | 1.097.625 | 1.116.641 | 356.305 |
| $1024^3$ | 8.101.857 | 4.376.273 | 4.504.929 | 1.377.457 |
| $2048^3$ | 32.803.289 | 17.467.009 | 18.095.137 | 5.415.561 |
| $4096^3$ | 131.827.905 | 69.798.129 | 72.457.401 | 21.471.993 |

**Tabelle 10:** Umfangssteigerung in Voxeln ausgewählter Modelle mit zunehmender Auflösung

effizienter als diejenigen welche bei id Software eingesetzt werden. Aus [Oli08] kann geschlussfolgert werden, dass bei ihrer Datenstruktur im Arbeitsspeicher das Verhältnis von Nutzdaten zu Metadaten $1 : 6,42$ beträgt, also sechseinhalb Datenstruktur-Bytes pro einem Byte an eigentlicher Information. In der hier entwickelten Datenstruktur beträgt das Verhältnis $1 : 1$. Bei den Dateiformaten ist der Unterschied mit $1 : 0,142$ gegen $1 : 0,032$ ebenfalls signifikant besser.

### 4.1.3 Darstellungsqualität

Die erzielbare Darstellungsqualität war nach der Geschwindigkeit und dem Speicherverbrauch die drittwichtigste Priorität im Entwurf und der Entwicklung des Gesamtsystems. Aufgrund der ausschließlich diffusen Phong-Beleuchtung der Normalen, nicht vorhandenen Spiegelungen und nicht gegebener Transparenzen können die erzeugten Bilder im Vergleich mit „Hochglanz-Renderings“ aus dem Bereich des Raytracings von Polygonen oder der Rasterisierung von Polygonen nicht mithalten. Das Potential, einen ähnlichen Grad an Photorealismus zu erreichen wie das Raytracing von Polygonen oder Freiformflächen ist aber gegeben. Bezüglich des Detailreichtums der Modelle kann es das Raytracing von Polygonen und Freiformflächen sogar übertreffen.
Abbildung 46 auf Seite 105 zeigt das Modell „Soccerball“ mit einer Auflösung von $2048^3$, von farbigen Lichtquellen beleuchtet. Abbildung 43 auf Seite 104 zeigt das Modell „Porsche“ mit einer Auflösung von $4096^3$ und drei Halbschatten am Außenspiegel. Deutlich störend erkennbar sind in

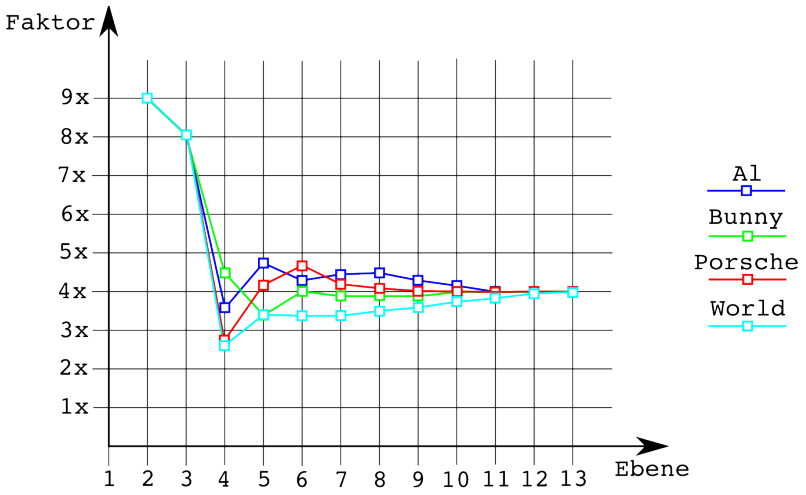


**Abbildung 36:** Darstellung der Umfangssteigerung von Modellen zwischen den Auflösungen

diesem Bild Selbstverschattungen des Modells. Kapitel 4.2.2 geht näher auf die Problematik der Selbstverschattung ein. Abbildung 44 auf Seite 104 zeigt das Modell „Al“ mit einer Auflösung von $4096^3$. Was in dieser Abbildung deutlich zum Vorschein tritt sind die sichtbaren Kanten der Polygone aus denen das Modell generiert wurde. Wie bereits erwähnt, wird der mit den Voxeln potentiell erzielbare Detailreichtum durch das Fehlen eines Modellierungswerkzeuges auf den geringeren Detailreichtum der konvertierten Polygonmodelle eingeschränkt.
Abbildung 22 auf Seite 57 zeigt, wie die Darstellungsqualität durch das Speichern und Anzeigen von individuellen Normalen eines jeden Voxels erheblich gesteigert werden konnte. Die Normalen sind dabei mit nur 16 Bit kodiert, die mit diesen Bitkombinationen darstellbaren 65.536 verschiedenen Normalen sind im Vergleich zu kartesischen Normalen geringfügig verlustbehaftet. Die geringere Auswahl an Normalen sollte sich in leicht abgestuften Schattierungen sichtbar machen. Dass diese kaum auffallen zeigt Abbildung 45 auf Seite 105. In dieser Abbildung des Modells „Bunny“ sind die entlang der Kanten der Polygone aus denen das Modell generiert wurde entstandenen Diskontinuitäten der Normalen deutlich störender sichtbar als die Abstufungen der Normalen innerhalb der Dreiecksflächen selbst.
Kritische Punkte, welche die erzielbare Darstellungsqualität von Volumendaten mindern, sind die im nachfolgenden Kapitel 4.2 aufgeführten Treppeneffekte und Selbstverschattungen.

## 4.2 Probleme

Bei der Realisierung der in dieser Diplomarbeit gesetzten Ziele stellte sich als allgemeine Herausforderung das komplette Fehlen einer Verarbeitungskette für das Raytracing von Volumendaten. Mit Ausnahme des Augenblick-SDK, welches einen guten Rahmen für die Entwicklung eines Raytracer bot, gab es keine möglichen Anknüpfungspunkte an bereits bestehende, öffentlich verfügbare Spezifikationen.
Für die Zukunft des Raytracings von Volumendaten stellt das Fehlen von Modellierungswerkzeugen und standardisierten Dateiformaten das klassische „Henne-Ei-Problem“ dar. Solange es für den Massenmarkt keine Raytracer für Volumendaten gibt werden auch keine Modellierungswerkzeuge und standardisierte Dateiformate entwickelt und solange es keine standardisierten Dateiformate gibt behindert dies die Entwicklung von massenmarkttauglichen Raytracern für Volumendaten.
Es bleibt abzuwarten, ob die bei id Software und co. vermutlich intern existierenden Werkzeuge und Konverter in Zukunft veröffentlicht werden und helfen diesen Deadlock zu beheben.

