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
(Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter: Dr. Oliver Abert
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 Voxel .......t .......................... 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

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


Ein vielversprechender Ansatz zur Bewältigung der Datenmengen und weiterer 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 forschten 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 Motivation


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


<table>
<thead>
<tr>
<th>Typ</th>
<th>Modell</th>
<th>Transistoren</th>
<th>Speicherbandbreite</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7 9xx</td>
<td>0,7 Mrd.</td>
<td>12 Gb/s</td>
</tr>
<tr>
<td>CPU</td>
<td>AMD Athlon II X2</td>
<td>0,2 Mrd.</td>
<td>33 Gb/s</td>
</tr>
<tr>
<td>GPU</td>
<td>Nvidia GeForce GTX 295</td>
<td>2 x 1,4 Mrd.</td>
<td>223,8 Gb/s</td>
</tr>
<tr>
<td>GPU</td>
<td>AMD Radeon HD 5870</td>
<td>2,1 Mrd.</td>
<td>153,6 Gb/s</td>
</tr>
</tbody>
</table>

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 Rechenleistung 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-

1http://www.intel.com/design/corei7/documentation.htm
4http://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.


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 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.

- 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.

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

---

3http://www.heise.de/newsticker/meldung/143128
4http://sites.amd.com/us/fusion/Pages/index.aspx
5http://www.intel.com/technology/visual/microarch.htm
1.1 Vor- und Nachteile der CPU

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]).


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-Beschleunigungskarte an, welche das Raytracing im Vergleich zur Ausführung auf der CPU um den Faktor 20 beschleunigen soll. 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 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

---

7http://www.caustic.com/caustic-rt_caustic-one.php
8http://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\(^9\).


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 Program-
mieren 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 hardwarebe-
dingt 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 Datenstruk-
tur und eines auf SIMD-Operationen optimierten Programmcodes erreicht
der Augenblick Raytracer auf modernen CPUs echtzeitfähige Bildwieder-
holraten [Abe08]. Mehrere Gründe sprechen für eine Umsetzung auf Basis
des Augenblick-SDK.

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

2. Kernelemente eines Raytracers, wie der Szenengraph zur Verwal-
tung 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.


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

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

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:


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

\textsuperscript{13}Kaustiken sind Helligkeitsmuster die bei der Brechungen von Lichtstrahlen bei dem durchqueren von Materialien mit unterschiedlichen Brechungsindizes entstehen.

1.3.2 Raytracing von Voxeln


Wenn man von Rasterisierung auf Raytracing umsteigt, entfällt der Zwang mit Polygonen zu arbeiten\(^\text{15}\). 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.


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

16

nachempfinden und wird gerade für Quake Wars umgesetzt (siehe [Poh09]).


In der impliziten räumlichen Anordnung der Voxel in ihrer Datenstruktur steckt allerdings auch ein großer Nachteil. Es ist erheblich aufwändig, 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]).
1.3.3 Medizinische Volumendaten


\(^{17}\)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


Gemeinfreies Bild aus: http://commons.wikimedia.org/wiki/File:Dürer-_Man_Drawing_a_Lute.jpg
2.1 Raytracing

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.


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

Abbildung 3: Schematisches Konzept des Raytracing\textsuperscript{19}

\textsuperscript{19}Enthält eine gemeinfreie Grafik (Glühbirne) aus: http://www.openclipart.org

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:

\[ \overrightarrow{r} = \overrightarrow{o} + \lambda \cdot \overrightarrow{d} \] (1)

\( \overrightarrow{o} \) ist der Ortsvektor (der Vektor zum optischen Zentrum) und \( \overrightarrow{d} \) 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.


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).
2.1 Raytracing


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.


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“.


Ein für das interaktive Rendering nicht zu unterschätzender Nachteil des 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


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:

\[ d^e = \frac{d^{e-1}}{2} \]  

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


Abbildung 5 zeigt die Unterteilung eines Raumes durch den nebenstehenden abgebildeten Octree.

**Abbildung 5:** Ein beispielhafter Octree und die dazugehörige Raumenteilterung
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 angeordnet 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 Definition 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


- SISD entspricht dem Normalfall, bei dem ein Prozessor einen Befehl auf einen Datensatz anwendet.
2.4 SIMD und SEE

Abbildung 7: Ein Beispielhafter Quadtree und die dazugehörige Raumnuteilung

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>

Tabelle 2: Die Flynn’sche 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 Befehlssatzweiterung 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
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.


