Speicher- und Kompressionsverfahren für Volumenvisualisierungshardware

Dissertation
der Fakultät für Informations- und Kognitionswissenschaften
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Phys. Gregor Wetekam
aus Iserlohn

Tübingen
2005

Tag der mündlichen Qualifikation:	29.06.2005
Dekan:	Prof. Dr. Michael Diehl
1. Berichterstatter:	Prof. Dr. Wolfgang Straßer
2. Berichterstatter:	Prof. Dr. Wolfgang Rosenstiel

Danksagung:

Hiermit möchte ich mich bei Prof. Straßer für die freundliche Aufnahme in seine Arbeitsgruppe und die fortdauernde Unterstützung dieser Arbeit bedanken.
Mein herzlicher Dank gilt auch der VIZARD II-Mannschaft, Mike, Michael, Urs, Johannes und Alexander, die mir durch ihre außergewöhnlich freundschaftliche Zusammenarbeit das Teilhaben an einem hervorragenden Projekt ermöglichten.
Besondere Erwähnung gebührt auch Dr. Michael Wand, der mir stets mit Rat und Tat zur Seite stand und nicht müde wurde wertvolle Hinweise zu dieser Arbeit zu leisten.
Weiter danke ich allen meinen anderen Kollegen und Studenten, die an dieser Forschungsarbeit beteiligt waren und meiner Familie, die mich immer unterstützte.
Abschließend gilt mein besonderer Dank allen Bürgern der Europäischen Union, die ein Großteil dieser wissenschaftlichen Arbeit über das Projekt „DynCT“ der EU-Kommission finanzierten.

Kurzfassung

In der Computergraphik hat sich die Volumenvisualisierung als wertvolle Technik etabliert. Besonders im Bereich der medizinischen Visualisierung, wissenschaftlichen Simulation und in der Geophysik konnte sich dieses Verfahren zur Darstellung volumetrischer Datensätze durchsetzen. Den stetig wachsenden Anforderungen in Bezug auf Datenmenge, Darstellungsgeschwindigkeit und Interaktivität, wurde durch die Verwendung spezieller Volumenvisualisierungshardware Rechnung getragen. Hierbei werden die zur Verfügung stehenden Ressourcen bestmöglich an den Visualisierungsalgorithmus angepasst. Die dominante Herausforderung der Volumenvisualisierung bleibt dennoch die sehr großen Datenmenge, die zu schwerwiegenden Problemen in Bezug auf Speicherplatz und -bandbreite führt. Der bisherige Ansatz bestehender Visualisierungshardware resultiert in einem linearen Zusammenhang zwischen Datensatzgröße und Speicherbedarf. Eine Methode die Komplexität sowohl hinsichtlich Speicherbedarf als auch Rechenzeit auf $O(\log n)$ zu reduzieren, besteht in der Verwendung eines Multiskalenmodells.

Diese Arbeit stellt erstmals die Integration eines solchen waveletbasierten Multiskalenmodells in eine Hardwarearchitektur zur Volumenvisualisierung vor. Dies erlaubt Datensätze darzustellen, die um eine Größenordnung umfangreicher sind als bisher. Das Multiskalenmodell erfordert jedoch eine dynamische Speicherverwaltung der Volumendaten im lokalen Speicher der Visualisierungshardware. Um dieser Anforderung gerecht zu werden, wird durch eine neuartige cachebasierte Speicherschnittstelle, genannt VoxelCache, die notwendige Virtualisierung der Speicherverwaltung durchgeführt. Der VoxelCache führt darüber hinaus durch seinen on-chip Cache zu einer signifikanten Reduzierung der erforderlichen Speicherbandbreite zwischen der Visualisierungspipeline und ihrem lokalen Speicher.

Um auch die Bandbreite zwischen Visualisierungshardware und Host-Computer zu reduzieren, wird die Decodierung der komprimierten Waveletkoeffizienten des Multiskalenmodells in Hardware durchgeführt. Somit kann der Datendurchsatz der Dekompression auf ein Vielfaches, gegenüber bestehenden softwarebasierten Dekompressionsimplementierungen, gesteigert werden. Der Host-Computer wird damit von allen rechenintensiven Aufgaben befreit und ist nur noch für die Verwaltung des Multiskalenmodells verantwortlich.

Da die aufwendige Vorverarbeitung des Multiskalenmodells nicht in allen Anwendungszenarien möglich ist, wird für in Echtzeit generierte dynamische Volumendaten ein Kompressionsschema vorgestellt, dass sich durch geringe Ressourcenanforderungen hinsichtlich der Kompression auszeichnet und kaum zusätzliche Latenz in den Zeitraum zwischen Aufnahme und Visualisierung der Daten einführt. Dies ist von entscheidender Bedeutung bei der Echtzeitvisualisierung von Volumendaten, wie sie zum Beispiel bei intraoperativen Anwendungen in der Medizin eingesetzt werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Visualisierung von volumetrischen Datensätzen hat in der Computergraphik den Sprung von der theoretischen Betrachtung in die alltägliche Praxis seit langem vollzogen. In der Medizin, der wissenschaftlichen Simulation, der Materialforschung, der Geologie und der Meteorologie hat sich diese Technik etabliert und ist ein wertvolles Werkzeug in der Betrachtung und Untersuchung von internen Strukturen. Hierbei steht nicht die ästhetische Darstellung der Daten im Vordergrund, sondern die exakte und unverfälschte Wiedergabe der im Datensatz enthaltenen Informationen.

Volumetrische Datensätze entstehen durch Berechnung, Abtastung oder Modellierung der Eigenschaften eines Objektes auf beliebigen dreidimensionalen Raumpunkten. Dabei werden diskrete Werte meist, aber nicht zwangsläufig, auf ein reguläres Gitter abgebildet. Im Unterschied zur polygonalbasierten Graphik können so, neben der Hülle, auch das Innere eines Objekts erfasst werden. Hat sich die Computergraphik in diesem Bereich vor wenigen Jahren noch auf das Finden neuer Verfahren zur Visualisierung konzentriert, so wurden inzwischen Algorithmen und Techniken entwickelt, die die Hauptanforderungen an die Visualisierung, den Erkenntnisgewinn, zufrieden stellend erfüllen. Gleichzeitig entstand durch die kontinuierliche Verbesserung der Geräte und der Verfahren zur Datensatzgenerierung eine neue Problematik. Heutige medizinische Scanner und Simulationen erzeugen eine äußerst große Menge an Daten, die im Bereich von mehreren Gigabyte liegen. Die Bewältigung der enormen Datenmengen bezüglich Speicherplatz und -bandbreite stellt somit heutzutage die Hauptherausforderung der Volumenvisualisierung dar.

Eine Schlüsselbeobachtung bei der Bewältigung dieser Aufgabe ist, dass ein regelmäßig abgetastetes Gitter von diskreten Werten eine Form der Repräsentation ist, die hohe Redundanz beinhaltet. Benachbarte Daten in einem Datensatz sind meist stark korreliert und eignen sich somit gut für Datenkompressionsverfahren. Eine Methode, die sich hierbei als besonders effizient erwiesen hat, ist die Waveletkompression. Durch die Transformation in eine hierarchische Funktionsbasis werden die Daten zunächst dekorreliert und in einem zweiten Schritt über einen Redundanzdecoder komprimiert. Bei vollständig verlustfreier Kompression kann die Datenmenge typischerweise um den Faktor 2–4 reduziert werden. Über eine Quantisierung der Waveletkoeffizienten kann zusätzlich eine verlustbehaftete Kompression

vorgenommen werden. Hierbei sind Kompressionsraten mit einem Faktor von 20–50 erreichbar, ohne größere sichtbare Artefakte in den Bildern zu verursachen [31]. Für Anwendungen, bei denen diese minimalen Artefakte akzeptable sind, kann somit die Datenmenge, die für die Visualisierung benötigt wird, drastisch reduziert werden. Für andere Bereiche in denen ein Verlust an Information nicht tolerierbar ist, wie die Medizin, kann über die verlustfreie Kompression trotzdem noch eine signifikante Einsparung hinsichtlich des Speicherplatzbedarfs erzielt werden.

Neben der Anforderung an den Speicher, stellt die Volumenvisualisierung sehr hohe Ansprüche im Bezug auf Rechenleistung. Als Grundlage dieser Arbeit wird daher eine eigens entwickelte Visualisierungshardware genutzt, die speziell an die Erfordernisse des Darstellungsalgorithmus angepasst ist und eine hochoptimierte Speicherschnittstelle besitzt. Die Verwendung einer dedizierten Hardwareimplementierung für die Visualisierung erlaubt die Integration der Waveletdekompression auf dem Chip der Visualisierungspipeline. Neben dem reduzierten Speicherplatzbedarf wird somit auch die erforderliche Speicherbandbreite zwischen Host-Computer und Visualisierungshardware signifikant reduziert. Dies ist von besonderer Bedeutung bei der Generierung von Bildern, bei denen das Working Set, d.h. die für die Darstellung benötigten Daten, nicht vollständig in den begrenzten lokalen Speicher der Visualisierungshardware passt und Daten permanent vom Host-Computer nachgeladen werden müssen.

Die Verwendung eines Multiskalenmodells stellt neue Anforderungen an die Speicherschnittstelle der Visualisierungshardware. Durch das sich dynamisch ändernde Working Set für die Bildsynthese kann kein direktes Adressierungsschema der Volumendaten mehr zum Einsatz kommen. Somit muss eine Virtualisierung der Speicherschnittstelle durchgeführt werden, ohne die Bereitstellung der Volumendaten zu verzögern und die Pipelineauslastung merklich zu reduzieren.

1.1 Verfahren zur Volumenvisualisierung

Die klassische Computergraphik benutzt zur Bildsynthese Primitiva wie Punkte, Linien und Polygone. Aus diesen Primitiva können durch beliebige Kombinationen oberflächenbasierte dreidimensionale Szenen generiert werden. Wird allerdings hinter die Oberflächen der Szenenobjekte geblickt, so sind im Inneren keine Informationen vorhanden, da die Modellierung der internen Strukturen sehr schwierig und nur mit großem Aufwand möglich ist. Die Volumenvisualisierung hingegen ist ein Verfahren zur Erzeugung zweidimensionaler Bilder aus dreidimensionalen Datensätzen, in welchen die inneren Strukturen eines Objekts repräsentiert sind. Drei grundlegend verschiedene Herangehensweisen der Volumenvisualisierung haben sich in den letzten Jahren herausgebildet. Im Folgenden werden die Ansätze in Software, auf Standardgraphikkarten sowie auf dedizierten Erweiterungskarten vorgestellt. Ein Schwerpunkt wird dabei auf die VIZARD II-Beschleunigerkarte gelegt, da sie als Grundlage dieser Arbeit verwendet wird.

1.1.1 Volumenvisualisierung in Software

Softwareimplementierungen der Volumenvisualisierung basieren vorwiegend auf der Shear-Warp-Faktorisierung [48], und werden meist auf parallelen Systemen mit „Shared Memory“ [47, 63] oder „Distributed Memory“ [54, 62] eingesetzt. Daneben entstanden Ansätze auf leistungsfähigen PCs, die unter Verwendung assembleroptimierten Routinen und Ausnutzung neuer Befehlssatzerweiterungen, wie „Multi Media Extension“ (MMX) und „Streaming SIMD (Singel-Instruction Multiple-Data) Extensions“ (SSE) versuchten interaktive Bildraten zu erzielen [42].

1.1.2 Volumenvisualisierung mit Standardgraphikkarten

Als weiteres Verfahren ist Volumenvisualisierung auf Standardgraphikkarten in den Fokus der Forschung gerückt. Anfänglich wurde hierfür hauptsächlich die 2D- und 3D-Texturemappingfunktionalität [13, 17] der Graphikkarten genutzt. Durch die Erweiterung der Graphikkarten mit konfigurierbaren Komponenten („Registercombinern“) und später programmierbaren Funktionseinheiten („Shadern“) wurde die Graphikkarte zum allgemeinen Vektorprozessor und somit auch hinsichtlich der Volumenvisualisierung deutlich leistungsfähiger [59, 76, 21, 84].

1.1.3 Volumenvisualisierung mit Spezialerweiterungskarten

Als dritter Ansatz haben sich spezielle dedizierte Beschleunigerkarten für die Volumenvisualisierung etabliert. Hiervon wurden zwar viele Architekturen vorgeschlagen [53, 19, 40, 51, 11, 46, 18, 75, 89, 74], aber nur wenige wurden als Prototyp [32, 45, 3] implementiert oder sind als kommerzielles System [66] erhältlich.

Alle vorgeschlagenen und umgesetzten Architekturen haben gemeinsam, dass sie ihren Schwerpunkt auf die Entwicklung einer effizienten Speicheranbindung legen. Eine zwingende Voraussetzung für eine effiziente Speicherschnittstelle für die Volumenvisualisierung ist die Möglichkeit, in jedem Zyklus die nötigen Daten zur Berechnung eines Abtastpunkts des Volumenvisualisierungsintegrals bereit zu stellen. Durch die Verwendung von trilinearer Interpolation für die Berechnung eines Abtastpunkts erfordert dies die Bereitstellung von acht Volumenelementen (Voxel).

Im Folgenden soll ein Überblick über die verschiedenen Ansätze der Speicheranbindung gegeben werden, wie sie sich in der Literatur präsentieren.