### 4.2.1 Architekturbedingte Probleme

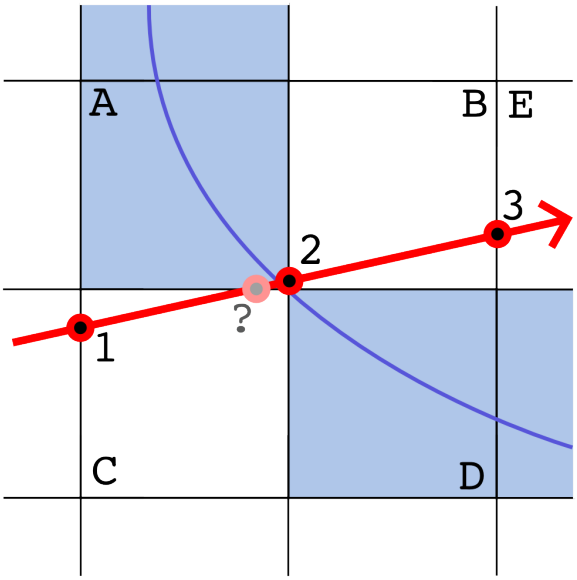


**Abbildung 37:** Fehlerhafte Abtastung der Voxel die der Sehstrahl durchquert

Ein zentrales, während der Entwicklung dieses Raytracers immer wieder auftauchendes Problem war die limitierte Gleitkommagenauigkeit. Alle mathematischen Operationen sind auf Gleitkommawerten einfacher

Genauigkeit implementiert, da dieses Format dasjenige ist welches sich mit der SSE-Einheit moderner CPUs am schnellsten berechnen lässt.
In Kapitel 3.5.1 wurde behauptet, dass kein Voxel den ein Strahl im Raum durchquert von dem hier entworfenen Raytracing-Algorithmus übersprungen würde. Diese Aussage ist nur theoretisch korrekt, in der praktischen Umsetzung kann es aufgrund der limitierten Genauigkeit von Gleitkommaoperationen zu übersprungenen Voxeln und daraus resultierenden Darstellungsfehlern kommen. So kommt es vor, dass Strahlen eine eigentlich lückenlose Oberfläche von Voxeln ohne einen Schnitt durchqueren. Abbildung 37 illustriert diesen Umstand für vier sehr kleine Voxel, welche eine infinitesimal dünne Oberfläche (blaue Kurve) annähern. In dieser Abbildung wird ein Strahl durch drei Punkte abgetastet, zu denen der Suchalgorithmus die Voxel C, B und E finden wird. Dass Voxel A, welchen der Strahl ebenfalls durchquert, übersprungen wird liegt daran, dass aufgrund der Ungenauigkeit der Gleitkommazahlen der falsche Schnittpunkt als nächster Punkt entlang des Sehstrahls ausgewählt wurde. Abbildung 38 zeigt ein reales Beispiel in dem dieser Fehler erkennbar ist.
Dieses Problem lässt sich prinzipbedingt nicht vollständig vermeiden.

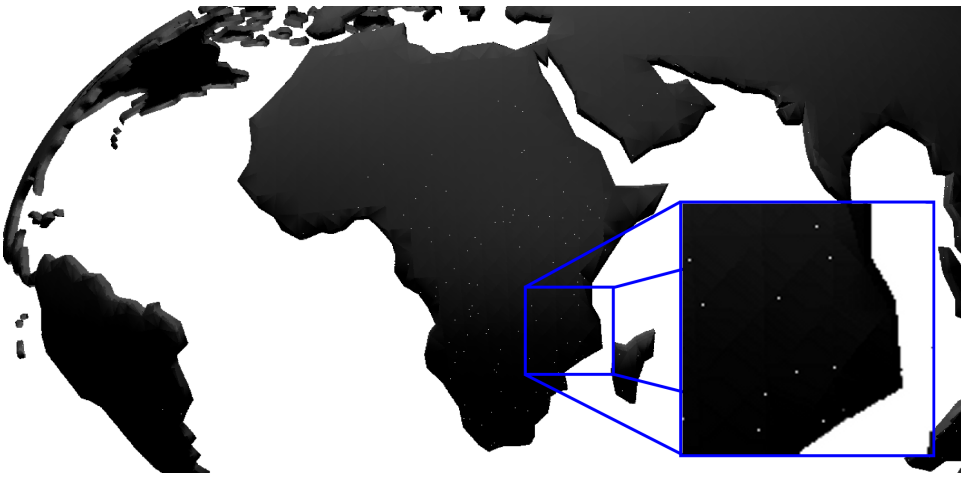

**Abbildung 38:** Beispiel einer Bildsynthese mit fehlerhaften Pixel aufgrund nicht getroffener Oberflächen

Durch Anpassungen am Algorithmus, welche bestimmte Abbruchkriterien flexibler auslegen und Justierungen an Wertebereichen des Voxel-Geometrieobjekts konnten diese Darstellungsfehler weitestgehend eliminiert werden. Allerdings verkomplizieren sich durch die Fehlertoleranz die Abbruchbedingungen. Eine Rundung der Gleitkommazahlen auf die nächstgelegene Ganzzahl würde die Problematik ebenfalls abschwächen, ist aber, wie in Kapitel 3.5.3 ausgeführt, zu teuer. Die Schwankungen der

Gleitkommazahlen durch Verwendung von Gleitkommazahlen doppelter Genaugigkeit (64 Bit statt 32 Bit pro Gleitkommazahl) zu reduzieren würde zum Einen nur die Symptome lindern und zum Anderen zu viel Geschwindigkeit kosten. Letztlich muss man bei der Entwicklung von Algorithmen die endliche Genauigkeit von Gleitkommaoperationen berücksichtigen und Vorkehrungen treffen welche die wie auch immer gelagerten Konsequenzen der begrenzten Genauigkeit kompensieren.
Eine letzte Möglichkeit, die aus der Ungenauigkeit der Gleitkommazahlen resultierenden Darstellungsfehler zu vermeiden wäre es, das Innere der Modelle im Konverterprogramm mit einem Flood-Fill-Algorithmus zu füllen. Wäre in Abbildung 37 Voxel B mit der gleichen Farbe und Normale gesetzt wie der Voxel A oder Voxel D, wäre zwar das Problem dass ein Voxel übersprungen wurde nicht behoben, für die resultierende Farbe des Pixels würde aber vermutlich das richtige Ergebnis zurückgeliefert.

### 4.2.2 Probleme der Volumendaten

Bezüglich der erzielbaren Darstellungsqualität zeigten sich bei den Volumendaten zwei zentrale Probleme:

- Treppeneffekte an Rändern und Kanten der Modelle
- Selbstverschattungen bei aktivierten Schattenfühlern

Die beiden Problemen zugrunde liegende Ursache ist die Annäherung und Diskretisierung einer glatten Oberfläche durch kubische Voxel. Sie führt zum Einen zu Treppeneffekten die denen aus dem zweidimensionalen, diskretisierten Bildraum nicht unähnlich sind und zum Anderen zu einer Selbstverschattung der Voxel.