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

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
2.5.2 Arbeitsteilung


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:


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 nicht sinnvoll. Ferner würde so jede Möglichkeit genommen, die

\[20\text{Jede Bounding Box im Szenengraphen von Augenblick ist } 32\ \text{Byte groß. Kapitel 3.4.4 geht näher darauf ein.}\]
Schnitttests auf die Rahmenbedingungen des Voxel-Octree (achsenparallele, mittige Raumunterteilung) hin zu optimieren.

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\textsuperscript{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\textsuperscript{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

\textsuperscript{21}Es ist anzunehmen, dass vollständige Verarbeitungsketten in den Entwicklerstudios von id Software und co. existieren.

\textsuperscript{22}Die Verwendung von DICOM-Datensätzen kam, wie in Kapitel 1.3.3 erläutert, nicht in Frage.
3.1 Das Konverterprogramm

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/plutoObj/plutoObj.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


29Enthält eine Grafik aus: http://commons.wikimedia.org/wiki/File:RGB_color_solid_cube.png, Lizenz: Creative Commons Attribution ShareAlike 3.0
### 3.1 Das Konverterprogramm

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.


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 wichtiger 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.


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 dem 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.


Zum Zweiten werden, um die negativen Auswirkungen auf die Darstellungsgüte 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

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


\[
x^m = \frac{x^1 \cdot v^1 + x^2 \cdot v^2}{v^1 + v^2}
\]

(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 Darstellungsschärfe 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.


Abbildung 11 zeigt das Schema der 3D-Scan-Konvertierung. Dargestellt ist ein Dreieck mit den Eckpunkten $V_0$, $V_1$ und $V_2$. $\vec{p}$ 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 $B$. $\text{of}\vec{s}P$ ist der Vektor vom Ursprung der Hypotenuse ($V_0$) zum Basispunkt $B$, $\text{of}\vec{s}Q$ 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 $\text{of}\vec{s}P$ (und somit $B$) entlang der Hypotenuse $\vec{p}$. $B$ bildet den Ausgangspunkt für die innere Schleife, die $\text{of}\vec{s}Q$ (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. \( L_1 \) und \( L_2 \) sind die Schnittpunkte der Geraden entlang \( \vec{q} \) mit \( \vec{s} \) und \( \vec{t} \). Die Lambda-Werte zu den Schnittpunkten \( L_1 \) und \( L_2 \) sind das Ergebnis der Schnittpunktberechnungen, der Kleinere von beiden gibt an ab welcher Länge \( of\vec{f}Q \) 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 \( of\vec{f}P \) 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:

\[
\begin{pmatrix}
0.5 \\
0.3 \\
0.1
\end{pmatrix} \div 0.5 = \begin{pmatrix}
1 \\
0.6 \\
0.2
\end{pmatrix}
\] (4)


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...
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).


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

### 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.


Zum Zweiten ist eine Komprimierung überflüssiger Details ein

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


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 gemeinsam ist die Dateiendung „.vox“. 

<table>
<thead>
<tr>
<th>Header</th>
<th>Voxel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic Number</td>
<td>Version</td>
</tr>
<tr>
<td>4D 42</td>
<td>04 00</td>
</tr>
<tr>
<td>05 00</td>
<td>A1 76</td>
</tr>
<tr>
<td>1C AA</td>
<td>02 00</td>
</tr>
<tr>
<td>04 00</td>
<td>F9 C5</td>
</tr>
</tbody>
</table>

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

Abbildung 17 zeigt eine simple Beispielseite 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.
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“.

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 $8 + 3 \times Palettengrösse$ 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 (de-serialisiert) 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.

Die Magic Number 0x4D42 wird von keinem dort aufgeführten Format belegt.
3.2 Das Voxel-Dateiformat


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.

\textsuperscript{34}Physisch ist der RAM zweidimensional, da die Bytes auf den Chips über Zeilen- und Spaltennummern adressiert werden

\textsuperscript{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 einem 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
3.2 Das Voxel-Dateiformat
3.3 Das Voxel-Loader-Plugin


```c
void deserialize(Voxel* voxel, File* fileStream) {
    // get the palette index and normal
    fread(&((voxel)->paletteIndex), 2, 1, fileStream);
    fread(&((voxel)->theta), 1, 1, fileStream);
    fread(&((voxel)->phi), 1, 1, fileStream);

    // if the palette index is uneven the voxel has children
    if (((voxel)->paletteIndex)%2) {
        voxel->children=(Voxel*)malloc(8*voxelSize);
        for (int i=0; i<8; i++) {
            deserialize(&((voxel)->children[i]), fileStream);
        }
    } else {
        voxel->children=NULL;
    }

    // shift the palette by one bit to the right
    voxel->paletteIndex>>=1;
}
```

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

Die Methode deserialize() in Listing 1 erwartet als Eingabeparameter


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.