VIZARD (Visualization Accelerator for Realtime Display) [41, 45], der Vorgänger zu VIZARD II, ist eine PCI-Karte, die als Volumenvisualisierungs-Coprozessor dient. Sie besitzt selbst keinen Speicher für Volumendaten, sondern nutzt den Speicher des Host-Computers. Je nach Bedarf werden die Daten zum Coprozessor nachgeladen. Um die benötigte Speicherbandbreite zu reduzieren, werden die Daten in einem Vorverarbeitungsschritt verlustbehaftet komprimiert und zum Teil in einem kleinen Zwischenspeicher auf der Karte kurzfristig vorgehalten. Der 24 Kilobyte große Zwischenspeicher wird über die Manhattan-Distanz des Abtastpunkts adressiert. Die Wahrscheinlichkeit, die notwendigen Daten im schnellen Zwischenspeicher zu finden, ist vom Blickpunkt abhängig und variiert von 98 %–35 % bei Nahaufnahmen bzw. bei größeren Bildausschnitten.

Die Architekturvorschläge von Knittel et al., VERVE (Voxel Engine for Real-time Visualization and Examination) [40] und VOGUE [44, 43], besitzen keine Anforderungen an das Speicherinterface außer der Fähigkeit simultan acht Voxel liefern zu können. In diesem Architekturentwurf verwendet der Autor daher acht unabhängige Speicherbänke.

VIRIM (Virtual Reality in Medicine) von Günther et al. [32] ist eine auf Digitalen-Signal-Prozessoren (DSP) basierende Multiprozessor Karte. Als Speicher für die Volumendaten werden acht RAM-Bausteine verwendet, auf die parallel zugegriffen wird. Jeder RAM-Baustein enthält ein Achtel der Daten, wobei die genaue Anordnung der Daten im Speicher nicht veröffentlicht ist.

Ebenfalls acht unabhängige Speicherbausteine verwendet die vorgeschlagene Speicherschnittstelle von De Boer et al. [12]. Jeder der acht SDRAM-Speicher besitzt hier zusätzlich zwei oder mehr SRAM-Speicher, die als schnelle Zwischenspeicher (Cache) verwendet werden. Die Größe des gesamten Cache, inklusive der notwendigen Verwaltungstabellen, beträgt zwischen 32 MB[1] und 100 MB. Die Menge ist abhängig von der Größe der Datenblöcke, die als kleinste atomare Einheit im Cache verwaltet werden. Die Betrachtung ist allerdings nur für Datensatzgrößen mit einer maximalen Auflösung von 512 Abtastpunkten pro Raumrichtung gültig. Für höhere Auflösungen steigt der Bedarf an teurem SRAM signifikant an.

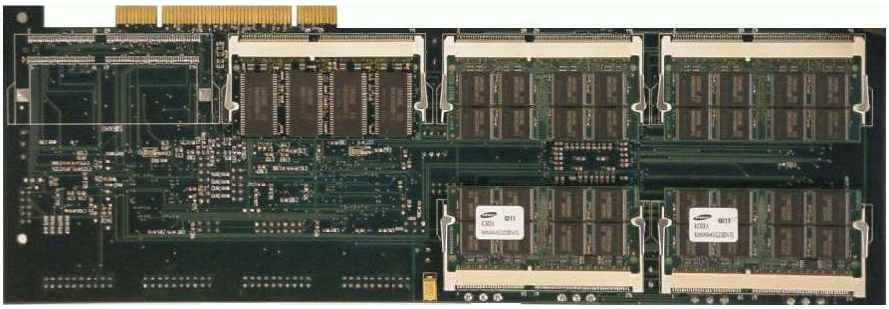
Vettermann et al. [89] setzten in ihrer Volumenvisualisierungshardware auf algorithmische Optimierungen und benutzen ebenfalls acht unabhängige Speicherbausteine zum Ablegen der Volumendaten. Um die beste Performanz der Pipeline zu erzielen, werden aber zusätzliche Anforderung an das SDRAM hinsichtlich seines Aufbaus aus Speicherbänken und SRAM-Caches gestellt, um die verschränkte Art der im Speicher abgelegten Daten zu ermöglichen. Dies bedeutet, der Aufbau des Speicherbausteins ist eng an die verwendete Pipeline gekoppelt.

VolumePro [66] ist die einzige kommerziell erhältliche Umsetzung einer hardwarebasierten Zusatzkarte. Die Implementierung ging aus den Architekturentwürfen Cube–3 [67] und Cube–4 [68] von Pfister et al. hervor. Mit der aktuellen Version VolumePro1000 [87] ist eine Dual-Chip Version mit bis zu 3,2 GB Volumenspeicher erhältlich. Die vier parallel arbeitenden Visualisierungspipelines sind so angelegt, dass alle Daten nur einmal pro Bild geholt werden müssen. Die Daten werden zwischen den Pipelines durch Puffer auf dem Chip ausgetauscht. Die enge Kopplung und der notwendige Austausch der Daten zwischen den Pipelines behindern aber algorithmische Erweiterungen und Optimierungen.

RACE II (Resample and Composite Engine II) von Ray et al. [75] versucht die erforderliche Speicherbandbreite durch das lokale Verarbeiten von Unterblöcken der Daten zu lösen. Trotzdem werden ebenfalls mindestens vier SDRAM-Speichermodule für die Volumendaten benötigt. Dazu kommen vier weitere SRAM-Speichermodule für den Bildspeicher, da bei diesem Verfahren die Bilddaten nicht nur einmal am Ende, sondern im Verlauf der Berechnung immer wieder als Zwischenergebnis geschrieben und gelesen werden.

[1] Die Datenmengen in dieser Arbeit werden in der SI-Nomenklatur angegeben, obwohl sie sich durchgängig auf Zweierpotenzen beziehen. Auf die Verwendung der korrekten Binärprefix (MiB, GiB, usw.) wurde aus Gründen der Verständlichkeit verzichtet.

(a)



(b)

Abbildung 1.1: *Die VIZARD II-Beschleunigerkarte zur Volumenvisualisierung. Auf der Oberseite (a) sind der Virtex FPGA, der Analog Devices DSP sowie die SRAM-Speicher erkennbar. Auf der Unterseite (b) sind die Speichermodule für den Voxelspeicher angebracht.*

Der Architekturentwurf GI-CUBE von Dachille et al. [18] integriert globale Beleuchtung in das Volumenvisualisierungsverfahren. Hierfür werden zwei SDRAM-Module für die Volumendaten sowie ein SDRAM für den Bildspeicher benötigt. Zusätzlich werden 32 MB „embedded DRAM“ vorgesehen. Zu dem verwendeten Cache-Schema und der Verwaltungsstruktur werden in der Publikation aber keine Angaben gemacht.

Die folgende Hardwarearchitektur zur Volumenvisualisierung und die dabei verwendete Speicheranordnung soll im Detail betrachtet werden, da sie die Grundlage für diese Arbeit ist.

1.1.4 VIZARD II

VIZARD II ist eine Spezialkarte zur interaktiven, hochqualitativen Volumenvisualisierung, die am WSI/GRIS in Zusammenarbeit mit Philips Medical Systems entwickelt wurde (vgl. Abb. 1.1). Die Karte liegt als Prototyp vor und wurde im Information Society Technologies (IST) Projekt „DynCT, Real Time Motion Compensated Reconstruction and Visualisation for Dynamic Computed Tomography“ [6, 23] der Europäischen Kommission verwendet.

Die Karte basiert auf dem Raycasting-Algorithmus [49], und wurden gezielt auf

FPGAs (Field Programmable Gate Arrays) ausgerichtet. Durch den Einsatz dieser reprogrammierbaren Chip Technologie bietet sich die Möglichkeit einmal entwickelte Komponenten auf neuen Chips wiederzuverwenden und dadurch die Entwicklungszeit deutlich zu verkürzen. Darüber hinaus sind Modifikationen an der Implementierung in kurzer Zeit umsetzbar, womit auch neue algorithmische Verfahren, wie z. B. „Preintegrated Volume Rendering“ [21], innerhalb von Wochen der Pipeline hinzugefügt werden konnten. Neben der Flexibilität ist auch die Performanz der Pipelinestruktur ein Hauptmerkmal. So konnte gezeigt werden, dass das theoretische Maximum von einem Abtastpunkt pro Taktzyklus erreicht werden kann [3]. Bei einer maximalen Taktrate von 70 MHz der Pipeline und der Verwendung von vier VIZARD II-Karten in einem PC können 280 Millionen Abtastpunkte pro Sekunde berechnet werden.

Um die notwendigen Volumendaten in ausreichender Menge zur Verfügung stellen zu können, wird dabei eine spezielle hochoptimierte Speicherschnittstelle verwendet, wie von Doggett et al. [20] beschrieben. Sie besteht aus vier SDRAM DIMM-Modulen, die Aufgrund ihres günstigen Preises und des 64 Bit breiten Datenbusses ausgewählt wurden. Da im Handel erhältlich PCI-Karten mit FPGAs nicht über eine solche Speicherausstattung verfügen, wurde die Karte selbst entworfen und gebaut. Mit dem zweifach replizierten Ablegen der Volumendaten und den vier unabhängig adressierbaren Speichermodulen können im jedem Zyklus alle acht Voxel zur Berechnung eines Abtastpunktes bereitgestellt werden. Der lokale SRAM-Puffer der Speicherbaustein wurde dabei als effizienter Cache benutzt. Fast die vollständige maximale theoretische Bandbreite der Speichermodule konnte auf diese Art durchgehend genutzt werden.

Die Kombination aus Rechenleistung der FPGA-basierten Visualisierungspipeline und Bandbreite des 2 GB Volumendatenspeicher stellt somit die Stärke des VIZARD II dar. Datensätze bis zu einer Größe von $1\,024 \cdot 512 \cdot 512$ Voxel können sowohl in Parallelprojektion als auch in perspektivischer Projektion interaktiv dargestellt werden.

Für eine detailliertere Beschreibung der VIZARD II-Karte, einschließlich der verwendeten Komponenten, der Speicheranbindung sowie den implementierten Funktionen wird auf die Arbeit von Meißner [57] verwiesen.

1.2 Kompression von Volumendaten

Der Volumenvisualisierung stellt sich das Problem, dass sich die zu visualisierenden Volumendaten stetig vergrößern. Das „Visible Human Project“ [88] zum Beispiel besteht aus der kompletten dreidimensionalen Digitalisierung eines männlichen und eines weiblichen Körpers in einer Auflösung von weniger als einem Millimeter. Die dabei entstehende Datenmenge beläuft sich auf mehrere Gigabyte. Um Datensätze dieser Größe darstellen zu können, haben sich in der Volumenvisualisierung Multiskalenmodelle etabliert. Multiskalenmodelle mit Wavelets haben sich dabei als besonders geeignet herausgestellt [90, 77, 35, 22]. Guthe et al. [31] konnten zeigen, dass sich die Komplexität hinsichtlich Rechenzeit und Working Set eines Datensatzes mit

der Größe n^3 in Multiskalendarstellung auf $O(\log n)$ reduziert. Zusätzlich kann durch die Multiskalendarstellung die Abtastrate adaptiv an lokale Frequenzkomponenten des Datensatzes angepasst werden.

Neben dem waveletbasierten Ansatz wurden weitere Vorschläge zur Kompression von Volumendaten gemacht. Vektorquantifizierung [64], Fourier/Cosinus-Transformationen [93], Lauflängencodierung [45] oder S3TC Texturkompression können bei vergleichbarer Bildqualität aber in Bezug auf die Kompressionsrate nicht mit Wavelets gleichziehen.

1.3 Überblick der wissenschaftlichen Beiträge

In dieser Arbeit wird erstmals eine Multiskalendarstellung von Volumendaten in eine Hardwarearchitektur für Volumenvisualisierung integriert. Dadurch kann die Effizienz der waveletbasierten Multiskalenhierarchie mit der Rechenleistung der VIZARD II-Visualisierungspipeline kombiniert werden. Um die notwendige Virtualisierung der Speicherschnittstelle des VIZARD II vornehmen zu können, wird eine neue cachebasierte Speicherschnittstelle, der VoxelCache, in Kapitel 3 eingeführt. Der VoxelCache erreicht durch eine zusätzliche Indirektion bei der Adressierung des lokalen Volumenspeichers und der Verwendung eines on-chip Caches auf dem FPGA eine effektive Trennung der Bildgebungspipeline von den internen Details des verwendeten Speichers. Hierdurch ist nicht nur die Verwendung eines Multiskalenmodells möglich, sondern auch die konsequente Weiterentwicklung des FPGA-Ansatzes des VIZARD II gegeben. Ohne Änderungen der Pipeline und mit alleiniger Anpassung des dem VoxelCache vorgeschalteten Speichercontrollers können neueste FPGA- und Speichertechnologien verwendet werden. Die fixe Kopplung des VIZARD II an vier unabhängige Speichermodule hat dies in der Praxis bisher verhindert, da in Ermangelung eines solchen, kommerziell verfügbaren Boards eine neue zeitaufwendige Eigenentwicklung der Trägerplattform erforderlich gewesen wäre.

In Kapitel 4 wird die Wavelettransformation vorgestellt, die als Basis für die Generierung der hierarchischen Volumendaten verwendet wird. Um die speziellen Rahmenbedingungen einer effizienten Implementierung in Hardware zu erfüllen, wird das Integer-Lifting-Schema zur Wavelettransformation verwendet. Dies zeichnet sich durch seinen minimalen arithmetischen Aufwand, geringen Speicherbedarf und sein regelmäßiges Speicherzugriffsschema aus. Eine Implementierung zur Rücktransformation der Waveletkoeffizienten in VHDL wird vorgestellt und evaluiert.

Um die VIZARD II-Pipeline an die Multiskalenhierarchie anpassen zu können, müssen mehrere Komponenten der Visualisierungspipeline modifiziert werden. Diese Änderungen werden in Kapitel 5 im Detail erläutert. Darüber hinaus werden anhand zweier Datensätze im Gigabyte-Bereich ermittelt, welche zusätzlichen Anforderungen durch diese Modifikationen entstehen.