Unter Treppeneffekten versteht man wahrnehmbare Spitzen oder „Treppenstufen“ entlang von Linien oder anderweitigen geometrischen Formen. Sie entstehen im zweidimensionalen Bildraum wenn geometrische Objekte auf ein Pixelraster mit einer festen, endlichen Auflösung diskretisiert werden. Bei zu geringer Auflösung des Rasters werden Abstufungen in den Pixeln, welche eigentlich eine kontinuierliche Linie annähern sollen, sichtbar. Die Treppeneffekte im Bildraum sind unabhängig von der gewählten Datenrepräsentationsform, treten sowohl bei der Rasterisierung als auch beim Raytracing von Polygonen, Freiformflächen oder auch Voxeln auf. Im Bildraum bekämpft man diesen Effekt durch Kantenglättungsalgorithmen, welche den Bildraum intern mit einer höheren Auflösung rasterisieren und dann auf das gröbere Raster des Ausgabemediums interpolieren.

Beim Raytracing von Voxeln können die Treppeneffekte aber zusätzlich bereits im dreidimensionalen Modell auftreten, da die Modelle bereits im Ursprungsraum zu Voxeln diskretisiert in einem dreidimensionalen Raster vorliegen. Die Treppeneffekte auf dem Modell werden allerdings nur dann für den Betrachter sichtbar, wenn die in den Bildraum abgebildeten Voxel größer sind als ein Pixel im Bildraum. Abbildung 47 auf Seite 106 zeigt in drei Einzelbildern das Modell „World“ in einer Auflösung von $8192^3$ Voxeln. Erst auf der letzten Zoomstufe werden einzelne Voxel und damit die Treppeneffekte an der Südküste Schwedens erkennbar. Solange die Voxel auf eine Fläche abgebildet werden die kleiner ist als die Fläche die ein Pixel einnimmt, werden die Treppeneffekte des Bildraumes diejenigen des Modells überdecken. Sie sind nicht verschwunden, sondern lediglich nicht sichtbar. Bewegt sich das optische Zentrum näher an den Volumendatensatz heran und gibt es keine höhere Detailstufe mehr in der das Modell beim Raytracing aufgelöst werden kann, werden die Treppeneffekte auf dem Modell auch im Bildraum sichtbar. Die Kantenglättungsalgorithmen sind auf Treppenstufen welche größer sind als ein Pixel nicht mehr wirksam. Idealerweise würde man das Problem schon im Modell, im Ursprungsraum „an der Wurzel packen“ , was aber kein triviales Unterfangen ist.

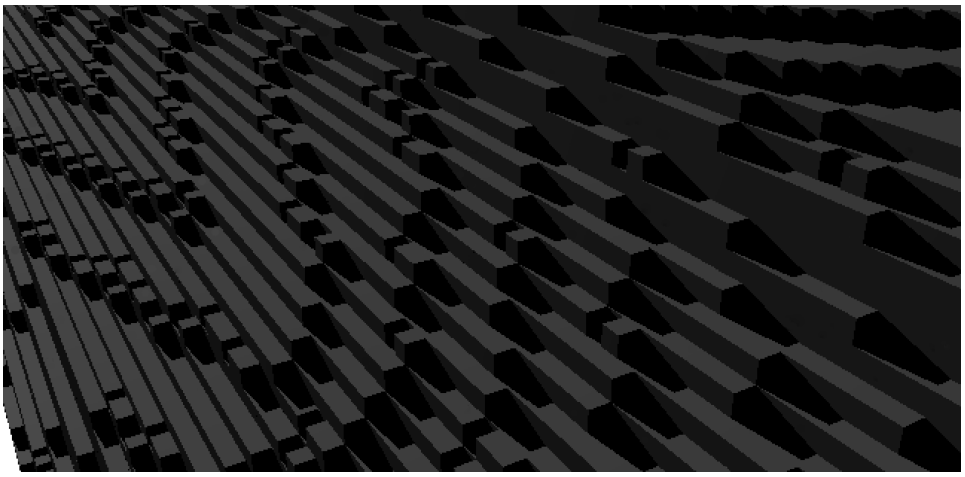

**Abbildung 39:** Nahaufnahme eines Modells mit schräg einfallendem Licht und deutlich erkennbarer Selbstverschattung der Voxel

In der kubischen Form der Voxel liegt auch die Ursache für das zweite Problem der Voxeldaten, der Selbstverschattung. Sie entsteht, wenn vom Shading Execution State Schattenfühler von den Auftreffpunkten der Primärstrahlen zu den Lichtquellen versendet werden. Dabei kann es bei

**Abbildung 40:** Selbstverschattungen in zwei verschiedenen Auflösungen des Modells „Teapot“

flach einfallendem Licht passieren, dass die Schattenstrahlen benachbarte Voxel schneiden, somit befinden sich Teile der Oberflächen der Voxel im Schatten ihrer Nachbarn. Abbildung 39 zeigt eine Nahaufnahme einer solchen Selbstverschattung. Abbildung 40 zeigt die Selbstverschattung anhand des Modells „Teapot“in den Auflösungen $512^3$ (links) und $4096^3$ (rechts). Wie der Grafik zu entnehmen ist, tritt der Selbstverschattungseffekt im linken Modell viel stärker in Erscheinung als in der rechten, höher aufgelösten Variante. Physikalisch ist die berechnete Selbstverschattung vollkommen korrekt, denn die Voxel werfen ja tatsächlich einen Schatten auf ihre Nachbarn, bloß dass in diesem Fall diese Schatten nicht gewünscht sind, da die Voxel ja eine glatte Oberfläche annähern sollen. Durch die vielen kleinen Schatten wird die kantige Struktur des aus Voxeln angenäherten Modells stark betont. Bei höherer Auflösung werden die negativen Auswirkungen der Selbstverschattung reduziert, da sich die Selbstverschattung auf viele einzelne Pixel reduziert. Dennoch sind auch in dem rechten Modell durch die Selbstverschattung ausgelöste, störende, wiederkehrende Muster erkennbar. Eine vollständige Vermeidung der Selbstverschattung ist nicht trivial, Ansätze wie etwa die Ausgangsposition der Schattenfühler etwas von der Oberfläche des Modells abzuheben wären denkbar, führen aber auch zu unerwünschten Nebeneffekten oder verfälschen das Resultat.