![Uniform Grid](image)

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


Der große Nachteil des Uniform Grid ist sein Speicherverbrauch. Mit steigender Auflösung wächst der Speicherverbrauch exponentiell. Für einen Raum mit einer Auflösung von 1024 Voxel 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. Das Uniform Grid unternimmt nichts um diese Gleichförmigkeit zu komprimieren, in Bezug auf den Speicherbedarf 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


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 
\[
\frac{((256 \times 4) - (85 \times 8)) \times 100}{256 \times 4} = 33,59\%.
\]

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

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.

<table>
<thead>
<tr>
<th>Modell</th>
<th>Tiefe</th>
<th>Auflösung</th>
<th>Voxel</th>
<th>Speicher</th>
<th>Ersparnis</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>13</td>
<td>4096^3</td>
<td>131.827.905</td>
<td>0,98 Gb</td>
<td>260x</td>
</tr>
<tr>
<td>Bunny</td>
<td>13</td>
<td>4096^3</td>
<td>69.798.129</td>
<td>532,5 Mb</td>
<td>492x</td>
</tr>
<tr>
<td>Porsche</td>
<td>11</td>
<td>1024^3</td>
<td>4.504.929</td>
<td>34,3 Mb</td>
<td>119x</td>
</tr>
<tr>
<td>Porsche</td>
<td>12</td>
<td>2048^3</td>
<td>18.095.137</td>
<td>138 Mb</td>
<td>237x</td>
</tr>
<tr>
<td>Porsche</td>
<td>13</td>
<td>4096^3</td>
<td>72.457.401</td>
<td>552,8 Mb</td>
<td>474x</td>
</tr>
<tr>
<td>Soccerball</td>
<td>12</td>
<td>2048^3</td>
<td>48.210.017</td>
<td>367,8 Mb</td>
<td>89x</td>
</tr>
<tr>
<td>World</td>
<td>14</td>
<td>8192^3</td>
<td>85.519.385</td>
<td>652,4 Mb</td>
<td>3214x</td>
</tr>
</tbody>
</table>

Tabelle 3: Speicherverbrauch und Kompressionsfaktor verschiedener Modelle im Octree

Die der Tabelle zugrunde gelegte Formel lautet \( f = \frac{a^3}{16} \), 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
3.4 Die Octree-Datenstruktur

<table>
<thead>
<tr>
<th>Ebenen</th>
<th>Auflösung</th>
<th>Blätter</th>
<th>Knoten</th>
<th>Total</th>
<th>Speicher</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$1^3$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8 Byte</td>
</tr>
<tr>
<td>2</td>
<td>$2^3$</td>
<td>8</td>
<td>1</td>
<td>9</td>
<td>72 Byte</td>
</tr>
<tr>
<td>3</td>
<td>$4^3$</td>
<td>64</td>
<td>9</td>
<td>73</td>
<td>584 Byte</td>
</tr>
<tr>
<td>4</td>
<td>$8^3$</td>
<td>512</td>
<td>73</td>
<td>585</td>
<td>4.57 Kb</td>
</tr>
<tr>
<td>5</td>
<td>$16^3$</td>
<td>4.096</td>
<td>585</td>
<td>4.681</td>
<td>36.57 Kb</td>
</tr>
<tr>
<td>6</td>
<td>$32^3$</td>
<td>32.708</td>
<td>4.681</td>
<td>37.449</td>
<td>292.57 Kb</td>
</tr>
<tr>
<td>7</td>
<td>$64^3$</td>
<td>262.144</td>
<td>37.449</td>
<td>299.593</td>
<td>2.28 Mb</td>
</tr>
<tr>
<td>8</td>
<td>$128^3$</td>
<td>2.097.152</td>
<td>299.593</td>
<td>2.396.745</td>
<td>18.28 Mb</td>
</tr>
<tr>
<td>9</td>
<td>$256^3$</td>
<td>16.777.216</td>
<td>2.396.745</td>
<td>19.173.961</td>
<td>146.28 Mb</td>
</tr>
<tr>
<td>10</td>
<td>$512^3$</td>
<td>1.34 * 10$^8$</td>
<td>1.91 * 10$^7$</td>
<td>1.53 * 10$^8$</td>
<td>1.14 Gb</td>
</tr>
<tr>
<td>11</td>
<td>$1024^3$</td>
<td>1.07 * 10$^9$</td>
<td>1.53 * 10$^8$</td>
<td>1.22 * 10$^9$</td>
<td>9.14 Gb</td>
</tr>
<tr>
<td>12</td>
<td>$2048^3$</td>
<td>8.55 * 10$^9$</td>
<td>1.22 * 10$^9$</td>
<td>9.81 * 10$^9$</td>
<td>73.14 Gb</td>
</tr>
<tr>
<td>13</td>
<td>$4096^3$</td>
<td>6.87 * 10$^{10}$</td>
<td>9.81 * 10$^9$</td>
<td>7.85 * 10$^{10}$</td>
<td>585.14 Gb</td>
</tr>
<tr>
<td>14</td>
<td>$8192^3$</td>
<td>5.49 * 10$^{11}$</td>
<td>7.85 * 10$^{10}$</td>
<td>6.28 * 10$^{11}$</td>
<td>4.57 Tb</td>
</tr>
</tbody>
</table>