Durch die hierarchische Darstellung beschränkt sich der Performanzgewinn bisher auf die Reduktion der Daten, die für die Bildgebung notwendig sind. Um eine weitere Reduktion der Volumendaten zu erreichen und die notwendige Bandbreite zwischen Visualisierungshardware und Host-Computer zu verringern, werden in Ka-

pitel 6 Methoden zur Kompression von Volumendaten präsentiert. Im ersten Teil wird eine effiziente Kompression zur Codierung der Waveletkoeffizienten und eine Implementierung der Decodierung in Hardware vorgestellt. Somit können alle notwendigen Schritte der Multiskalendarstellung in Hardware durchgeführt werden. Der Host-Computer ist ausschließlich für die Verwaltung der vorberechneten Multiskalenhierarchie zuständig. Die vollständige Einbindung der Waveletkompression in Hardware ermöglicht es, den Datendurchsatz zwischen Host-Computer und Visualisierungseinheit um ein Vielfaches gegenüber bestehenden softwarebasierten Dekompressionen zu steigern. Die vorgestellte Multiskalenhierarchie ist aufgrund der aufwendigen Vorverarbeitungsschritte nicht für dynamische Volumendaten geeignet, die in Echtzeit generiert werden und mit möglichst geringer Latenz visualisiert werden sollen. Hierfür wird im zweiten Teil des Kapitel 6 ein neuartiger 3D-Huffmandecoder vorgestellt der besonders an diese Erfordernisse ausgerichtet wurde.

Kapitel 2

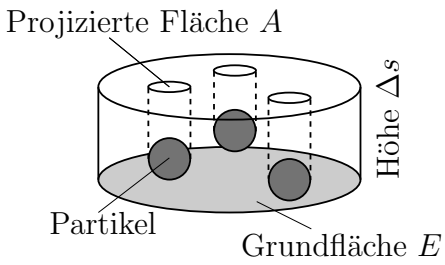
Theoretische Grundlagen der Volumenvisualisierung

Als theoretisches Gerüst der Volumenvisualisierung dient die Lichttransporttheorie, die die Interaktion des Lichts mit absorbierenden, emittierenden, reflektierenden und streuenden Materialien beschreibt. Da reflektierende und streuende Phänomene bei der in dieser Arbeit beschriebenen Implementierung nicht berücksichtigt werden, soll an dieser Stelle vor allem die theoretische Herleitung der ersten beiden Komponenten beschrieben werden.

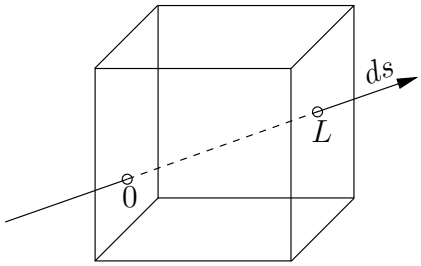
Allgemein generiert die Volumenvisualisierung eine zweidimensionale Abbildung des dreidimensionalen Datensatzes. Hierfür kann in einem Vorverarbeitungsschritt eine Zwischenrepräsentation erzeugt werden, die dann zur Bilderzeugung verwendet wird. Diese Methode der indirekten Volumenvisualisierung verwendet hierzu üblicherweise eine polygonale Repräsentation, die z. B. mittels des „Marching Cube Algorithmus“ [55] berechnet werden können. Alternativ kann ohne Zwischenrepräsentation das Bild direkt aus dem Datensatz erzeugt werden. In diesem Fall wird von direkter Volumenvisualisierung gesprochen. Hierbei wird ein Modell verlangt, das beschreibt, wie der Datensatz Licht emittiert und absorbiert. Im Folgenden wird das Modell von Max [56] vorgestellt, welches diese und weitere optische Interaktionen zwischen Licht und Medium abbildet. Um dieses kontinuierliche Modell auf den diskreten Datensatz anwenden zu können, wird als Vorbedingung angenommen, dass eine Interpolation existiert, die für beliebige Punkte s innerhalb des Datensatzes eine skalare Funktion $f(s)$ definiert. Den interpolierten Werten $f(s)$ können dann optische Merkmale wie Farbe und Opazität über eine Transferfunktion zugeordnet werden. Diese optischen Merkmale werden entlang eines Sehstrahls aufintegriert und bilden über alle Sehstrahlen das abschließende Bild.

2.1 Direkte Volumenvisualisierung

Im zugrunde liegenden Modell wird die Emission und die Absorption durch infinitesimale Partikel verursacht, die zur Vereinfachung als Kugeln mit Radius r angenommen werden, welche eine projizierte Fläche von $A = \pi r^2$ besitzen (vgl. Abb. 2.1(a)).



(a) Zylindrische Scheibe des optischen Mediums.



(b) Sehstrahl durch das Volumen.

Abbildung 2.1: *(a) Partikelsystem des Optischen Modells und (b) Darstellung der Grenzen des Linienintegrals vom Eintrittspunkt $s = 0$ bis zum Austrittspunkt $s = L$.*

Absorption

Die Dichte der Partikel in einem Medium sei ρ. In einem Zylinder mit einer Tiefe Δs und einer Grundfläche E beträgt die Anzahl der Partikel $N = E\rho\Delta s$. Für kleine Δs verschwindet die Wahrscheinlichkeit von überlappenden Partikeln und der Anteil des nicht transmittierten Lichtanteils durch die Grundfläche ist:

$$\frac{AN}{E} = \frac{AE\rho\Delta s}{E} = A\rho\Delta s \ .$$

Für $\Delta s \to 0$ folgt somit die Differentialgleichung:

$$\frac{dI}{ds} = -\rho(s)AI(s) = -\tau(s)I(s) \ , \tag{2.1}$$

wobei s der Streckenparameter entlang des Sehstrahls und $I(s)$ die Lichtintensität an der Stelle s ist. $\tau = \rho(s)A$ gibt hierbei den Anteil der Lichtverdeckung an und wird Extinktionskoeffizient genannt. Die Lösung der vorangegangenen Differentialgleichung ist:

$$I(s) = I_0 e^{-\int_0^s \tau(t)dt} \ , \tag{2.2}$$

wobei I_0 die Lichtintensität des Strahls beim Eintritt in das Volumen an der Stelle $s = 0$ beschreibt.

Emission

Den Partikeln wird neben der Absorptionseigenschaft nun zusätzlich eine Emission zugeschrieben. Der Einfachheit halber soll zunächst einzig die Emission betrachtet werden, bevor im nächsten Abschnitt Absorption und Emission kombiniert werden. Der Fall der reinen Emission kann zum Beispiel in einem heißen Gas beobachtet

werden, das aus sich selbst glüht, dabei aber nahezu transparent ist und somit kaum Licht absorbiert.

Mit der Zuordnung einer Abstrahlintensität C pro Partikel und Einheitsfläche und der bereits hergeleiteten projizierten Fläche der Partikel $AN = E\rho A\Delta s$ ergibt sich eine zusätzliche Strahlungsintensität $CE\rho A\Delta s$ der Grundfläche E. Die Änderung der Strahlungsintensität $I(s)$ ist daher

$$\frac{dI}{ds} = C(s)\rho(s)A = C(s)\tau(s) \; . \tag{2.3}$$

Die Lösung dieser Differentialgleichung ist gegeben durch:

$$I(s) = I_0 + \int_0^s C(t)\tau(t)dt \; . \tag{2.4}$$

Absorption und Emission

Um ein realistisches Modell der Wirklichkeit zu bekommen, werden nun Absorption und Emission kombiniert. Aus Gleichung 2.1 und 2.3 folgt somit folgende Differentialgleichung:

$$\frac{dI}{ds} = C(s)\tau(s) - \tau(s)I(s). \tag{2.5}$$

Als Lösung dieser Gleichung erhält man das Volume Rendering Integral (VRI):

$$I(L) = I_0 e^{-\int_o^L \tau(t)dt} + \int_o^L C(s)\tau(s)e^{-\int_s^L \tau(t)dt} ds \; , \tag{2.6}$$

wobei L die Länge des Strahls durch das optische Medium ist (vgl. Abb. 2.1(b)).

Obwohl das VRI für Spezialfälle [56] analytisch gelöst werden kann, muss es für den generellen Fall numerisch integriert werden. Hierzu wird das Integral durch die Riemannsumme

$$\int_0^L f(x)dx \approx \sum_{k=1}^{n} f(x_k)\Delta x$$

approximiert und das Integral in n gleichgroße Teilstücke $\Delta s = s_k - s_{k-1}$ mit der Länge $\frac{L}{n}$ zerteilt. Jeder Strahlpunkt s_k wird so gewählt, dass $(k-1)\Delta s \leq s_k \leq k\Delta s$ gilt. Der erste Exponentialterm aus Gleichung 2.6 wird damit angenähert durch:

$$e^{-\int_0^L \tau(s)ds} \approx e^{-\sum_{k=1}^{n} \tau(k\Delta s)\Delta s} = \prod_{k=1}^{n} e^{-\tau(k\Delta s)\Delta s} = \prod_{k=1}^{n} t_k \; , \tag{2.7}$$

wobei zur Vereinfachung $s_k = k\Delta s$ angenommen wurde. Der Term $t_k = e^{-\tau(k\Delta s)\Delta s}$ kann hierbei als Transparenz des k-ten Liniensegments entlang des Sehstrahls gedeutet werden. In gleicher Weise kann der letzte Exponentialterm aus Gleichung 2.6

approximiert werden:

$$e^{-\int_s^L \tau(t)dt} \approx e^{-\sum_{j=k+1}^{n} \tau(i\Delta t)\Delta t} = \prod_{j=k+1}^{n} t_j \; . \tag{2.8}$$

Die Transparenz kann durch die Opazität $\alpha_k = 1 - t_k$ ersetzt werden. Ist τ über Δs konstant kann α_k durch die Taylor-Reihe angenähert werden:

$$\alpha_k = 1 - t_k = 1 - e^{-\tau(s_k)\Delta s} = \tau(s_k)\Delta s - \frac{(\tau(s_k)\Delta s)^2}{2!} - \frac{(\tau(s_k)\Delta s)^3}{3!} \cdots$$

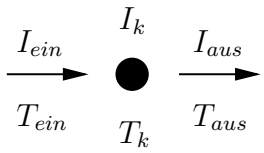
Werden für α alle Terme außer dem Ersten vernachlässigt, ergibt sich aus Gleichung 2.6 mit 2.7 und 2.8 das Diskrete Volume Rendering Integral (DVRI):

$$I(L) = I_0 \prod_{k=1}^{n}(1-\alpha_k) + \sum_{k=1}^{n} C(s_k)\alpha_k \prod_{j=k+1}^{n}(1-\alpha_j) \; . \tag{2.9}$$

Der Vorgang die Summe des DVRI zu bilden, wird Komposition genannt und kann auf zwei Arten erfolgen. Entweder man beginnt am Strahleintrittspunkt und summiert entlang der Strahlrichtung zum Strahlaustrittspunkt (Front-to-Back) oder man startet am Austrittspunkt und läuft entgegen der Strahlrichtung zum Eintrittspunkt (Back-to-Front).

Front-to-Back Compositing

```
Trans = 1,0
Inten = I[0]
for (k = 1; k < n; k++) {
   Trans = Trans * T[k - 1];
   Inten = Inten + Trans * I[k];
}
```



Back-to-Front Compositing

```
Inten = I[0]
for (k = 0; k < n; k++) {
   Inten = Inten * T[k] + I[k];
}
```

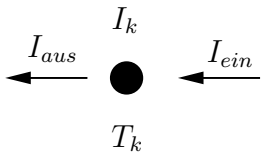


Abbildung 2.2: *Unterschiedliche Sortierung des Compositing-Verfahrens. Transparenz T_k und die Intensität $I_k = C_k \cdot \alpha_k$ werden entweder entlang oder entgegengesetzt des Sehstrahls akkumuliert.*

Der Vorteil des Back-to-Front Compositing besteht im reduzierten arithmetischen Aufwand, da die Transparenz nicht mitberechnet und zwischengespeichert werden muss. Dafür bietet Front-to-Back Compositing die Möglichkeit zu Performanzoptimierungen, indem die Berechnung des DRVI abgebrochen wird, sobald die Transparenz einen gewissen Grad (z. B. 1 %) unterschritten hat und weitere Strahlpunkte keinen erkennbaren Beitrag mehr zum Pixelwert leisten. Dies ist in der Literatur als Early-Ray-Termination [50] bekannt.