Um sowohl die Treppeneffekte als auch die Selbstverschattung zu kaschieren bietet es sich an, das erzeugte Bild im Bildraum weich zu zeichnen. Abbildung 41 stellt ein original Rendering des Modells „Teapot“ in einer Auflösung von $4096^3$ einer weichgezeichneten Variante des gleichen Modells gegenüber. In den vergrößerten Ausschnitten ist zu erkennen, dass jene durch die Selbstverschattung verursachten Streifenmuster in der weichgezeichneten Variante verschwunden sind. In diesem Beispiel wurde

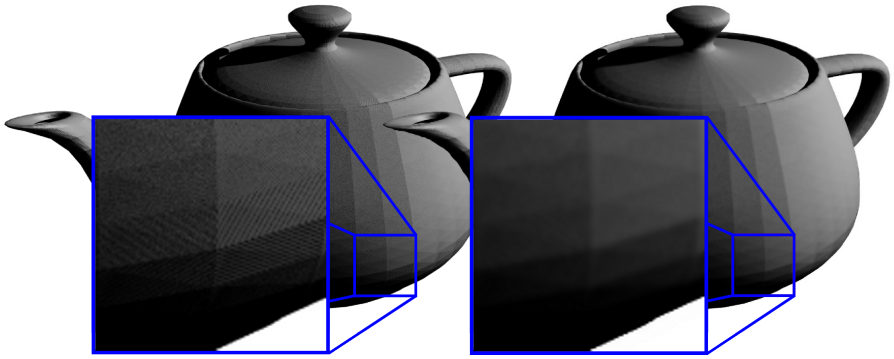

**Abbildung 41:** Vergleich des selbstverschatteten Originalbildes mit einer weichgezeichneten Version des Modells „Teapot“

ein Gauss-Filter mit einem Radius von vier Pixeln gewählt. Aus [Oli08] geht hervor, dass id Software bei der id Tech 6 Engine mit einem „Post Blur Filter“ in Abhängigkeit der Voxelgröße die beim Raytracing der Voxel entstandenen Artefakte glättet.

Um die Treppeneffekte im Bildraum, nicht die Treppeneffekte im Modell, zu beheben kann ferner „Progressive Rendering“ eingesetzt werden. Beim Progressive Rendering werden für ein Standbild kontinuierlich zusätzliche Strahlen in die Szene versandt, welche das resultierende Bild inkrementell verfeinern. Dieses Verfahren ist geeignet, hochwertige Offline-Renderings zu produzieren[50], für interaktives Raytracing kommt es jedoch nicht in Frage. Abbildung 44 auf Seite 104 eigt ein mit aktiviertem Progressive Rendering erzeugtes Bild des Modells „Al“.

## 4.3 Ausblick

In diesem Kapitel sollen zum Einen Wege aufgezeigt werden wie die hier entstandene Verarbeitungskette für das Raytracing von Volumendaten weiter verbessert werden könnte und zum Anderen ein Ausblick auf die Zukunft dieser Technologie gegeben werden.

### 4.3.1 Weitere Beschleunigung des Raytracers

Bisher wird der Detailgrad der Modelle im hier implementieren Raytracer für das komplette Modell stets einheitlich gewählt, in Abhängigkeit der Entfernung der Kamera zum Geometrieobjekt. Noch effizienter wäre es,

[50] Einige der hier enthaltenen Abbildungen wurden mittels Progressive Rendering verfeinert.

die Voxelgröße in Abhängigkeit des Durchmessers des von einem Pixel abgedeckten Raumes zu wählen. Denn für jeden Pixel im Bild wird nur ein Sehstrahl in die Welt verfolgt, tatsächlich deckt jeder Pixel der Bildebene im dreidimensionalen Raum aber einen Pyramidenstumpf ab. Abbildung

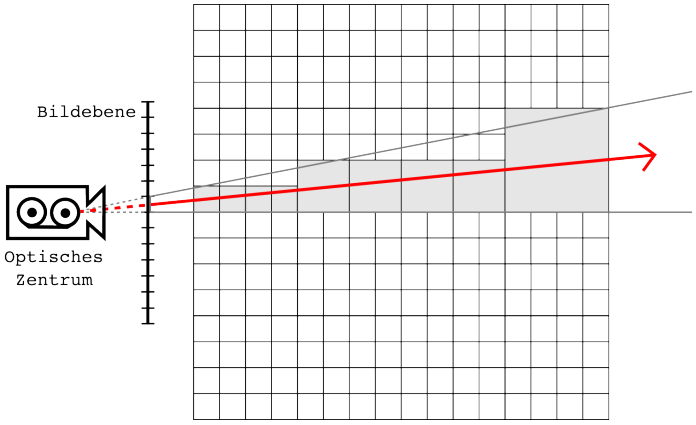


**Abbildung 42:** Illustration des Pyramidenstumpfes den ein Pixel abdeckt

42 illustriert den pyramidenstumpfförmigen Teil des Raumes den ein Pixel abdeckt. Grau hinterlegt sind diejenigen Voxel welche je nach Abstand zum optischen Zentrum die beste Annäherung an den Pyramidenstumpf liefern. [KWH09] stellt im Detail einen Algorithmus vor welcher Strahlenpakete durch einen Octree verfolgt und dabei die Voxelgröße der Größe der Pixel anpasst. Durch den Einsatz eines solchen, pixelgenauen LOD würde das Raytracing von Volumendaten mit steigender Datenmenge noch besser skalieren.

Um unabhängig von der Position der Kamera im Raum eine konstante Bildwiederholrate zu garantieren würde es sich anbieten, in Abhängigkeit der zur Verfügung stehenden Zeit den Detailgrad des Modells so zu variieren, dass eine definierte Obergrenze für die Berechnungszeit eines Bildes nicht überschritten wird. In diesem Fall würde die Darstellungsqualität in kritischen Szenen teilweise reduziert um Unterbrechungen im Animationsfluss zu vermeiden.

### 4.3.2 Erhöhung der Datenkompression

Wie Kapitel 3.4 detailliert ausführt, besteht der Octree im Arbeitsspeicher aus Knoten und Blättern die allesamt Voxel sind und jeder Voxel setzt sich aus einen vier Byte großen Zeiger und vier Byte Nutzdaten zusammen. In dieser Anordnung der Daten gibt es noch einiges Potential den Speicherverbrauch weiter zu reduzieren.

Die größte Redundanz und Verschwendung der eingesetzten Datenstruktur ist es, in jedem Blatt einen vier Byte großen Zeiger auf Null abzuspeichern. Denn das Verhältnis von Knoten zu Blättern in jedem Octree beträgt stets annäherungsweise $1 : 7$, wie Kapitel 3.4.2 ausführt. D.h. sieben von acht Zeigern in jedem Octree sind Zeiger auf Null und jeder Zeiger verursacht die Hälfte des Speicherverbrauchs eines Voxels. *Daraus folgt, dass in etwa* $\frac{3,5}{8} * 100 = 43,75\%$ *des von jedem Octree belegten Speichers aus Zeigern auf Null besteht*. Um festzulegen ob ein Voxel keine Kinder mehr hat würde aber theoretisch ein einziges Bit ausreichen, anstatt vier Byte dafür zu reservieren. Dies bedeutet, dass durch entsprechende Modifikationen an der Datenstruktur theoretisch vierzig Prozent des Speicherverbrauchs eingespart werden könnten.
Des Weiteren wäre denkbar, über zusätzliche Marker-Bits festzulegen ob ein Knoten seine Farbe und/oder seine Normale an seine Kinder vererbt. Voxel, die von einem Vorfahren Farbe und Normale erben, würden statt vier Byte für die Nutzdaten nur noch ein Bit an Nutzdaten tragen. Dieses Bit würde festlegen, ob dieser Voxel mit der vom Vorfahren geerbten Farbe gefüllt oder transparent ist. So wären ganze Teilbäume des Octree denkbar in denen alle Voxel die gleiche Farbe und ggf. auch die gleiche Normale haben.