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.

acht Vaterelementen wiederum ein Vaterelement usw.

\[ \sum_{n=1}^{\infty} \frac{1}{8^n} = \frac{1}{7} \]  

(5)


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 zugreifen 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,
3.4 Die Octree-Datenstruktur
dennoch ist es um einiges langsamer bis zu 13 Voxelknoten zu besuchen bevor man auf das gesuchte Voxelblatt zugreifen kann als ein direkter Zu-
griff. 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 Charakte-
ristika mit nicht-uniformen Octrees teilt. In kd-Trees wird pro Knoten eine
Ebene definiert die den Raum an beliebige Stelle in zwei Hälfte 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 Ver-
hältnis 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
Speicherbedarf 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 Hie-
rarchien sind, sind BVHs objektunterteilend. Eine BVH kann beispielswei-
se „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 Spei-
cherung 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 redun-
dant.
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 Voxel-Daten-sätze 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.


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 allozieren.

3.4 Die Octree-Datenstruktur

Abbildung 21: Naive und optimierte Kodierung des Octree im Vergleich


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 erkauf 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. Daher muss man, um

37Einzelne 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.
38Mehrere Gigabyte an Arbeitspeicher an einem Stück zu allocieren erweist sich, wie bereits in Kapitel 3.2.2 erwähnt, als sehr problematisch.
39Die 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.


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.


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-


\footnote{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.}
Die Verarbeitungskette 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 Farbpaletteindex (zur Motivation dahinter siehe Kapitel 3.1.2) und zwei Byte für die Normale des Voxels zusammen. Von den zwei Byte für den Farbpaletteindex muss ein Bit leider ungenutzt bleiben, da die Farbpalette im „.vox“-Dateiformat nur einen Umfang von 15 Bit hat.


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.

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 verschiedene 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 Normal 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 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

**Abbildung 22:** Gegenüberstellung von zur Laufzeit generierten Normalen auf den Seitenflächen der Voxel und den aus Eckpunktnormalen interpolierten Normalen
die Kantigkeit der Modelle stark reduziert\textsuperscript{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 verfüg-
baren 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 Kugelko-
ordinaten gespeichert, dabei werden die Winkel $\theta$ und $\phi$ mit jeweils acht
Bit kodiert. Üblicherweise werden Normalen in kartesischen Koordinaten
in einfacher Gleitkommagenaugigkeit ($3 \times 32$ 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 abzuspe-
ichern. 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 wer-
den, 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 Repräsen-
tation 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

\textsuperscript{41}Mit den zur Laufzeit generierten Normalen entsteht ein „LEGO-Look“, welcher durch-
chaus einen gewissen Reiz besitzt, dem Realismus aber nicht dienlich ist.
3.4 Die Octree-Datenstruktur

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 durch 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.

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.


Abbildung 23 illustriert das grundlegende Konzept des Raytracers. Die vom Algorithmus berechneten Schnittpunkte sind von eins bis sechs
3.5 Der Voxel-Raytracer

Abbildung 23: Die Vorgehensweise des Raytracers beim Durchqueren eines Quadtree

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


3.5 Der Voxel-Raytracer

Abbildung 25: Schnitt- und Versatzpunkte eines Strahls.