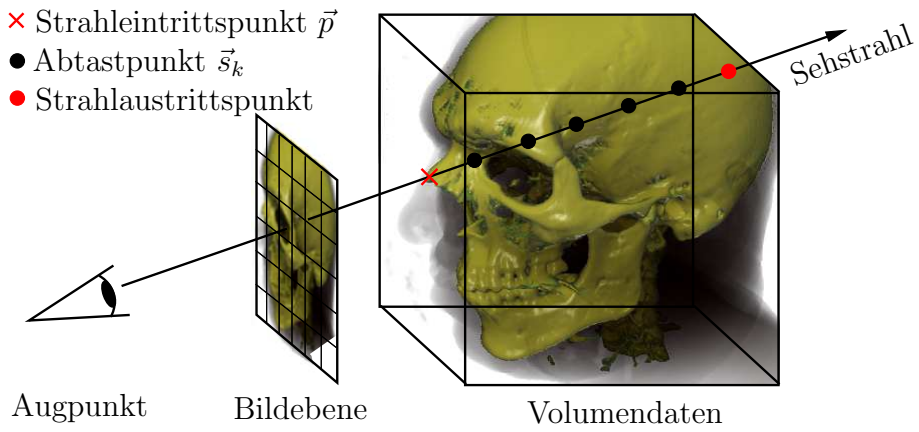


Abbildung 2.3: *Schematische Darstellung des Raycasting-Verfahrens. Vom Augpunkt aus wird für jedes Pixel der Bildebene ein Sehstrahl durch das Volumen geschickt. Beginnend beim Strahleintrittspunkt wird in regelmäßigen Abständen das Signal des Volumendatensatz abgetastet und zu einem einzigen Pixelwert zusammengefasst.*

2.2 Raycasting-Pipeline

Zur Berechnung des DVRI stehen eine Vielzahl von Methoden zur Verfügung. Prinzipiell lassen diese sich in zwei Klassen aufteilen. Die Objektraumverfahren, wie „Shear-Warp“ [48], „Splatting“ [91] und „Texture-Mapping“ [13] gehen dabei über alle Voxel im Datensatz und berechnen, welchen Beitrag ein Voxel zu welchen Pixeln leistet. Im Gegensatz dazu wird bei den Bildraumverfahren für jedes Pixel der Beitrag der Voxel berechnet. Diese im Prinzip äquivalenten Verfahren haben im Bezug auf Bildqualität, Performanz und Geschwindigkeit unterschiedliche Stärken und Schwächen. Ein ausführlicher Vergleich der Verfahren zur Volumenvisualisierung im Bezug auf diese Aspekte kann in der Arbeit von Meißner [60] gefunden werden. Da die VIZARD II-Pipeline eine Implementierung des Raycasting-Verfahrens ist, soll an dieser Stelle nur auf diese Methode eingegangen werden.

Beim Raycasting werden von einem Augpunkt aus Sehstrahlen durch die Pixel der Bildebene in das Volumen geschickt (vgl. Abb. 2.3). Entlang des Sehstrahls wird in regelmäßigen Abständen das Signal durch Abtastpunkte abgetastet. Als Optimierung können mit „Space Leaping“ [58] leere Bereiche übersprungen werden. Im Regelfall erfolgt die Ermittelung des Wertes eines Abtastpunkts durch trilineare Interpolation der benachbarten Voxel. Danach werden den Abtastpunkten Opazität und Farbwerte durch die Klassifikation zugeordnet und optional ein Beleuchtungsmodell auf den Punkt angewendet. Der kombinierte Wert aus Klassifikation und Beleuchtungsmodell wird im Folgenden mit den anderen Werten der Abtastpunkte entlang des Sehstrahls aufsummiert.

Die einzelnen Stufen des Raycastings können hintereinander in einer Pipeline ausgeführt werden und sind somit gut für eine Implementierung in Hardware geeignet. Abbildung 2.4 stellt die einzelnen Stufen, sowie deren Ablauf dar. Für eine detaillierte Beschreibung der einzelnen Stufen und deren Implementierung in der

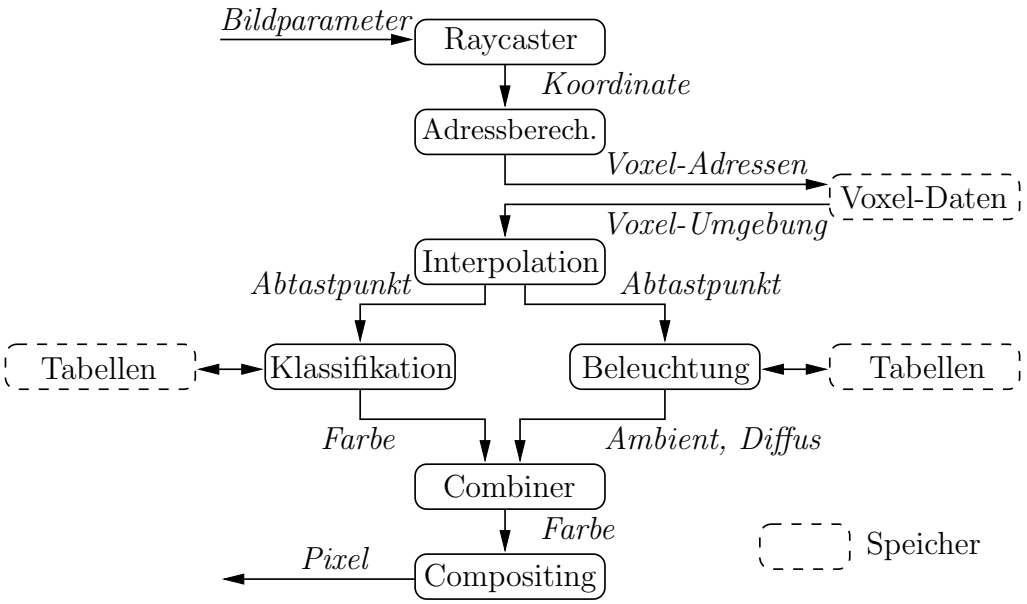


Abbildung 2.4: *Illustration der einzelnen Schritte der Volumenvisualisierungspipeline und ihrer Reihenfolge anhand der VIZARD II-Architektur.*

VIZARD II-Pipeline wird auf die Arbeiten von Meißner [57, 3] und Kanus et al. [2] verwiesen.

Raycaster

Der Raycaster bestimmt anhand der vorgegebenen Bildparameter (Augpunkt, Volumengrenzen sowie Auflösung und Lage der Bildebene) den Eintrittspunkt $\vec{p}_E = (p_x, p_y, p_z)$ eines Sehstrahls in das Volumen. Zusammen mit dem Inkrementvektor $\vec{i} = (i_x, i_y, i_z)$ werden daraus die Abtastpunkte

$$\vec{s}_k = \vec{p} + k \cdot \vec{i} \quad k \in \mathbb{N}$$

entlang des Sehstrahls berechnet. Deren Position wird dann mit den Volumengrenzen verglichen, um mit dem Austrittspunkt den Strahl zu beenden. Die Abtastrate kann über Verkürzung oder Verlängerung des Inkrementvektors $\vec{i}$ erhöht bzw. verkürzt werden.

Interpolation

Die Voxel des Volumendatensatzes liegen üblicherweise nur an diskreten Punkten des Volumens vor. Fällt ein Abtastpunkt nicht auf einen dieser Punkte, muss aus den ihn umgebenden Voxel ein Wert interpoliert werden. Der Auswahl des Rekonstruktionsfilters für die Interpolation kommt somit eine großen Bedeutung zu. Besonders bei einer Implementierung in Hardware muss hierbei aber die Qualität der rekonstruierten Funktion gegen den damit verbundenen Aufwand abgewogen werden.
Am einfachsten lässt sich das „Nearest Neighbor“ Verfahren implementieren, welches den Wert des am nächsten liegenden Voxel als Interpolationsergebnis liefert.

Dieses Verfahren stellt somit im engeren Sinn kein Interpolations-, sondern ein Auswahlverfahren dar.

Typischerweise werden lineare Interpolationsverfahren gewählt, da ihr arithmetischer Aufwand begrenzt ist und die Filterkerne separabel sind und somit in alle Raumrichtungen einzeln angewendet werden können. Als Standard hat sich die trilineare Interpolation etabliert, die aus den acht Voxel der Umgebung den Interpolationswert anhand ihres Filterkerns

$$f(d) = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \cdot d + f(x_0) \quad d \in [0, 1]$$

ermittelt.
Interpolationsverfahren höherer Ordnung kommen aufgrund ihres arithmetischen Aufwands und der Größe des Filterkerns kaum zum Einsatz.

Klassifikation

Die Klassifikation ist die Zuordnung von Voxelwerten $f(s_k)$ zu optischen Eigenschaften des Materials wie Farbe C und Opazität α. D. h. für jede Materialeigenschaft M_i existiert eine Transferfunktion

$$M_i = O_i(f(s_k)),$$

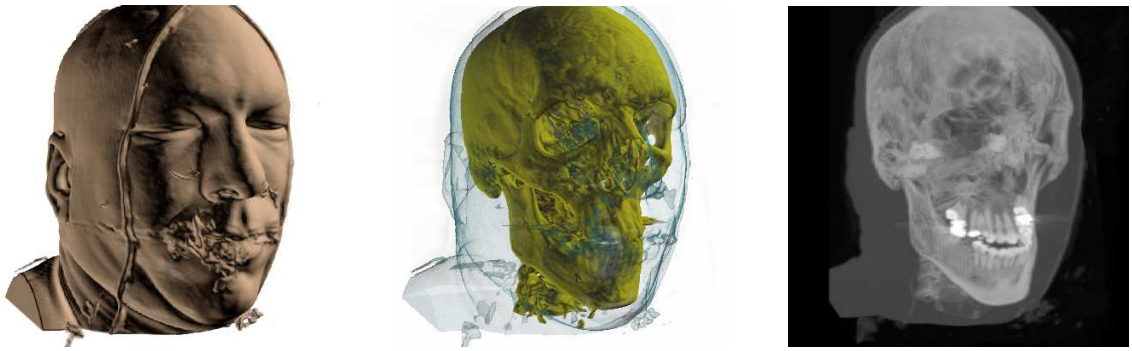
die dem skalaren interpolierten Abtastpunkt einen Wert zuordnet. Die Klassifikation kann entweder Punkt- oder Intervallbasiert [21] erfolgen. Neben dem skalaren Voxelwert können auch weitere Voxeleigenschaften als Parameter der Transferfunktion fungieren. So können unter Verwendung der Voxelgradienten auch mehrdimensionale Transferfunktionen genutzt werden [39]. Durch die Wahl der Transferfunktion können so unterschiedliche Werte und Bereiche hervorgehoben werden. Abbildung 2.5 (a und b) verdeutlicht dies durch zwei Aufnahmen, die sich nur in der verwendeten Transferfunktion unterscheiden.

Beleuchtung

Der dreidimensionale Eindruck des Volumendatensatzes kann durch die Anwendung eines Beleuchtungsmodells verstärkt werden. Dieses kann über die Hervorhebung von Grenzflächen und den damit verbundenen verstärkten räumlichen Eindruck die Interpretation der Volumendaten deutlich erleichtern. Um die Reflexion des Lichts an den Grenzflächen zu berechnen wird für ein Beleuchtungsmodell die Normale der Fläche benötigt. Hierzu wird der Gradient benutzt, der innerhalb des Datensatzes die Richtung der Änderung der Voxelwerte angibt. Die Gradienten können optional vorberechnet oder auf Anforderung erzeugt werden. Üblicherweise kommt als lokales Beleuchtungsmodell das Phong-Modell [72] zur Anwendung:

$$C = k_a \cdot C(s_k) + k_d \cdot C(s_k) \cdot I_d + k_s \cdot I_s \ .$$

Die Koeffizienten für den ambienten, diffusen und spekularen Anteil (k_a, k_d und k_S) werden als weitere Materialeigenschaft bei der Klassifikation ermittelt. I_d und I_s



(a) *(b)* *(c)*

Abbildung 2.5: *Durch die Verwendung unterschiedlicher Klassifikationen können verschiedene Merkmale des Datensatzes hervorgehoben werden ((a) und (b)). Wird statt des DVRI das MIP-Verfahren zum Compositing angewendet, erhält man röntgenbildartige Ergebnisse (c).*

geben die ambiente bzw. die spekular Intensität des reflektierten Lichts der Lichtquellen an.
Im Fall der VIZARD II-Pipeline wurde die direkte Berechnung des Phong-Modells durch ein tabellenbasiertes Verfahren ersetzt [33]. Diese Besonderheit macht auch im VIZARD II eine zusätzliche Pipelinestufe, den „Combiner“, notwendig, der die Werte aus Beleuchtungsrechnung und Klassifikation zusammenführt (vgl. Abb. 2.4).

Compositing

Die Summation des DVRI erfolgt in der Compositing Einheit. Hier werden alle Abtastpunkte zum finalen Pixelwert zusammengefügt. Neben dem DVRI existieren noch andere einfachere Compositing-Verfahren. Die bedeutendsten sind hierbei „Maximum-Intensity-Projection“ (MIP) und „Minimum Intensity Projection“ (mIP), die entlang eines Strahls jeweils nur den maximalen bzw. minimalen Wert als finalen Pixelwert übernehmen. Hiermit lassen sich röntgenbildartige Aufnahmen erzielen (vgl. Abb. 2.5(c)).

Kapitel 3

VoxelCache: Eine cachebasierte Speicherarchitektur

Der Schwerpunkt aller in Abschnitt 1.1.3 vorgestellten Architekturen zur Volumenvisualisierung liegt in der effizienten Speicheranbindung der Visualisierungspipeline. Die Bereitstellung der vollständigen Voxelumgebung für die Abtastung des Signals und der Berechnung von Gradienten in jedem Taktzyklus ist das Hauptaugenmerk aller bisherigen Vorschläge. Um einen günstigen Zugriff während des Darstellens zu ermöglichen, wurden viele verschiedene Schemata vorgeschlagen, wie die Daten im Speicher abzulegen sind. Durch die vorgeschlagenen optimierten Speicherzugriffe entsteht aber eine enge Kopplung zwischen der Visualisierungspipeline und der verwendeten Speicherart. Um eine dynamische Verwaltung der Speicherinhalte zu ermöglichen, wie sie für eine spätere Integration des Multiskalenmodells erforderlich ist, und Working Sets zu unterstützen, die nicht vollständig in den lokalen on-board Speicher der Karte passen, muss eine Virtualisierung der Speicherschnittstelle durchgeführt werden. Die Adressierung der Daten im Speicher wird nicht mehr direkt von der Visualisierungspipeline vorgenommen, sondern erfolgt über eine weitere Komponente, die die erforderlichen Daten dynamisch bereitstellt.