Die Umsetzung dieser Überlegungen zur weiteren Speicherplatzersparnis ist mit Sicherheit in der einen oder anderen Form möglich, stellt sich aber auch nicht als vollkommen unproblematisch dar. So profitiert der Raytracer in der derzeitigen Implementation davon, dass alle Voxel einen einheitlichen Datentyp haben. Würden stattdessen verschiedene Typen von Voxeln unterschiedlichster Größe mit Zeigern oder ohne Zeigern, mit Farbe oder ohne Farbe, mit Normale oder ohne Normale eingeführt, würde dies im Raytracer einige zusätzliche Abfragen, Verzweigungen und Kontrollen notwendig machen welche die Geschwindigkeit des Algorithmus zweifellos reduzieren würden. Dennoch erscheint zumindest der Ansatz, die vielen Zeiger auf Null in der Blättern des Baumes zu eliminieren derart lohnenswert, dass es den zusätzlichen Mehraufwand im Raytracer rechtfertigen würde.

### 4.3.3 Steigerung der Darstellungsqualität

Wie schon Kapitel 4.1.3 erwähnt, kann die in den hier abgebildeten Renderings erzielte Darstellungsqualität nicht mit Hochglanz-Renderings aus der Rasterisierung oder dem Raytracing von Polygonen und Freiformflächen mithalten. Dies liegt schon in den limitierten, vom Konverterprogramm pro Voxel abgespeicherten Informationen begründet. Der in das Augenblick-SDK eingebettete Raytracer holt mit Farbe, Phong-Beleuchtung, Verschattungsberechnung und Progressive Rendering so

ziemlich das Maximum des Möglichen an Darstellungsqualität aus den gegebenen Datensätzen heraus. Aus Sicht des Raytracers wäre es kein Problem, diesen neben den Primärstrahlen und den Schattenfühlern auch zur Versendung von Sekundärstrahlen einzusetzen. Dies würde weitere Effekte wie Reflexionen und Transparenzen ermöglichen, womit sich die Darstellungsqualität erheblich steigern ließe. Damit diese Möglichkeiten des Raytracers ausgenutzt werden können, ist es aber unumgänglich weitere Informationen in den Voxeln abzuspeichern.
Ein Ansatz, der es ermöglichen könnte weitere Informationen in den Voxeln abzuspeichern ohne den Speicherverbrauch explodieren zu lassen wäre die Umwandlung der Farbpalettenindizes in Materialpalettenindizes. Was genau ein Material umfasst wäre dabei flexibel. Dies wäre natürlich die Farbe des Materials, aber auch weitere Eigenschaften wie Transparenz, Brechungszahlen oder Reflexionseigenschaften. Mit jeder zusätzlichen Eigenschaft eines Materials wird der Raum an Materialkombinationen um eine Dimension erweitert und umso schwieriger wird es, diesen Raum auf eine begrenzte Palette zu reduzieren. Problematisch wäre es beispielsweise, Normalen mit in die Materialeigenschaften zu übernehmen, dies würde die kombinatorischen Möglichkeiten explodieren lassen. Durch eine moderate Vergrößerung des Palettenumfangs von zwei auf drei Byte ließe sich die Palette in eine Materialpalette umwandeln, mit der 16,7 Millionen verschiedene Materialien dargestellt werden können.

Ein Ansatz, mit den gegebenen Informationen pro Voxel die Darstellungsqualität zu steigern wäre es, mehr Information aus den Polygonmodellen zu gewinnen. Hier ist vor allem das Auslesen und Auswerten von Texturen zu nennen. Aus bilinear interpolierten Texturkoordinaten eine Texelfarbe auf einen Voxel zu übernehmen wäre problemlos möglich, wurde hier aber aus Zeitmangel nicht umgesetzt. Ferner können aus Texturen nicht nur Farbinformationen gewonnen werden. In der Rasterisierung wurden trickreiche Techniken entwickelt, welche den ebenen Dreiecksflächen durch besondere Texturen mehr Details verleihen oder diese vortäuschen. Diese Verfahren werden Bump Mapping, Displacement Mapping und Normal Mapping genannt. Sie wären für das Konverterprogramm wertvolle Informationsquellen die es erlauben würden, den erzielbaren Detailreichtum der Volumendaten stärker auszureizen. Aufgrund der in den Displacment Maps enthaltenen Information könnte der 3D-Scan-Konverter dazu übergehen, die generierten Voxel nicht auf die Fläche des Dreiecks einzuschränken, sondern den Informationen aus der Displacement Map entsprechend räumlich anzuordnen. Aus Normal Maps könnten individuelle Normalen pro Voxel gewonnen werden anstatt diese aus den Eckpunktnormalen bilinear zu interpolieren.

Schließlich ließe sich die Darstellungsqualität noch weiter steigern

wenn es Modellierungswerkzeuge gäbe die von vornherein auf Volumendaten ausgelegt sind. Solange ein Konverterprogramm zur Erzeugung der Volumendaten notwendig ist, ist die Darstellungsqualität des Voxel-Raytracings auf den Informationsgehalt des Polygonmodells limitiert und kann somit zwangsläufig die Darstellungsqualität des Polygon-Raytracings höchstens erreichen, nicht aber übertreffen. Ein Modellierungswerkzeug für Volumendaten könnte von Haus aus Modelle erzeugen die nicht nur auf Ihre Oberfläche reduziert sind. Eine klar definierte Oberfläche wäre gar nicht mehr notwendig, stattdessen könnte die Oberfläche, wenn gewollt, eine sehr unregelmäßige Form annehmen. Es wäre prädestiniert für raue, unebene Oberflächen wie sie in der Realität an vielen Stellen anzutreffen sind, insbesondere Haut, Haare, Wände, Erdboden, Gras uvm. Für derartige raue Oberflächen würde auch die Selbstverschattung der Voxel ein geringeres Problem darstellen, sie entspräche in der Regel einem physikalisch korrekten, gewünschten Effekt. Problematisch bleibt die Selbstverschattung nur dann, wenn durch Voxel glatte, ebene Flächen angenähert werden sollen.

### 4.3.4 Die mögliche Zukunft des Raytracings von Volumendaten

Angesichts des Interesses welches die Industrie in jüngster Zeit für das Raytracing von Volumendaten gezeigt hat, scheint die Zukunft für diese Technologie spannend zu werden. Auf Seiten der Software wird es vor allem interessant sein zu sehen was id Software mit der für 2012 geplanten Spiele-Engine „id Tech 6“ aus dieser Technologie macht, sofern John Carmack seiner Ankündigung, diese Technologie in der nächsten Engine einzusetzen (siehe [Shr08]), Taten folgen lässt. Womöglich wird dies die erste Implementation sein die den Massenmarkt erreicht und auf unterschiedlichsten PCs, Konsolen, Betriebssystemen, CPUs und GPUs lauffähig sein muss. Die bisher veröffentlichten Details klingen vielversprechend (siehe [Oli08]).
Auf Seiten der Hardware bleibt natürlich die Entwicklung der CPUs und GPUs das beherrschende Thema. OpenCL und die Compute Shader aus DirectX 11 (siehe [Boy08]) könnten helfen, den GPGPU-Sektor zu vereinheitlichen und somit massenmarkttaugliche Raytracer von Volumendaten auf den GPUs der nächsten Generation ermöglichen. Vor allem das Larrabee-Projekt von Intel hat das Potential, mit seiner prognostizierten Rechengewalt von 2 oder mehr TeraFLOPS den Markt der Computergrafik kräftig aufzumischen. Diese neue, noch flexiblere Grafik-Hardware bietet vielleicht noch stärker als die traditionellen GPUs die Chance Raytracing massenmarkttauglich zu machen, ob nun von Voxeln, Polygonen oder beidem.