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:

\[ Ax + By + Cz + D = 0 \] (6)

und der Geradengleichung in der Form:

\[
\vec{x} = \begin{pmatrix} x_o \\ y_o \\ z_o \end{pmatrix} + \lambda \cdot \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix}
\] (7)
folgt durch Einsetzen, dass
\[ \lambda = \frac{-(Ax_o + By_o + Cz_o + D)}{Ax_d + By_d + Cz_d} \]  
(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:
\[ \lambda = \frac{-D - x_o}{x_d} \]  
(9)

für Seitenflächen die parallel zur YZ-Ebene liegen,
\[ \lambda = \frac{-D - y_o}{y_d} \]  
(10)

für Seitenflächen die parallel zur XZ-Ebene liegen und
\[ \lambda = \frac{-D - z_o}{z_d} \]  
(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 Schnittpunktberechnungen 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 Voxeln 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 Paletteinindex 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:

\[
\begin{pmatrix}
  x \\
  y \\
  z
\end{pmatrix} = \begin{pmatrix}
  \sin \theta \cdot \cos \phi \\
  \sin \theta \cdot \sin \phi \\
  \cos \theta
\end{pmatrix}
\]


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.


- 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 \( 2 \times 6 = 12 \) 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.


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.


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.


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

```c
// make sure the X component of the direction vector does not contain zero
// branch variant, single ray only
if (dir[0]==0) dir[0]=Augenblick->epsilon();

// mask select variant, four rays in parallel and no branch required
dir[0] = dir[0].select(Float4::epsilon(), (dir[0]==Float4::zero()));
```

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


Bei der Umstellung der Implementation auf SIMD konnten durch den Einsatz von SIMD-Masken und den select()-Befehl die meisten Verzweigungen im Programmfuss 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.