Zusätzlich macht eine enge Kopplung zwischen Visualisierungspipeline und Speicher es schwer die fortdauernden technologischen Fortschritte in der Speichertechnologie zu nutzen. Jeder Wechsel zu einer anderen Speicherart, -technologie und -anzahl macht einen Eingriff in die Visualisierungspipeline erforderlich. Durch das verwendete Adressierungsschema sind zum Beispiel bei VIZARD II [3, 2] zwingend vier SO-DIMM Module erforderlich, die aus genau 16 SDRAMs mit jeweils vier Bänken bestehen müssen, da die SRAM-Cachelines des SDRAM als Cache für die Pipeline verwendet werden. Eine andere Art von Speicher hätte dramatische Einbußen der Performanz zur Folge, die den Geschwindigkeitsvorteil einer Hardware Implementierung zunichte macht. Da die Entwicklung einer Visualisierungspipeline in Hardware trotz spezieller Hochsprachen wie VHDL [8] eine personen- und zeitintensive Aufgabe ist, ist es langfristig erforderlich einen Weg zu finden, der es erlaubt sowohl den Geschwindigkeitsvorteil einer Hardwareimplementierung als auch den des allgemeinen technologischen Fortschritts zu nutzen und eine dynamische Verwaltung

der Volumendaten zu ermöglichen.

Der im Folgenden vorgestellte VoxelCache erfüllt genau diese Anforderungen. Er bietet der Visualisierungspipeline eine effiziente Schnittstelle, die alle Details der verwendeten Speicherart des Voxelspeichers vor der Pipeline verbirgt und er ermöglicht die Voxeldaten ohne festes Muster im Voxelspeicher ablegen zu können. Auf diese Weise bietet sich die Option, mittels Multiskalenmodellen Datensätze zu visualisieren, die aufgrund ihrer Größe bisher außerhalb der Möglichkeiten von dedizierter Volumenvisualisierungshardware lagen.

Um die Auswirkungen des VoxelCache auf die Performanz der Visualisierungspipeline zu ermitteln, wurde anhand einer zyklenakkuraten C++-Simulation für verschiedene Speicher- und Bustypen die Effizienz ermittelt [1]. Basierend auf den positiven Ergebnissen der C++-Simulation, die eine Trefferrate von über 98 % ergab, wurde der VoxelCache in VHDL implementiert und wiederum bezüglich der Cache Trefferrate und der Pipelineauslastung in einer zyklenakkuraten Simulation getestet.

3.1 Systemüberblick

Der VoxelCache soll zu einer effektiven Trennung zwischen Visualisierungspipeline und Speicher der Voxeldaten führen, so dass alle Details des Speichers vor der Visualisierungspipeline verborgen sind. Abbildung 3.1 zeigt das Zusammenspiel zwischen Visualisierungspipeline, VoxelCache und Speicherschnittstelle. Alle spezifischen Details der Bildgebung sind in der Visualisierungspipeline und dem VoxelCache gekapselt, während alle relevanten Details des Speichers in der Speicherschnittstelle gesammelt sind.

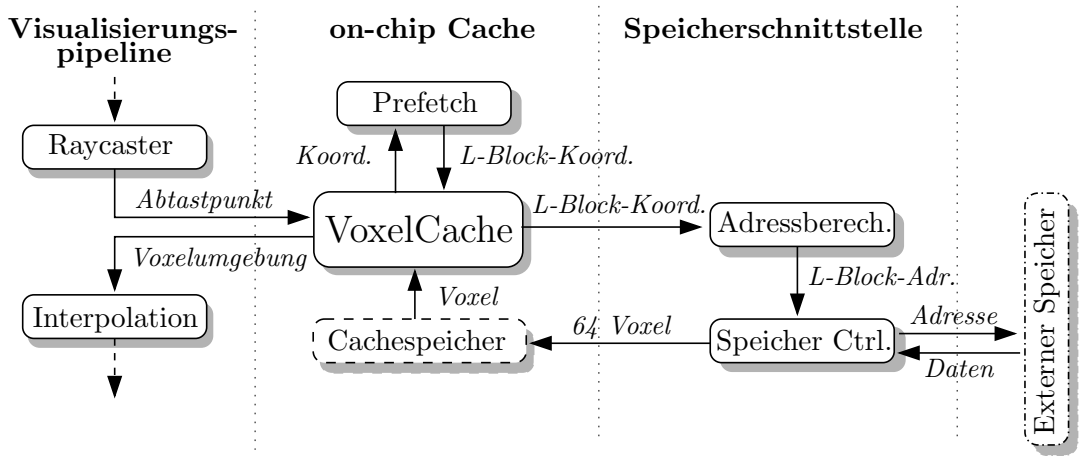


Abbildung 3.1: *Überblick über das Zusammenspiel von Visualisierungspipeline, VoxelCache und Speicherschnittstelle.*

Die Koordinate des momentanen Abtastpunktes wird von der Visualisierungspipeline an den VoxelCache übergeben. Dieser prüft, ob sich alle Voxel für die notwendige Voxelumgebung bereits im on-chip Speicher des Cache befinden. Sind die Daten vorrätig, werden diese der Interpolationseinheit übergeben. Sind keine oder

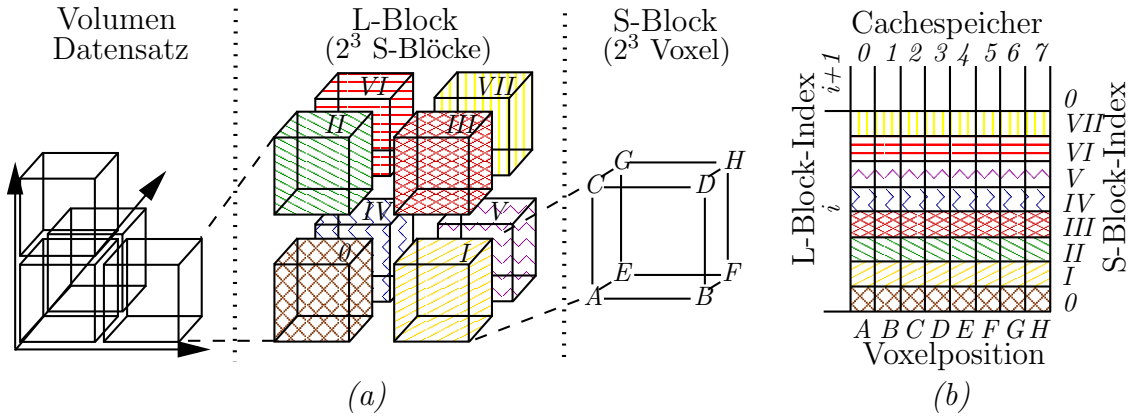


Abbildung 3.2: *(a) Hierarchische Unterteilung des Volumendatensatzes in L- und S-Blöcke, sowie (b) deren Anordnung in den acht Cachespeichern. Die Adresse jedes Voxel berechnet sich aus L-Block-Index, S-Block-Index und Voxelposition.*

nicht alle Daten vorhanden um die Anfrage zu bedienen, werden über eine feste Schnittstelle die notwendigen Informationen über die fehlenden Blöcke an die Speicherschnittstelle übergeben. Diese holt daraufhin die Daten aus dem Voxelspeicher und initiiert die Aktualisierung des Cache mit den angeforderten Daten.

3.2 Cache Organisation

Um die Verwaltung der Voxel zu vereinfachen, wurde eine hierarchische Speicherorganisation gewählt (vgl. Abb. 3.2). Die kleinste Einheit bilden die S-Blöcke. Sie bestehen aus zwei Voxel in x-, y-, z-Richtung und formen genau eine Achterumgebung. Jeweils acht S-Blöcke werden zur nächsthöheren Hierarchiestufe, dem L-Block, zusammengefasst. Die L-Blöcke stellen im Bezug auf Speichertransfer eine atomare Einheit dar. Die Größe der L-Blöcke von 4^3 Voxel wurde ausgewählt, da bei verschränkter Abarbeitung von Strahlen die Wahrscheinlichkeit für einen fehlgeschlagenen Chachezugriff bei 2^3 sehr hoch würde. L-Block Größen über 4^3 würden zwar den Verwaltungsaufwand deutlich reduzieren, würden aber zu einer sehr hohen Latenz beim Nachladen der notwendigen L-Blöcke führen und die maximale Anzahl der L-Blöcke im Cache signifikant reduzieren.

Wie bereits in Abschnitt 2.2 vorgestellt, beruht die Interpolation des Strahlpunktes (s_k) auf einer Umgebung von acht Voxel. Um in jedem Taktzyklus einen Interpolationswert berechnen zu können, muss der VoxelCache in der Lage sein, eine komplette Achterumgebung parallel zur Verfügung zu stellen. Dies muss auch garantiert werden, sollte diese Achterumgebung aus Voxel unterschiedlicher S- und L-Blöcke bestehen. Die hierzu häufig verwendete Methode, Voxeldaten im Speicher zu replizieren, kann aufgrund des limitierten on-chip Cachespeichers nicht angewendet werden. Um pro Taktzyklus unabhängig von ihrer Zugehörigkeit zu S- und L-Block eine Achterumgebung vom VoxelCache an die Visualisierungspipeline transferieren zu können, sind die Daten in acht unabhängig adressierbaren Cacheschichten des

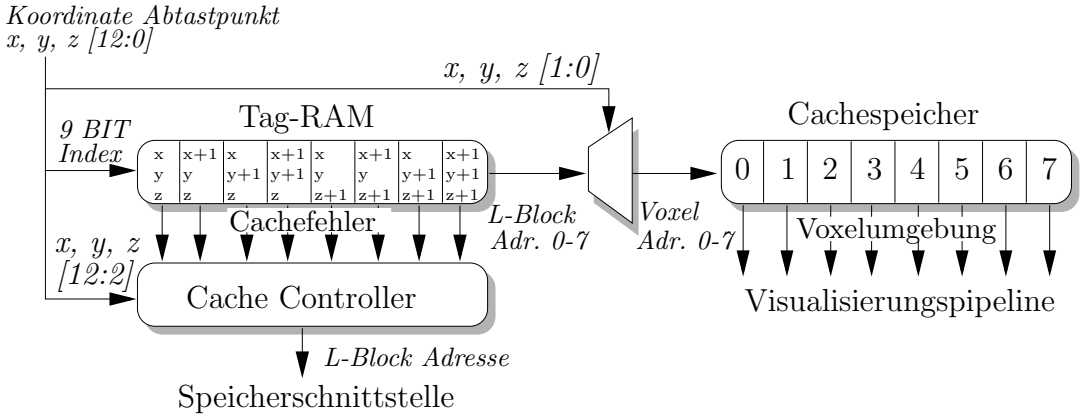


Abbildung 3.3: *Schematischer Aufbau des VoxelCache mit 8 Tag-RAMs und 8 Cachespeichern. Zu jedem Abtastpunkt werden die 8 Voxel der Voxelumgebung der Visualisierungspipeline übergeben, oder die notwendigen Daten über die Speicherschnittstelle angefordert.*

Cachespeichers gespeichert. Jede Voxel Position (A–H) innerhalb der S-Blöcke wird dabei in einer anderen Cacheschicht abgelegt (vgl. Abb. 3.2(b)).

Die Zuordnung von Abtastpunkt zu Voxelumgebung erfolgt in zwei Schritten. Im ersten wird in einem Tag-RAM der L-Block-Index im Cachespeicher bestimmt (vgl. Abb. 3.3). Die Adressierung des Tag-RAM ist „Direct Mapped“ und beruht auf der x-, y-, z-Koordinate des Abtastpunkts. Im zweiten Schritt werden über den L-Block-Index die Voxel im Cachespeicher adressiert. Durch das explizite Ablegen der L-Block-Indizes im Tag-RAM kann jeder L-Block an jeder Position des Cachespeichers gespeichert werden. Die Zuordnung von Tag-RAM zu Cachespeicher ist somit voll assoziativ. Diese ist nützlich in Systemen, in denen nicht genügend Platz zur Verfügung steht, um 512 L-Blöcke zu speichern, da in diesem Fall L-Blöcke austauschbar sein müssen. In der aktuell implementierten Version des VoxelCache können bis zu 128 L-Blöcke abgespeichert werden.

Sollten nicht alle acht benötigten Voxel der Achterumgebung im selben L-Block liegen, müssen alle L-Blöcke einzeln im Tag-RAM nachgeschaut werden. Um dieser nicht unwahrscheinlichen Situation vorzubeugen, ist das Tag-RAM achtfach instantiiert, so dass alle L-Block-Indizes gleichzeitig ermittelt werden können und nicht nacheinander abgearbeitet werden müssen.

3.3 Cache Adressierung

Als Eingabe erhält der VoxelCache den Ganzzahlteil der Koordinate des aktuellen Strahlpunkts. Die Koordinate besteht dabei aus jeweils 13 Bit für die x-, y- und z-Dimension. Von diesen 39 Bit werden 9 Bit als Tag-RAM-Adresse benutzt. Die Verteilung der 9 Adressbit erfolgt entweder gleichmäßig (je 3 Bit) oder wird nach der Hauptrichtung des Sehstrahls gewichtet. Die Komponente der Hauptrichtung wird für diesen Fall mit 5 Bit, die anderen Komponenten mit jeweils 2 Bit berücksichtigt. Dies führt zu $2^9 = 512$ Einträgen im Tag-RAM und entspricht $128 \times 16 \times 16$

konfliktfrei adressierbaren Voxel.

Jeder Tag-RAM-Eintrag hat neben dem L-Block-Index ein Gültigkeitsbit und die übrigen Bits der Koordinate. Sollte das Gültigkeitsbit gesetzt sein, werden diese abgelegten Koordinatenbits mit den Bits der Koordinaten, die nicht zur Adressierung des Tag-RAM benutzt wurden, verglichen. Im Falle einer Übereinstimmung wird der gespeicherte L-Block-Index zur Adressierung des Cachespeichers weitergegeben.