Wenn Voxel-Raytracing die Rasterisierung *vollständig* ablösen sollte, muss in Zukunft geklärt werden wie Volumendaten effizient zu animieren sind. Bei Polygondatensätzen werden die kartesischen Koordinaten eines jeden Eckpunktes gespeichert, und diese lassen sich mit Matrixmultiplikationen leicht rotieren, skalieren und verschieben. Durch den Einsatz eines Szenengraphen können diese Operationen statt auf einzelne Dreiecke auf ganze Objekte oder Gruppen von Objekten angewandt werden. Mit „Skeletal Animation“ und „Motion Capturing“ gibt es ausgereifte Systeme zur Bewegungsanimation.
Bei Volumendaten, und Voxel-Octress im Speziellen, steckt die Positionsinformation der geometrischen Grundeinheit Voxel implizit in der Datenstruktur. Will man einen Voxel verschieben, kann nicht analog zu den Eckpunktkoordinaten der Dreiecke ein Wert in der Datenstruktur verändert werden, sondern es muss die Position des Voxels in der Datenstruktur verändert werden. Dies würde selbst für kleinste Animationen umfangreiche Lese- und Schreibzugriffe auf der Datenstruktur verursachen, welche dies für die Praxis untauglich machen. Eventuell könnte durch den Einsatz mehrerer Octrees, welche teils dynamisch, teils statisch in der Welt positioniert, separat gerendert und anschließend auf Grundlage der Tiefeninformation verrechnet werden einen Teil der Animationsprobleme gelöst werden. Dies hätte logischerweise negative Auswirkungen auf die Geschwindigkeit der Bildsynthese.

Ein Weg über den das Animierbarkeitsproblem umgangen werden kann ist, die Animationen nach wie vor auf Polygonen auszuführen und erst in einem anschließenden Verarbeitungsschritt den Polygondatensatz in einen Volumendatensatz umzuwandeln. [ED06] stellt mit der „Fast Scene Voxelization“ einen Algorithmus vor mit dem auf modernen Grafikkarten intern und zur Laufzeit die Dreiecke innerhalb des View Frustums in ein Uniform Grid von Voxeln transformiert werden können. Da ein solcher Ansatz weiterhin im Kern auf Polygondatensätzen beruht, löst er aber nicht die in Kapitel 1.3.2 beschriebenen Probleme die man mit Volumendaten zu beheben versucht.
Sofern das Animationsproblem der Volumendaten eines Tages gelöst wird, könnten Volumendatensätze in noch fernerer Zukunft neben ihren optischen Eigenschaften um zusätzliche physikalische Eigenschaften wie Masse, Härte, Reibung, etc. ergänzt werden. Aus Masse durch Volumen ergäbe sich die Dichte des Voxels. Mit diesen Eigenschaften versehen könnten auch physikalische Effekte direkt auf die Volumendaten angewandt werden. Diese Überlegungen setzen jedoch erhebliche technologische Fortschritte voraus, welche eine sehr flexible und dynamische Speicherung der Voxel ermöglichen. Es ist auch fraglich, ob sich das „Teile und Herrsche“-Prinzip der Hierarchien überhaupt mit den Anforderungen nach größtmöglicher physikalischer Dynamik vereinen lässt.

Da das Animationsproblem vorerst ungelöst bleibt ist klar, dass Volumendaten mittelfristig die Rasterisierung oder das Raytracing von Polygonen *nicht* vollständig ersetzen kann. Ein solch radikaler Ansatz ist auch in sofern unsinnig, als jede zukünftige Hardware-Generation, egal wie flexibel sie ausgelegt sein wird, in jedem Fall auch immer ein schneller Rasterisierer sein wird und es wäre suboptimal diese Fähigkeiten brach liegen zu lassen. Daher erscheint es am sinnvollsten, in einem *Hybrid-Ansatz* die Vorteile der Polygondaten mit denen der Volumendaten zu kombinieren. Es macht Sinn, die Polygondaten überall dort einzusetzen wo sie ihre Vorteile ausspielen können. Dies sind vor allem Objekte die animiert werden müssen, sowie alle glatten, glänzenden Oberflächen. Ob man das Sichtbarkeitsproblem für die Polygondaten durch Rasterisierung oder Raytracing löst wäre dabei von untergeordneter Bedeutung. Die Volumendaten haben ihre Vorteile im immensen Detailreichtum welcher durch die Repräsentation durch Voxel in einem Raster und die bessere Skalierbarkeit der Datenmengen erzielt werden kann. Sie sind ideal für große, statische Geometrien mit rauen Oberflächen. Dies macht Volumendaten zu guten Kandidaten um beispielsweise Gelände und Gebäude, Erdböden und Wände zu repräsentieren. Die Fusion der aus den Polygondaten und den Voxeldaten gewonnenen Bilder kann über einen Vergleich zweier Tiefenpuffer leicht gelöst werden. Problematischer wird es bei der gegenseitigen Verschattungsberechnung, aber auch dies ist kein unlösbares Problem.
Alles in allem scheinen Hybridansätze also die Zukunft des interaktiven Raytracings von Volumendaten in der Computerspiele-Industrie zu sein.