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

```cpp
index = Int4::zero();
bisectors = vMin+(vMax-vMin)*Float4::half();

position = (p[0]>=bisectors[0]);
index = index.select(Int4::one(),position);
vMin[0] = vMin[0].select(bisectors[0],position);
vMax[0] = vMax[0].select(bisectors[0],position);

position = (p[1]>=bisectors[1]);
index = index.select(index+two,position);
vMin[1] = vMin[1].select(bisectors[1],position);
vMax[1] = vMax[1].select(bisectors[1],position);

position = (p[2]>=bisectors[2]);
index = index.select(index+four,position);
vMin[2] = vMin[2].select(bisectors[2],position);
vMax[2] = vMax[2].select(bisectors[2],position);

// descend into child voxels for those points that are still flagged descend
((descend&&first).testAny())?v1=&(v1-c[index[0]]):v1=v1;
((descend&&second).testAny())?v2=&(v2-c[index[1]]):v2=v2;
((descend&&third).testAny())?v3=&(v3-c[index[2]]):v3=v3;
((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 Raytrakers.

In diesem Codeabschnitt geschehen zweierlei Dinge. Zum Einen werden für vier Voxel die Indizes der Kinder bestimmt in die abgestiegen werden soll und zum Zweitlen 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...


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 1024³, 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


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
Voxel bei der Suche im Baum zu reduzieren. 
Beı 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 roter die Pfade dar, entlang derer die Nachbarschaftssuche für einen beispielhaften Strahl den Baum traver-
3.5 Der Voxel-Raytracer

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.\footnote{Beispielsweise befinden sich zwei Voxel die sich an der Grenze des linken und rechten Halbraums des Gesamtvolumens berühren in zwei verschiedenen Teilbäumen der Wurzel.}


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. 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 Octrees mit der eines Uniform Grid vereint. Der Octree bleibt dabei unverän-

---

48Sie würde sich mindestens halbieren, da nicht alle Suchläufe erst auf der untersten Ebene des Baumes enden würden.
3.5 Der Voxel-Raytracer
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 il-
lustriert, dass die Speicherposition eines Voxels im Uniform Grid direkt aus
seiner Raumposition hergeleitet werden kann. Der große Nachteil des Uni-
form Grid ist sein explodierender Speicherverbrauch bei hohen Auflösun-
gen. 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 Betriebssys-
tem 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 zusätzlichen 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 zusätzlich-
er Speicherverbrauch und der daraus resultierende Geschwindigkeitszu-
wachs frei wählbar sind. Abhängig davon wie viel zusätzlichen Speicher
man bereit ist für das Uniform Grid auszugeben, kann die Geschwindigkeit
gesteigert werden.

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

Abbildung 31: Pfade in einer hybriden Datenstruktur


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

**Abbildung 32:** Hervorhebung der Strahlen mit den meisten inneren Schleifen-durchläufen 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-


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

4.1.1 Geschwindigkeit des Raytracing


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
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 verschiedene-
4.1 Ergebnisse

<table>
<thead>
<tr>
<th>Modell</th>
<th>Auflösung</th>
<th>Voxel</th>
<th>Bildrate MPRO</th>
<th>Bildrate MBP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Al</td>
<td>2048³</td>
<td>32,8 Mio.</td>
<td>12,6</td>
<td>2,1</td>
</tr>
<tr>
<td>Al</td>
<td>4096³</td>
<td>131,8 Mio.</td>
<td>10,8</td>
<td>1,8</td>
</tr>
<tr>
<td>Bunny</td>
<td>2048³</td>
<td>17,4 Mio.</td>
<td>13,3</td>
<td>2,2</td>
</tr>
<tr>
<td>Bunny</td>
<td>4096³</td>
<td>69,7 Mio.</td>
<td>11,0</td>
<td>1,9</td>
</tr>
<tr>
<td>Porsche</td>
<td>4096³</td>
<td>72,4 Mio.</td>
<td>12,4</td>
<td>2,1</td>
</tr>
<tr>
<td>Soccerball</td>
<td>2048³</td>
<td>48,2 Mio.</td>
<td>8,5</td>
<td>1,4</td>
</tr>
<tr>
<td>Teapot</td>
<td>2048³</td>
<td>18,4 Mio.</td>
<td>12,5</td>
<td>2,1</td>
</tr>
<tr>
<td>World</td>
<td>1024³</td>
<td>1,3 Mio.</td>
<td>16,8</td>
<td>2,9</td>
</tr>
<tr>
<td>World</td>
<td>2048³</td>
<td>5,4 Mio.</td>
<td>15,5</td>
<td>2,7</td>
</tr>
<tr>
<td>World</td>
<td>4096³</td>
<td>21,4 Mio.</td>
<td>14,2</td>
<td>2,5</td>
</tr>
<tr>
<td>World</td>
<td>8192³</td>
<td>85,5 Mio.</td>
<td>13,0</td>