Nachdem im Tag-RAM das Vorhandensein der angeforderten Voxel kontrolliert und bestätigt wurde, wird nun die genaue Adresse der Voxel im Cachespeicher bestimmt. Hierzu wird aus der im Tag-RAM gespeicherten L-Block Adresse und einem 3 Bit Index innerhalb des L-Blocks die endgültige Adresse für jede Cacheschicht gebildet. Der 3 Bit Index ergibt sich aus der logischen Kombination der untersten 2 Bit der Strahlpunkt-Koordinate (vgl. Abb. 3.4).

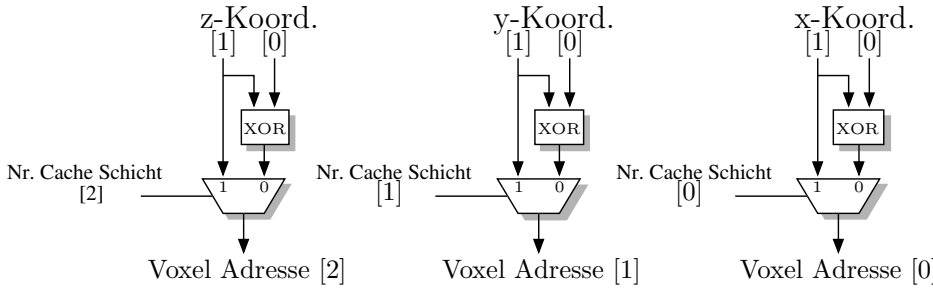


Abbildung 3.4: *Aus den untersten zwei Bit der x-, y-, z-Koordinate und den untersten 3 Bit des Cachespeicher Nummer wird die Position des Voxel innerhalb eines L-Block bestimmt.*

Da die Achterumgebung eines Strahlpunkts sich über bis zu acht L- und S-Blöcke erstrecken kann, jeder Cachespeichers aber nur genau eine Position des S-Block liefert, müssen die Indizes vor der Adressierung des jeweiligen Cachespeichers sortiert werden. Liegen alle acht Voxel innerhalb desselben L-Blocks ist diese Sortierung aufgrund der identischen L-Block-Indizes folgenlos. Erstreckt sich die Achterumgebung auf mehrere L-Blöcke (die letzten zwei Bits der Strahlpunktkoordinaten sind gesetzt), müssen die L-Block-Indizes der Cachespeicher nach folgender Abbildung getauscht werden:

z-Koordinate [1:0] = "11"	y-Koordinate [1:0] = "11"	x-Koordinate [1:0] = "11"
$0 \leftrightarrow 4$	$0 \leftrightarrow 2$	$0 \leftrightarrow 1$
$1 \leftrightarrow 5$	$1 \leftrightarrow 3$	$2 \leftrightarrow 3$
$2 \leftrightarrow 6$	$4 \leftrightarrow 6$	$4 \leftrightarrow 5$
$3 \leftrightarrow 7$	$5 \leftrightarrow 7$	$6 \leftrightarrow 7$

Der Austausch erfolgt sequentiell in z-, y-, x-Richtung, falls die letzten beiden Bit der Strahlpunkt-Koordinate in der entsprechenden Komponente gesetzt sind. Die Operation ist kommutativ und kann daher in beliebiger Reihenfolge durchgeführt werden.

Die Sortierung der L-Block-Indizes ist aber noch nicht ausreichend, um eine korrekt sortierte Achterumgebung herzustellen. Innerhalb eines L-Blocks muss die

Lage der Achterumgebung berücksichtigt werden. So kann z. B. Cacheschicht Nr. 0 nur die Position „A“ eines S-Blocks bereitstellen. Befindet sich die untere linke Ecke der Achterumgebung aber auf S-Block Position „A“, stellt nicht Cacheschicht Nr. 0 sondern Cacheschicht Nr. 1 diese zur Verfügung. Die Voxel von Cacheschicht Nr. 0 und Nr. 1 müssen in diesem Fall getauscht werden. Allgemein kann an dem niederwertigsten Bit der x-, y-, z-Strahlpunktkoordinate geprüft werden, ob ein Austausch in dieser Raumrichtung notwendig ist. Der Austausch folgt demselben Schema wie bei den L-Blocks.

3.4 Spekulatives Füllen des Cache / Prefetching

Die Latenz des VoxelCache ist im Falle einer nicht bedienbaren Anfrage an den Cache sehr hoch. Im schlimmsten Fall müssen für einen Strahlpunkt acht L-Blöcke vom externen Speicher in den Cachespeicher transferiert werden, bevor eine Achterumgebung für die Visualisierungspipeline bereitgestellt werden kann. Eine Möglichkeit, die Zahl der fehlerhaften Anfragen zu reduzieren, ist ein Schema zu entwickeln, welches spekulativ versucht den Cache frühzeitig mit den richtigen Daten zu füllen.

Prinzipiell muss bei perspektivischem Raycasting mit parallel bearbeiteten Sehstrahlen mit einem unstrukturiertem Zugriffsmuster der Abtastpunkte gerechnet werden. In der Praxis besitzen die Strahlen aber dennoch eine gewisse räumliche Kohärenz. Diese Kohärenz zu nutzen, ist die Aufgabe des Prefetching. Mit der Vorhersage soll die vorhandene Bandbreite zum externen Speicher besser genutzt werden. Das Prefetching bestimmt aufgrund des aktuellen L-Blocks, aus dem die Visualisierungspipeline ihre momentanen Voxel anfragt, eine Liste von L-Blocknachbarn, die als potentielle nächste Blöcke in Frage kommen. Diese Liste von Nachbarn wird nach einem Prioritätskriterium sortiert, und falls keine akuten Fehlanfragen zu bearbeiten sind, präventiv in den VoxelCache geladen. Die Zeit die zur Verfügung steht um spekulativ einen L-Block zu holen, ist stark von der Abtastrate entlang eines Sehstrahls abhängig, da die Anzahl der Strahlpunkte, welche innerhalb eines L-Blocks liegen, mit einer höheren Abtastrate zunimmt.

Die Bestimmung der Priorität hat einen maßgeblichen Einfluss auf die Effizienz des Prefetching Schemas und sollte daher sehr genau sein. Auf der anderen Seite sollte die Bestimmung der Priorität schnell und mit geringem arithmetischen Aufwand erfolgen, um die Ressourcen, die für diese Einheit aufgewendet werden müssen, gering zu halten. Es wurde daher eine einfache Methode gewählt, die die Priorität der Nachbarblöcke bestimmt, ohne exakt den nächsten L-Block entlang des Sehstrahls berechnen zu müssen. Durch das Vorzeichenbit der x-, y-, z-Komponente des Strahlpunktes wird ein Oktant bestimmt, der die Hauptrichtung des Sehstrahls codiert. Anhand dieses Oktanten können die sieben möglichen Nachbarn eines L-Blocks bestimmt werden, die potentiell als nächste von dem Sehstrahl betreten werden (vgl. Abb. 3.6).

Abbildung 3.5 zeigt die Hauptkomponenten und den Aufbau der Vorhersage-Einheit. Die Nachbar-Einheit berechnet anhand des Oktanten des Sehstrahls die sieben möglichen Nachbarn des L-Blocks. Bei diesen wird zwischen drei verschiede-

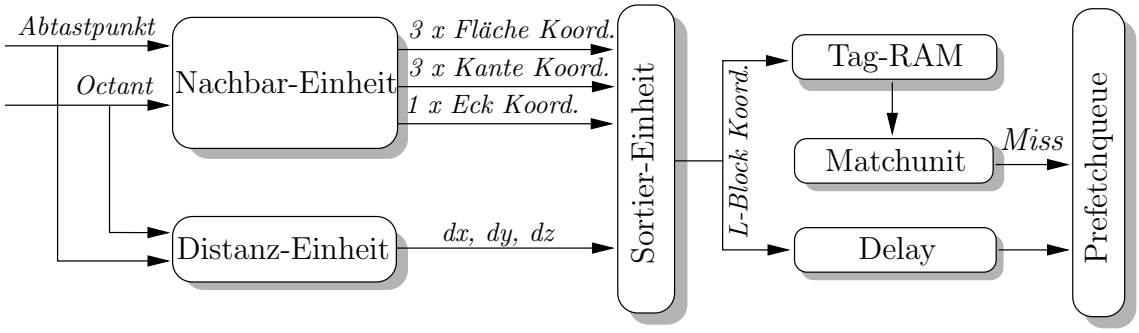


Abbildung 3.5: *Nachbar-Einheit des VoxelCache. Die Einheit ermittelt anhand der aktuellen Position des dominanten Strahls einer Strahlgruppe die möglichen Nachbarblöcke. Die möglichen Nachbarblöcke werden zunächst überprüft ob sie bereits im Cache vorhanden sind. Wenn dies nicht der Fall ist, werden sie zum Holen vorgemerkt.*

nen Arten unterschieden: drei Nachbarn verfügen über eine gemeinsame Fläche, drei über eine gemeinsame Kante und ein Nachbar besitzt eine gemeinsame Ecke. Die Nachbarn zum momentanen L-Block werden aus der aktuellen L-Block-Basis und der Länge eines L-Blocks (ΔL) berechnet. Hierbei wird in Abhängigkeit von den gesetzten Bits des Oktanten die Länge des L-Blocks addiert oder subtrahiert:

Flächen:	$(\mathrm{x}\pm\Delta L, \mathrm{y}, \mathrm{z})$, $(\mathrm{x}, \mathrm{y}\pm\Delta L, \mathrm{z})$, $(\mathrm{x}, \mathrm{y}, \mathrm{z}\pm\Delta L)$
Kanten:	$(\mathrm{x}\pm\Delta L, \mathrm{y}\pm\Delta L, \mathrm{z})$, $(\mathrm{x}\pm\Delta L, \mathrm{y}, \mathrm{z}\pm\Delta L)$, $(\mathrm{x}, \mathrm{y}\pm\Delta L, \mathrm{z}\pm\Delta L)$
Ecke:	$(\mathrm{x}\pm\Delta L, \mathrm{y}\pm\Delta L, \mathrm{z}\pm\Delta L)$

Parallel zu der Position der Nachbarn wird der jeweilige Abstand zwischen aktueller Strahlpunktposition und den Nachbarn durch die *Abstands-Einheit* berechnet. Ist das Oktantenbit der entsprechenden Komponente nicht gesetzt, wird der Abstand $\Delta(x, y, z)$ zwischen dem aktuellen Strahlpunkt und den Nachbarn berechnet, indem der Abstand des Strahlpunkts zum aktuellen Ursprung des L-Blocks von der L-Blocklänge ΔL abgezogen wird. Die Berechnung des Abstands vom Strahlpunkt zum aktuellen Ursprung des L-Blocks entspricht bei einer L-Blocklänge von vier Voxel den letzten zwei Bits der Strahlpunktkomponente. Ist das Oktantenbit gesetzt, werden die letzten zwei Bits der Strahlpunktkomponente direkt als Ergebnis genommen. Der Abstand des Ecknachbarn muss nicht berechnet werden, da bei der späteren Sortierung ihm immer die niedrigste Priorität zugeordnet wird.

Die Position der Nachbarn, sowie die jeweiligen Abstände zum Strahlpunkt, werden der Sortier-Einheit übergeben. Diese betrachtet zunächst alle Flächennachbarn und sortiert sie mit aufsteigendem Abstand in die Prefetchqueue. Die Flächennachbarn werden vor allen anderen Nachbarn betrachtet, da die Distanzen der Kantennachbarn identisch sind, diese aber eine deutlich niedrigere Wahrscheinlichkeit aufweisen als nächster L-Block benötigt zu werden. Nachdem die Flächennachbarn in die Prefetchqueue einsortiert sind, werden die Kantennachbarn ebenfalls nach aufsteigendem Abstand einsortiert. Der Ecknachbar ist immer der letzte der eingetragen wird.

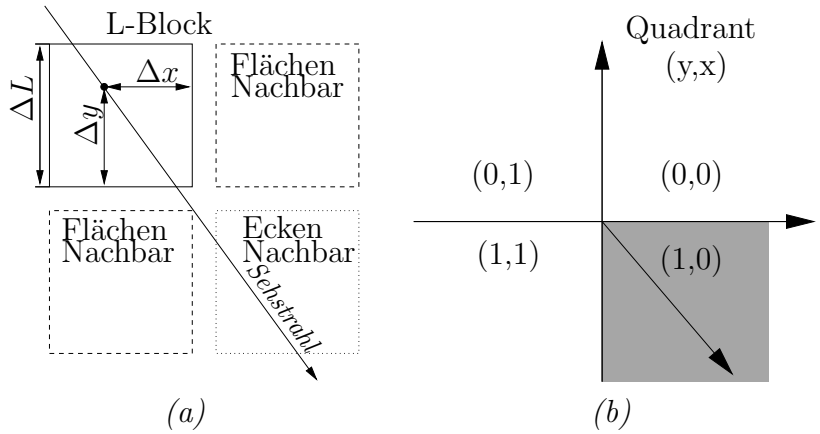


Abbildung 3.6: *(a) 2D Aufsicht auf einen L-Block mit seinen potentiellen Flächen- und Ecknachbarn, sowie (b) Darstellung der, dem Sehstrahl entsprechenden, Oktantencodierung (z-Richtung nicht dargestellt).*

Die Prefetchqueue wird mit Verlassen eines L-Blocks gelöscht und beim nächsten Sehstrahlpunkt neu erzeugt. Die L-Blöcke der Prefetchqueue werden vom Speicherkontroller nachrangig bedient. Auf diese Weise wird garantiert, dass Fehlanfragen an den Cache immer so schnell wie möglich bedient werden.