# 5 Anhang

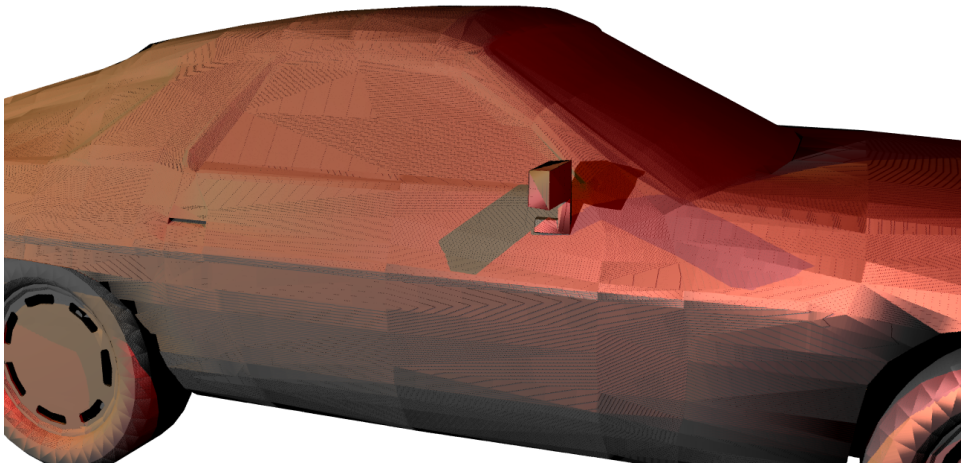

**Abbildung 43:** Modell „Porsche“ mit mehreren Halbschatten

**Abbildung 44:** Modell „Al“ mit aktiviertem Progressive Rendering

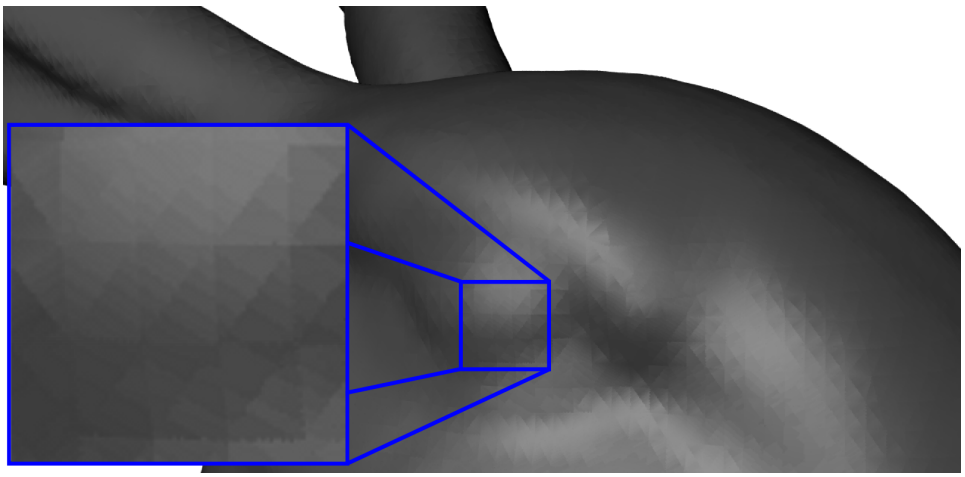

**Abbildung 45:** Nahaufnahme des Modells „Bunny“ mit leichten Abstufungen der Normalen

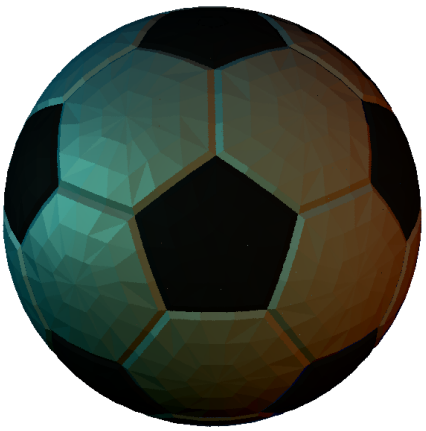

**Abbildung 46:** Modell „Soccerball“ mit farbigen Lichtquellen

**Abbildung 47:** Drei Zoomstufen des Modells „World“ mit einer Auflösung von 8192*8192*8192 Voxeln

# Abbildungsverzeichnis


1 Ausgabe des Voxel-Octree-Raytracers . . . . . 2
2 Albrecht Dürer: Mann, eine Laute zeichnend[51] . . . . . 14
3 Schematisches Konzept des Raytracing[52] . . . . . 15
4 Illustration der Instruktionsparalellität und Datenkohärenz . 18
5 Ein beispielhafter Octree und die dazugehörige Raumunterteilung . . . . . 19
6 Anordnung der Kinder in einem Octree . . . . . 20
7 Ein beispielhafter Quadtree und die dazugehörige Raumunterteilung . . . . . 21
8 Die Einzelprogramme und der Datenfluss zwischen diesen . 27
9 Reduktion eines Farbraums auf eine Palette[53] . . . . . 28
10 Iterative Reduktion eines Farbraums auf eine gewünschte Größe . . . . . 30
11 Schema der 3D-Scan-Konvertierung . . . . . 32
12 Flächendeckende Abtastung in einem gleichmäßigen Gitter entlang zweier orthogonaler Vektoren . . . . . 34
13 Diskretisierung einer Dreiecksfläche in Voxel im 3D-Scan-Konverter-Verfahren . . . . . 35
14 Bedeutung der Winkel Theta und Phi im Kugelkoordinatensystem[54] . . . . . 36
15 Vergleich zweier Detailstufen eines Modells . . . . . 37
16 Kompression überflüssiger Voxel in einem Quadtree . . . . . 38
17 Beispiel einer „.vox“-Datei in hexadezimaler Schreibweise . 39
18 Preorder-Serialisierung eines Quadtree . . . . . 42
19 Ein Uniform Grid aus 8x8x8 Voxeln und der Zugriff im Speicher . . . . . 46
20 Vergleich der Voxelzahl in einem Uniform Grid und einem Quadtree . . . . . 48
21 Naive und optimierte Kodierung des Octree im Vergleich . . 53
22 Gegenüberstellung von zur Laufzeit generierten Normalen auf den Seitenflächen der Voxel und den aus Eckpunktnormalen interpolierten Normalen . . . . . 57
23 Die Vorgehensweise des Raytracers beim Durchqueren eines Quadtree . . . . . 61
24 Die Schnittpunkte eines Strahls mit den Seitenflächen eines Voxels . . . . . 62
25 Schnitt- und Versatzpunkte eines Strahls. . . . . 63
26 Vergleich der Arten, Oktanten zu indizieren . . . . . 70
27 Kolorierung des ungleichmäßig verteilten Berechnungsaufwands im Bild . . . . . 73
28 Hervorhebung der Strahlen mit den meisten inneren Schleifendurchläufen in einem Bild . . . . . 74

29 Pfade einer Nachbarschaftssuche . . . . . 75
30 Hybride Datenstruktur aus Uniform Grid und einem Quadtree 77
31 Pfade in einer hybriden Datenstruktur . . . . . 78
32 Hervorhebung der Strahlen mit den meisten inneren Schleifendurchläufen in einem Bild, ohne Uniform Grid . . . . . 79
33 Verrechnung von Farbinformation (links) und Helligkeitsinformation (rechts) . . . . . 81
34 Darstellung der Bildwiederholraten von ausgewählten Modellen in verschiedenen Auflösungen . . . . . 87
35 Bildwiederholraten im Vergleich zu Voxelmengen, logarithmisch skaliert . . . . . 88
36 Darstellung der Umfangssteigerung von Modellen zwischen den Auflösungen . . . . . 91
37 Fehlerhafte Abtastung der Voxel die der Sehstrahl durchquert 92
38 Beispiel einer Bildsynthese mit fehlerhaften Pixel aufgrund nicht getroffener Oberflächen . . . . . 93
39 Nahaufnahme eines Modells mit schräg einfallendem Licht und deutlich erkennbarer Selbstverschattung der Voxel . . . 95
40 Selbstverschattungen in zwei verschiedenen Auflösungen des Modells „Teapot“ . . . . . 96
41 Vergleich des selbstverschatteten Originalbildes mit einer weichgezeichneten Version des Modells „Teapot“ . . . . . 97
42 Illustration des Pyramidenstumpfes den ein Pixel abdeckt . 98
43 Modell „Porsche“mit mehreren Halbschatten . . . . . 104
44 Modell „Al“mit aktiviertem Progressive Rendering . . . . . 104
45 Nahaufnahme des Modells „Bunny“mit leichten Abstufungen der Normalen . . . . . 105
46 Modell „Soccerball“mit farbigen Lichtquellen . . . . . 105
47 Drei Zoomstufen des Modells „World“mit einer Auflösung von 8192*8192*8192 Voxeln . . . . . 106