<td>2,3</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Auflösung</th>
<th>Al</th>
<th>Bunny</th>
<th>Porsche</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>1³</td>
<td>6,6</td>
<td>6,5</td>
<td>6,7</td>
<td>6,5</td>
</tr>
<tr>
<td>2³</td>
<td>5,8</td>
<td>5,8</td>
<td>5,8</td>
<td>5,9</td>
</tr>
<tr>
<td>4³</td>
<td>5,2</td>
<td>5,4</td>
<td>5,0</td>
<td>5,0</td>
</tr>
<tr>
<td>8³</td>
<td>4,9</td>
<td>4,5</td>
<td>4,8</td>
<td>4,7</td>
</tr>
<tr>
<td>16³</td>
<td>4,4</td>
<td>4,1</td>
<td>4,5</td>
<td>4,2</td>
</tr>
<tr>
<td>32³</td>
<td>4,0</td>
<td>3,8</td>
<td>4,2</td>
<td>3,9</td>
</tr>
<tr>
<td>64³</td>
<td>3,7</td>
<td>3,6</td>
<td>3,9</td>
<td>3,7</td>
</tr>
<tr>
<td>128³</td>
<td>3,4</td>
<td>3,3</td>
<td>3,6</td>
<td>3,5</td>
</tr>
<tr>
<td>256³</td>
<td>3,1</td>
<td>3,0</td>
<td>3,3</td>
<td>3,2</td>
</tr>
<tr>
<td>512³</td>
<td>2,7</td>
<td>2,8</td>
<td>3,0</td>
<td>2,9</td>
</tr>
<tr>
<td>1024³</td>
<td>2,4</td>
<td>2,5</td>
<td>2,7</td>
<td>2,7</td>
</tr>
<tr>
<td>2048³</td>
<td>2,1</td>
<td>2,2</td>
<td>2,4</td>
<td>2,5</td>
</tr>
<tr>
<td>4096³</td>
<td>1,8</td>
<td>1,9</td>
<td>2,1</td>
<td>2,3</td>
</tr>
</tbody>
</table>

Tabelle 7: Bildwiederholraten ausgewählter Modelle und Auflösungen auf Testsystem MBP

<table>
<thead>
<tr>
<th>Modell (Auflösung überall 1024³)</th>
<th>Al</th>
<th>Bunny</th>
<th>Porsche</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leerlauf (leerer Methodenaufruf)</td>
<td>10,0</td>
<td>9,8</td>
<td>9,9</td>
<td>10,1</td>
</tr>
<tr>
<td>Mit Custom Execution Chain</td>
<td>7,7</td>
<td>7,6</td>
<td>7,5</td>
<td>7,7</td>
</tr>
<tr>
<td>Mit Vorverarbeitung der Strahlen</td>
<td>7,1</td>
<td>7,0</td>
<td>7,2</td>
<td>7,1</td>
</tr>
<tr>
<td>Level Of Detail auf Ebene 1</td>
<td>6,6</td>
<td>6,5</td>
<td>6,7</td>
<td>6,5</td>
</tr>
<tr>
<td>Finale Version</td>
<td>2,4</td>
<td>2,5</td>
<td>2,7</td>
<td>2,7</td>
</tr>
<tr>
<td>Einzelstrahl-Implementation</td>
<td>0,8</td>
<td>0,9</td>
<td>0,9</td>
<td>0,9</td>
</tr>
</tbody>
</table>

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

einer Auflösung von 64³ und einem zusätzlichen Speicherverbrauch von 1 Mb.
4.1 Ergebnisse

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


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 Voxel 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 Voxelmenge, logarithmisch skaliert

- 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, die Speicherverbrauch mit nur einem Bit pro Voxel für die Datenstruktur so kompakt wie möglich ist.


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 Voxel ($1 + 8 + 64$). 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
4.1 Ergebnisse

<table>
<thead>
<tr>
<th>Modell (Auflösung überall 1024³)</th>
<th>Al</th>
<th>Bunny</th>
<th>Porsche</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kein Uniform Grid</td>
<td>1,6</td>
<td>1,6</td>
<td>1,8</td>
<td>1,9</td>
</tr>
<tr>
<td>Uniform Grid 1³, 4 Byte</td>
<td>1,4</td>
<td>1,5</td>
<td>1,7</td>
<td>1,8</td>
</tr>
<tr>
<td>Uniform Grid 2³, 32 Byte</td>
<td>1,6</td>
<td>1,7</td>
<td>2,1</td>
<td>2,3</td>
</tr>
<tr>
<td>Uniform Grid 4³, 256 Byte</td>
<td>1,8</td>
<td>1,9</td>
<td>2,1</td>
<td>2,3</td>
</tr>
<tr>
<td>Uniform Grid 8³, 2 Kb</td>
<td>2,0</td>
<td>2,1</td>
<td>2,3</td>
<td>2,5</td>
</tr>
<tr>
<td>Uniform Grid 16³, 16 Kb</td>
<td>2,2</td>
<td>2,3</td>
<td>2,5</td>
<td>2,7</td>
</tr>
<tr>
<td>Uniform Grid 32³, 128 Kb</td>
<td>2,3</td>
<td>2,4</td>
<td>2,6</td>
<td>2,8</td>
</tr>
<tr>
<td>Uniform Grid 64³, 1 Mb</td>
<td>2,4</td>
<td>2,5</td>
<td>2,7</td>
<td>2,7</td>
</tr>
<tr>
<td>Uniform Grid 128³, 8 Mb</td>
<td>2,3</td>
<td>2,4</td>
<td>2,6</td>
<td>2,6</td>
</tr>
<tr>
<td>Uniform Grid 256³, 64 Mb</td>
<td>2,0</td>
<td>2,0</td>
<td>2,3</td>
<td>2,2</td>
</tr>
</tbody>
</table>

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 2048³ 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 Octrees von 20 (das entspricht einer Auflösung von 524288³) 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
4.1.3 Darstellungsqualität


<table>
<thead>
<tr>
<th>Auflösung</th>
<th>Al</th>
<th>Bunny</th>
<th>Porsche</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1^3$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$2^3$</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>$4^3$</td>
<td>73</td>
<td>73</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>$8^3$</td>
<td>265</td>
<td>329</td>
<td>201</td>
<td>193</td>
</tr>
<tr>
<td>$16^3$</td>
<td>1.265</td>
<td>1.129</td>
<td>841</td>
<td>657</td>
</tr>
<tr>
<td>$32^3$</td>