Aus Gründen der Effizienz ist es für die Visualisierungspipeline besser, die einzelnen Sehstrahlen, nicht einzeln nacheinander abzuarbeiten, sondern in Strahlgruppen verschachtelt zu bearbeiten. Von jedem Strahl der Gruppe wird jeweils ein Strahlpunkt bearbeitet, bevor der jeweilige Sehstrahl um das nächste Intervall vorgeschoben wird [4]. Die einzelnen Strahlen besitzen zwar eine gewisse Kohärenz, müssen sich aber nicht notwendigerweise im selben L-Block befinden. Daraus folgt, dass aufeinander folgende Anfragen der Pipeline an den VoxelCache nicht im selben L-Block liegen. Dies würde zu einem konstanten Löschen und Neuerzeugen der Prefetchqueue führen. Um diese negative Beeinflussung der Sehstrahlen zu vermeiden, besitzt jede Strahlgruppe einen dominanten Strahl, der für das Prefetching relevant ist. Im Fall von sehr stark perspektivischen Projektionen kann dies die Effizienz des Prefetching reduzieren. Der dominante Strahl ist auch der einzige Strahl, der einem Oktanten zugeordnet wird. Dies führt zu einem einzigen Oktanten pro Strahlgruppe und reduziert die administrativen Informationen, die entlang der Pipeline mitgeführt werden müssen. Für pathologische Fälle können einzelne Strahlgruppen durch die perspektivische Projektion Strahlen vereinen, die nicht denselben Oktanten besitzen. Dies ist aber nur für wenige Strahlgruppen in der Bildmitte überhaupt möglich und damit ein vernachlässigbarer Effekt.

3.5 Implementierung

Neben der initialen C++-Implementierung wurde der VoxelCache in VHDL umgesetzt, synthetisiert und simuliert. Abbildung 3.7 zeigt das Zusammenspiel der einzelnen Einheiten, die Datenpfade und die zur Synchronisation notwendigen Verzöge-

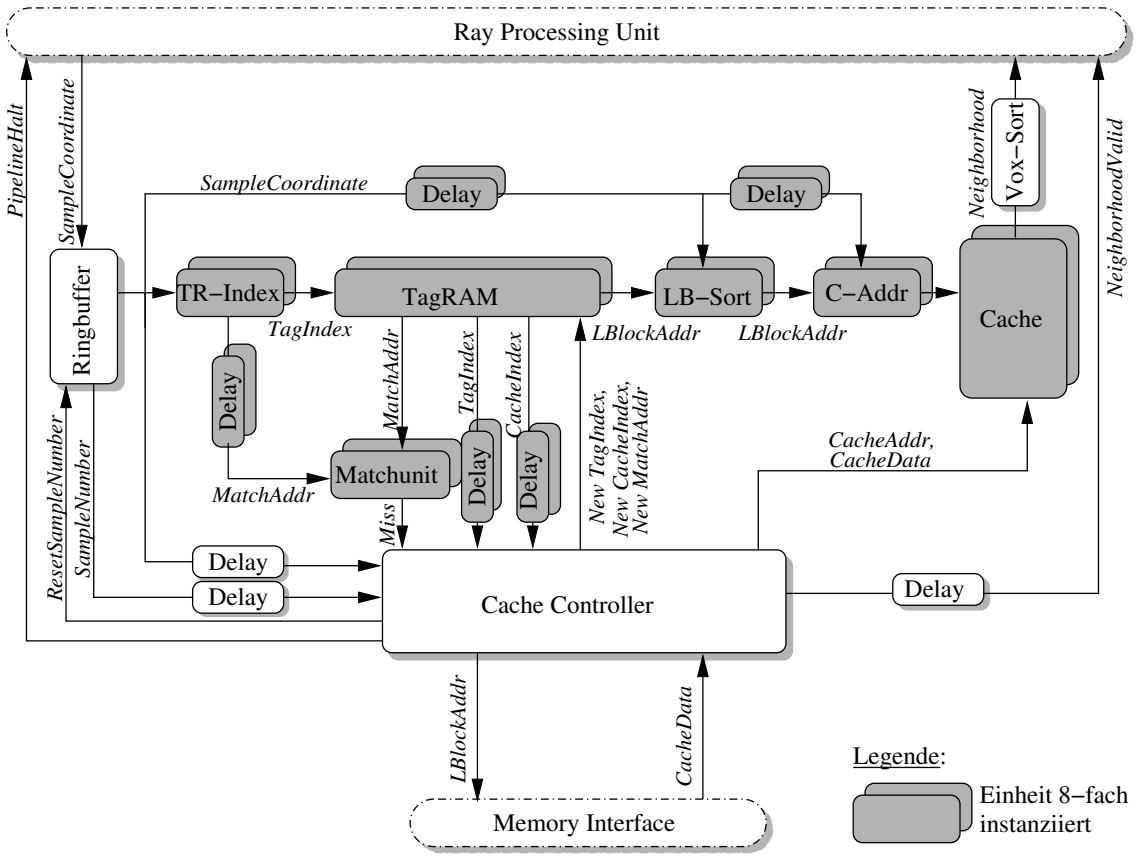


Abbildung 3.7: *Aufbau der Implementierung des VoxelCaches. Grau hervorgehoben sind die achtfach instantiierten Komponenten.*

rungseinheiten (Delays). In Grau gehalten sind die Einheiten, die achtfach repliziert wurden, um eine volle Achterumgebung pro Taktzyklus bearbeiten zu können.

Hauptkomponente ist der Cache-Kontroller. Er überwacht den kompletten VoxelCache und koordiniert den Datentransfer zwischen den einzelnen Einheiten. Er ist auch dafür verantwortlich im Falle einer Anfrage, die nicht bedient werden kann, die notwendigen Daten aus dem externen Speicher zu holen, die Daten in den Cachespeicher zu legen, die notwendigen Tabellen zu aktualisieren und den VoxelCache wieder in einen Zustand zu versetzten, der die aktuelle Anfrage bedient und neue Anfragen entgegen nimmt.

Der VoxelCache ist als Pipeline implementiert und kann in jedem Taktzyklus eine gültige Anfrage entgegen nehmen. Damit stellt sich aber das Problem, dass zum Zeitpunkt an dem eine Anfrage als nicht bedienbar erkannt wird, bereits weitere Anfragen im VoxelCache in Bearbeitung sind. Da die Bearbeitungszeit für eine nicht bedienbare Anfrage von der Verfügbarkeit des externen Speichers abhängt und damit nicht vorhersehbar ist, muss der VoxelCache komplett angehalten werden können. Um das Anhalten des VoxelCache nicht direkt mit dem Ablauf der Visualisierungspipeline zu koppeln wurde zu Beginn des VoxelCache ein Ringpuffer eingefügt, der mehrere Anfragen zwischenspeichern kann. Damit bei einem Anhalten des VoxelCache keine Anfragen verloren gehen, bekommt jeder Eintrag im Puffer

eine fortlaufende Nummer, die es ermöglicht, den VoxelCache mit einer spezifizierten Anfrage über die Nummer neu zu starten. Dieses Verhalten wird im Falle eines Cachemiss genutzt, um nach dem Anhalten und Bereitstellen der Daten den VoxelCache wieder mit der fehlgeschlagenen Anfrage zu starten. Weitere Anfragen, die sich im Falle einer Fehlanfrage bereits in der Pipeline befinden, basieren somit auf veralteten und vielleicht nicht mehr vorhandenen Daten. Daher müssen alle Daten in der Pipeline bis zur Matchunit neu bearbeitet werden. Die Anfragen, die hinter der Matchunit liegen, können noch bearbeitet werden, da garantiert ist, dass die Daten solange noch gültig sind, wie es nötigt ist, um die Daten fertig zu bearbeiten und aus der Pipeline zu spülen.

Da nur eine minimale Menge an Zwischenspeichern für das Vorhalten der Anfragen verwendet werden soll, wurde der Puffer als Ringspeicher implementiert, bei dem nach einer gewissen Zeit alte Einträge durch neue überschrieben werden und auch die Anfragenummern neu vergeben werden. Die Größe des Zwischenspeichers ist durch die Laufzeit der Daten durch den VoxelCache bestimmt, die benötigt wird bis eine etwaige Fehlanfrage bemerkt wird.

Cache-Kontroller

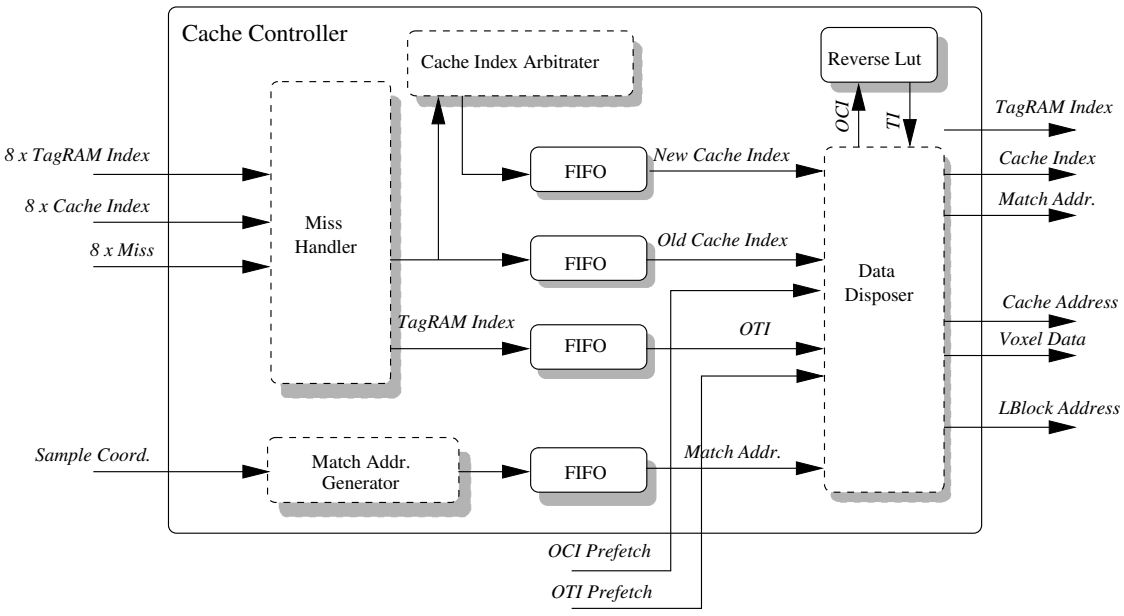


Abbildung 3.8: *Aufbau des Cache-Kontroller, mit Hauptprozessen und Speicherkomponeten. Der Miss-Handler erkennt die etwaigen Cachefehler und erstellt die notwendigen Informationen für die Tag-RAM Einträge und die Rückindizierungtabelle (Reverse LUT). Pro Cachefehler können bis zu acht L-Blöcke neu angefordert werden. Die Ergebnisse des Miss-Handler werden in FIFOs zwischengespeichert. Der Data-Disposer ist für die Aktualisierung der Tag-RAM-Inhalte sowie für die Abspeicherung der Voxeldaten zuständig.*

Der Cache-Kontroller ist die zentrale Koordinationseinheit des VoxelCache. Er

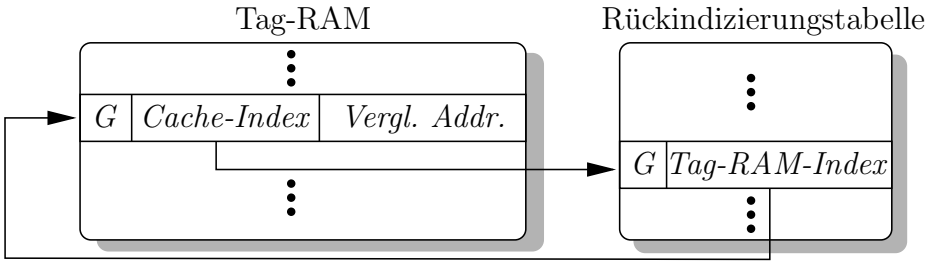


Abbildung 3.9: *Schematischer Zusammenhang zwischen Tag-RAM und Rückindizierungstabelle. Die Rückindizierungstabelle ermöglicht es Referenzen auf einen verwendeten Cache-Indizes im Tag-RAM zu finden. Jeder Eintrag im Tag-RAM und in der Rückindizierungstabelle enthält ein Bit (G), welches die Gültigkeit des Eintrags anzeigt.*

muss Cachefehler detektieren, die notwendigen L-Blöcke ermitteln und anfordern, die vom externen Speicher gelieferten Voxel Daten in den Cachespeicher ablegen und die zugehörigen Verwaltungsdaten aktualisieren. Die Struktur des Cache-Kontrollers ist in Abbildung 3.8 dargestellt. Hauptkomponenten sind drei Prozesse, bestehend aus „Miss Handler“, „Cache Index Arbitrator“ und „Data Disposer“, eine Tabelle zur Rückindizierung der Tag-RAM-Einträge („Reverse LUT“) sowie mehreren FIFOs um die Prozesse zu entkoppeln. Im Folgenden werden die Komponenten in ihrer Aufgabe und Struktur erläutert.