## Listings

1 Rekursive Methode zum Deserialisieren von „.vox“-Dateien 44
2 Umwandlung der Normalen von Kugelkoordinaten in kartesische Koordinaten . . . . . 65
3 Beispiel einer äquivalenten Verzweigungen und eines select()-Befehls . . . . . 69
4 Der Programmcode zur Berechnung der Kinderknoten . . . 71
5 Der Raytracer in Pseudocode . . . . . 82

# Tabellenverzeichnis


1 Vergleich aktueller CPUs und GPUs . . . . . 3
2 Die Flynnsche Klassifikation . . . . . 21
3 Speicherverbrauch und Kompressionsfaktor verschiedener Modelle im Octree . . . . . 48
4 Entwicklung des Speicherverbrauchs eines Octree im schlimmsten Fall . . . . . 49
5 Die Hardware der Testsysteme . . . . . 84
6 Bildwiederholraten pro Sekunde für ausgewählte Modelle . 85
7 Bildwiederholraten ausgewählter Modelle und Auflösungen auf Testsystem MBP . . . . . 86
8 Bildwiederholraten ausgewählter Modelle auf Testsystem MBP in verschiedenen Entwicklungsstufen des Raytracers . 86
9 Bildwiederholraten ausgewählter Modelle auf Testsystem MBP bei verschiedenen Auflösungen des Uniform Grids in der hybriden Datenstruktur . . . . . 89
10 Umfangssteigerung in Voxeln ausgewählter Modelle mit zunehmender Auflösung . . . . . 90

## Literatur

[Abe08] Oliver Abert. *Augenblick: Ein effizientes Framework für Echtzeit Ray Tracing*. PhD thesis, Universität Koblenz-Landau, Koblenz, November 2008.

[ABNS93] D. Allen, M.S. Banks, A.M. Norcia, and L. Shannon. Does chromatic sensitivity develop more slowly than luminance sensitivity? *VISION RESEARCH-OXFORD-*, 33:2553–2553, 1993.

[App68] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 37–45. ACM New York, NY, USA, 1968.

[Bet06] Carsten Bethin. *Realtime Ray Tracing On Current CPU Architectures*. PhD thesis, Universität des Saarlandes, Saarbrücken, January 2006.

[Boy08] Chas. Boyd. The DirectX 11 compute shader. Siggraph 2008, August 2008.

[Bur05] MF Burnyeat. Archytas and Optics. *Science in Context*, 18(01):35–53, 2005.

[CFLB06] P.H. Christensen, J. Fong, D.M. Laur, and D. Batali. Ray tracing for the movie'Cars'. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 1–6, 2006.

[Chr05] Martin Christen. Ray tracing on GPU. Master's thesis, Univ. of Applied Sciences Basel (FHBB), January 2005.

[CNL08] Cyril Crassin, Fabrice Neyret, and Sylvain Lefebvre. Interactive gigavoxels, June 2008. http://hal.inria.fr/docs/00/29/71/63/PDF/rap-rech2-num.pdf.

[CY01] S.C. Cheng and C.K. Yang. A fast and novel technique for color quantization using reduction of color space dimensionality. *Pattern Recognition Letters*, 22(8):845–856, 2001.

[ED06] E. Eisemann and X. Décoret. Fast scene voxelization and applications. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 71–78. ACM New York, NY, USA, 2006.

[Fly72] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[GKHS98] N. Gagvani, D. Kenchammana-Hosekote, and D. Silver. Volume animation using the skeleton tree. In *Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 47–53. ACM New York, NY, USA, 1998.

[GMI08] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008. Proc. CGI 2008.

[Hec82] Paul Heckbert. Color image quantization for frame buffer display. *ACM SIGGRAPH Computer Graphics*, 16(3):297–307, 1982.

[Inc08] W.G.A. Inc. Global Entertainment and Media Outlook: 2008-2012. *New York: Price Waterhouse Coopers LLP*, 2008.

[Jen96] H.W. Jensen. Global illumination using photon maps. *Rendering Techniques*, 96:21–30, 1996.

[KS87] A. Kaufman and E. Shimony. 3D scan-conversion algorithms for voxel-based graphics. In *Proceedings of the 1986 workshop on Interactive 3D graphics*, pages 45–75. ACM New York, NY, USA, 1987.

[KWH09] A.M. Knoll, I. Wald, and C.D. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer*, 25(3):209–225, 2009.

[LHK+04] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, New York, NY, USA, 2004. ACM Press.

[NC05] Siva G. Narendra and Anantha Chandrakasan. *Leakage in Nanometer CMOS Technologies (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Oli08] Jon Olick. Next generation parallelism in games. Siggraph 2008, August 2008. Slides from course: Beyond Programmable Shading.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[PGSS07] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, September 2007. (Proceedings of Eurographics).

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

[Poh09] Daniel Pohl. Light it up: Quake wars gets ray traced. *Visual Adrenaline Magazine*, 2:34–38, 2009.

[Rab08] Hanno Rabe. Ray-Tracing mit CUDA. Master's thesis, Universität Koblenz-Landau, September 2008.

[RPK00] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE micro*, pages 47–57, 2000.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

[Shr08] Ryan Shrout. John carmack in id tech 6, ray tracing, consoles, physics and more. Interview mit PC Perspective, http://www.pcper.com/article.php?aid=532, besucht im September 2009, March 2008.

[Swe09] Tim Sweeney. The end of the gpu roadmap. High Performance Graphics Conference 2009, August 2009. Keynote presentation.

[Ver95] O. Verevka. Color image quantization in windows systems with local k-means algorithm. In *Proceedings of VI Western Computer Graphics Symposium*, pages 74–79. Citeseer, 1995.

[Wal06] I. Wald. Realtime ray tracing and interactive global illumination. *IT-MUNCHEN-*, 48(4):242, 2006.

[Whi80] T. Whitted. An improved illumination model for shaded display. 1980.

[WS05] Sven Woop and Jörg Schmittler. Rpu: A programmable ray processing unit for realtime ray tracing. In *ACM Trans. Graph*, pages 434–444, 2005.

[Yer09] Cevat Yerli. The future of gaming graphics. Games Developer Conference Europe 2009, August 2009. Keynote presentation.