<td>5.473</td>
<td>4.553</td>
<td>3.937</td>
<td>2.217</td>
</tr>
<tr>
<td>$64^3$</td>
<td>128.57</td>
<td>17.825</td>
<td>16.553</td>
<td>7.489</td>
</tr>
<tr>
<td>$128^3$</td>
<td>110.257</td>
<td>70.377</td>
<td>68.633</td>
<td>26.361</td>
</tr>
<tr>
<td>$256^3$</td>
<td>474.561</td>
<td>275.697</td>
<td>277.113</td>
<td>95.145</td>
</tr>
<tr>
<td>$512^3$</td>
<td>1.979.673</td>
<td>1.097.625</td>
<td>1.116.641</td>
<td>356.305</td>
</tr>
<tr>
<td>$2048^3$</td>
<td>32.803.289</td>
<td>17.467.009</td>
<td>18.095.137</td>
<td>5.415.561</td>
</tr>
<tr>
<td>$4096^3$</td>
<td>131.827.905</td>
<td>69.798.129</td>
<td>72.457.401</td>
<td>21.471.993</td>
</tr>
</tbody>
</table>

Tabelle 10: Umfangssteigerung in Voxeln ausgewählter Modelle mit zunehmender Auflösung
4.1 Ergebnisse

Abbildung 36: Darstellung der Umfangssteigerung von Modellen zwischen den Auflösungen


Kritische Punkte, welche die erzielbare Darstellungsqualität von Volumendaten mindern, sind die im nachfolgenden Kapitel 4.2 aufgeführten Treppeffekte und Selbstverschattungen.
4.2 Probleme


4.2.1 Architekturbedingte Probleme

![Abbildung 37: Fehlerhafte Abtastung der Voxel die der Sehstrahl durchquert](image)

Ein zentrales, während der Entwicklung dieses Raytracers immer wieder auftauchendes Problem war die limitierte Gleitkommmagenauigkeit. Alle mathematischen Operationen sind auf Gleitkomma-
4.2 Probleme

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 Vox-
el, welche eine infinitesimal dünne Oberfläche (blaue Kurve) annähern. In dieser Abbildung wird ein Strahl durch drei Punkte abgetastet, zu de-

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

Durch Anpassungen am Algorithmus, welche bestimmte Abbruchkrite-
rien flexibler auslegen und Justierungen an Wertebereichen des Voxel-
Geometrieobjekts konnten diese Darstellungsfehler weitestgehend elimi-
niert werden. Allerdings verkomplizieren sich durch die Fehlertoleranz die Abbruchbedingungen. Eine Rundung der Gleitkommapperzen 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 Genauigkeit (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.

4.2 Probleme


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“


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 4096³ 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

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, für interaktives Raytracing kommt es jedoch nicht in Frage. Abbildung 44 auf Seite 104 eignet 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,


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 einem vier Byte großen Zeiger und vier Byte Nutzdaten zusammen. In dieser Anordnung der Daten gibt es noch einiges Potential den Speicherverbrauch weiter zu reduzieren.


4.3.3 Steigerung der Darstellungsqualität


Schließlich ließe sich die Darstellungsqualität noch weiter steigern
4.3 Ausblick


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]).


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 Voxel 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.
4.3 Ausblick

Da das Animationsproblem vorerst ungelöst bleibt ist klar, dass Volumendaten mittelfristig die Rasterisierung oder das Raytracing von Polygo- 
b lem. 
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 ................................. 14
3. Schematisches Konzept des Raytracing .......................................... 15
4. Illustration der Instruktionsparalellität und Datenkohärenz ............. 18
5. Ein beispielhafter Octree und die dazugehörige Raumentteilung .......... 19
6. Anordnung der Kinder in einem Octree .......................................... 20
7. Ein beispielhafter Quadtree und die dazugehörige Raumentteilung ........ 21
8. Die Einzelprogramme und der Datenfluss zwischen diesen ............... 27
9. Reduktion eines Farbraums auf eine Palette ................................. 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 orthogonalen Vektoren ............................................. 34
13. Diskretisierung einer Dreiecksfläche in Voxel im 3D-ScanKonverter-Verfahren ................................................................. 35
14. Bedeutung der Winkel Theta und Phi im Kugelkoordinatensystem ...... 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 Voxel .......... 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 Schleifen durchläufen 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 Voxelmenge, logarithmischt 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 Flynnische 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