Rückindizierungstabelle

Die Rückindizierungstabelle gibt Auskunft, ob und von welchem Tag-RAM-Eintrag ein Cache-Index referenziert wird. Diese Rückindizierung ist notwendig, um im Falle einer Neuvergabe eines bereits genutzten Cache-Index, einen hängenden Verweis im Tag-RAM zu verhindern. Würde dieser hängende Verweis nicht entfernt, käme es zwangsläufig zu Dateninkonsistenzen. Das Tag-RAM und die Rückindizierungstabelle stellen somit eine eins zu eins Beziehung her. Die Größe der Rückindizierungstabelle ist durch die Anzahl der L-Blöcke vorgegeben, die im Cachespeicher abgelegt werden können. Jeder Eintrag in der Rückindizierungstabelle enthält ein Gültigkeitsbit, welches anzeigt, ob der abgelegte Wert noch oder bereits von einem Tag-RAM referenziert wird, sowie einen Tag-RAM-Index. Abbildung 3.9 verdeutlicht die Beziehung zwischen Tag-RAM-Eintrag und Rückindizierungstabellen-Eintrag.

Miss-Handler

Aufgabe des Miss-Handler ist die Feststellung eines Cachefehlers, das Anhalten des Caches, die Ermittlung der benötigten L-Block Daten, sowie die Rücksetzung des Eingangspuffer auf den Eintrag, der den Fehler verursachte. In Abbildung 3.10 ist der Zustandsgraph des Miss-Handlers aufgezeigt. Im Anfangszustand befindet sich der Prozess im Zustand „Running“. Wird von einem der acht „Matchunits“ ein Cachefehler (Miss) angezeigt, wechselt der Zustand in „Miss Detected“. Hierin werden die aktuellen Daten aller acht „Matchunits“ in einem Fehlervektor gespeichert

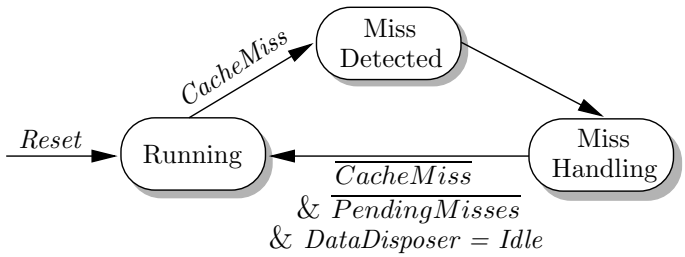


Abbildung 3.10: *Zustandsgraph des Prozesses „Miss-Handler“. Nach Erkennen eines Cachefehlers durch eine Matchunit (CacheMiss gesetzt) werden die Daten für die fehlenden L-Blöcke ermittelt. Nach dem der Data-Disposer-Prozess durch den Zustand „Idle“ wieder anzeigt, dass der Fehler behandelt wurde, die Matchunits keinen Fehler mehr signalisiert und auch kein spekulatives Holen eines L-Block erfolgt, wird der VoxelCache wieder in Gang gesetzt.*

sowie die dazu gehörenden Daten der acht Tag-RAMs bezüglich Tag-RAM-Index und verwendetem Cache-Index in ein Register übernommen. Parallel dazu wird der Teil des Caches bis zur Matchunit angehalten und alle sich davor befindlichen Abtastpunkte als ungültig gekennzeichnet. Die Korrektheit dieser Daten kann durch die beginnende Aktualisierung der Tag-RAMs nicht mehr garantiert werden. Um den Abtastpunkt, der einen Fehler verursachte, mit den aktualisierten Daten neu überprüfen zu können, wird der Ringbuffer am Anfang des Caches auf die Position dieses Abtastpunkts zurückgesetzt. Im nächsten Zyklus wird in den Zustand „Miss Handling“ gewechselt und der Fehlervektor nach der ersten Einheit durchsucht, die einen Fehler vermeldet hat. Ist die Position n des ersten Fehlers bestimmt, werden aus dem Cache-Indexvektor und dem Tag-RAM-Indexvektor der n-te Einträge in FIFOs geschrieben. Als nächstes wird überprüft, ob andere „Matchunits“ ebenfalls zum gerade behandelten Tag-RAM-Index einen Fehler detektiert haben. Die Blöcke der Achterumgebung sind zwangsweise benachbart. Derselbe Tag-RAM-Index kann daher innerhalb einer Achterumgebung nicht von verschiedenen Blöcken generiert werden. Die Eintragungen der entsprechenden Einheit im Fehlervektor können somit ohne weitere Maßnahmen gelöscht werden, da die benötigten Daten bereits angefordert werden. Dies verhindert das redundante Holen derselben L-Blöcke für einen Cachefehler. Sind alle Fehlereinträge aus dem Missvektor gelöscht, sind somit alle benötigten L-Blöcke angefordert. Der Prozess wartet nun bis alle Match-Einheiten keine Fehler mehr anzeigen, was gleichbedeutend mit der kompletten Aktualisierung der Cacheinhalte ist, und wechselt dann zurück in den Zustand „Running“, der auch wieder den Stop des VoxelCache aufhebt.

Cache-Index Generator

Für die angeforderten Voxeldaten muss eine Position im Cachespeicher bestimmt werden, an der die Daten abgelegt werden sollen. Hierzu wird eine einfache „Last In - First Out“-Strategie verwendet. Der zuletzt benutzte Cache Index wird um eins erhöht, solange bis die letzte Position erreicht ist. Danach wird vom Anfang an begonnen, bereits besetzte Positionen zu ersetzten.

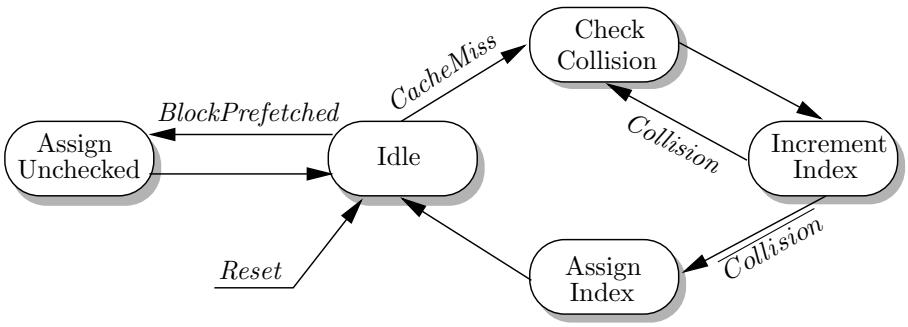


Abbildung 3.11: *Zustandsgraph des Prozess „Cache-Index Generator“. Vor der Nutzung des neuen Cache-Index wird überprüft, ob eine Kollision mit einem aktuell genutzten Cache-Index vorliegt. Für spekulative L-Blöcke ist eine solche Überprüfung nicht möglich und wird daher nicht durchgeführt.*

Um zu verhindern, dass während der Behandlung eines Cachefehlers Daten im Cache überschrieben werden, die für den aktuellen Abtastpunkt benötigt werden, wird der neue Cache-Index so lange erhöht bis er keine Übereinstimmung mehr mit einem Cache-Index besitzt, der zum Zeitpunkt des detektierten Fehlers von einem Tag-RAM angezeigt und von der zugehörigen Matchunit als gültig erkannt wurde (vgl. Abb. 3.11). Für einen spekulativ angeforderten L-Block muss ebenfalls ein neuer Cache-Index generiert werden. Da der Cache im Verlauf des Datentransfers weiter läuft, besteht keine Möglichkeit auszuschließen, dass im Verlauf des Transfers ein Abtastpunkt diese Cache Position benötigt. Es wird daher einfach der Cache-Index um eine Position erhöht und keine weitere Überprüfung vorgenommen. Im schlimmsten Fall führt dies zu einem Cachefehler und die soeben ersetzten Daten müssen wieder neu angefordert werden.

Data-Disposer

Der Data-Disposer ist für die Entgegennahme der neuen Voxel Daten zuständig, muss die entsprechenden Tag-RAM-Einträge ersetzen und die internen Verwaltungsstrukturen aktualisieren. Der Prozess bezieht seine Informationen aus den internen FIFOs die vom Miss-Handler gefüllt werden. Informationen für einen spekulativen L-Block werden durch externe Schnittstellen des Cache-Kontrollers bereitgestellt und stammen direkt von der „Prefetching Unit“. Der Zustandsgraph dieses Prozesses ist in Abbildung 3.13 illustriert. Der Prozess bleibt solange „Idle“ wie die internen FIFOs nicht gefüllt sind und auch kein spekulativer L-Block geholt wurde. Im Fall, dass der Miss-Handler L-Blöcke zur Anforderung vorgesehen und diese Information in die FIFOs geschrieben hat, lädt der Data-Disposer die Information und wechselt über die Zustände „Load“ und „WaitLoad“ in den Zustand „Invalidate Tag-RAM“.

In diesem Zustand werden zunächst die Einträge für den neuen Tag-RAM-Index (NTI) und den alten Tag-RAM-Index (ATI) invalidiert. Zusätzlich wird der Rückindizierungstabellen-Eintrag für den alten Cache-Index (ACI) als ungültig gekennzeichnet (vgl. Abb. 3.12). Dies ist notwendig, da im Falle eines spekulativen Blocks

der VoxelCache weiterläuft und die Konsistenz der Daten vor dem Abspeichern der Voxeldaten sichergestellt werden muss. Im nächsten Zustand „UpdateCache-Slice“ werden die vom externen Speicher bereitgestellten Daten an die neue Cache Position (NCI) geschrieben. Um den neuen L-Block auch im Tag-RAM wieder zu spiegeln, muss in der neuen Position der Verweis auf die neuen Cacheschicht eingetragen werden. Die Zuordnung von Cacheschicht zu Tag-RAM-Eintrag wird in der Rückindizierungstabelle über den neue Tag-RAM-Index vermerkt.

3.6 Performanz und Ressourcenverbrauch

Eine Ziel des VoxelCaches ist die einfache und schnelle Integration verschiedenster Speichertechnologien in die Hardwarearchitektur. Für die Ermittlung der Leistungsfähigkeit des VoxelCache werden folgende drei Speicherarten betrachtet: RDRAM [73], DDRAM [37] und SDRAM [36]. Neben diesen Speicherarten werden auch noch zwei Bustypen berücksichtigt, die als Schnittstelle zum externen Speicher des Host-Computer genutzt werden. Als Vertreter dieser Art wurden PCI [81] und PCI-X [82] gewählt. In Tabelle 3.1 sind die, für diese Untersuchung relevanten, Kennzahlen der Speicher- und Busarten aufgeführt. Um die Leistungsfähigkeit der Speicherarten zu messen, wurden die Charakteristika auf eine Referenzpipeline, die mit 133 MHz läuft, genormt. Die Latenz in Referenztaktzyklen l und die übertragene Anzahl von Bytes pro Referenztaktzyklus b_c sind in Klammern angegeben.

Im Gegensatz zur VHDL-Simulation konnte bei der C++-Simulation das Verhalten des Speicherinterface nicht exakt modelliert werden. Es wurden daher Strafzyklen eingerechnet, die die Wechselwirkung zwischen Cache und Visualisierungspi-

```
1:  case DataDiposerState is
2:    when InvalidateTag-RAM ⇒
3:      Tag − RAM[NTI] ⇐ Invalid
4:      Tag − RAM[RevLut[NCI]] ⇐ Invalid
5:      RevLut[ACI] ⇐ Invalid
6:    when UpdateCacheSlice ⇒
7:      for all 0 ≤ i < LBlockSize do
8:        CacheSlice[NCI+i] ⇐ VoxelData[i]
9:      end for
10:   when UpdateTag-RAM ⇒
11:     Tag − RAM[NTI] ⇐ NCI
12:     RevLut[NCI] ⇐ NTI
13: end case
```

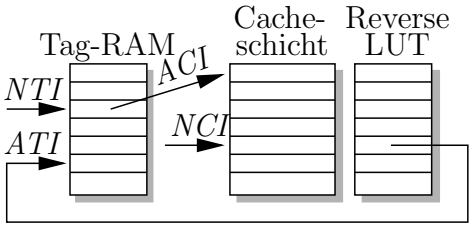


Abbildung 3.12: *Pseudocode zur Aktualisierung der Tag-RAM, Cachespeicher und Rückindizierungstabelle. Betrachtet sind nur die relevanten Zustände des Data-Disposers. Rechts ist der schematische Zusammenhang zwischen Tag-RAM-Indizes (Neuer Tag-RAM-Index = NTI, Alter Tag-RAM-Index = ATI), Cachespeicher und Rückindizierungstabelleindizes (Neuer Cache-Index = NCI, Alter Cache-Index = ACI) illustriert.*

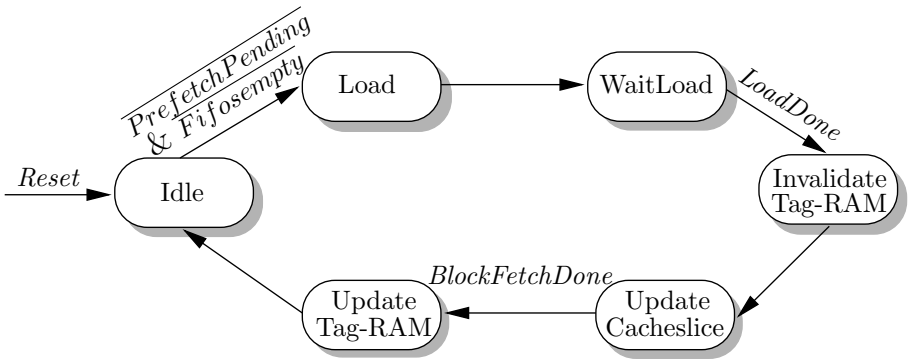


Abbildung 3.13: *Zustandsgraph des Prozesses „Data Disposer“. Nach dem Signalisieren, dass ein neuer L-Block angefordert wurde (spekulativ oder durch Cachefehler), lädt der Prozess die entsprechenden Informationen. Um die Datenkonsistenz zu gewährleisten werden zunächst die entsprechenden alten Informationen als ungültig markiert, dann die neuen Daten im Cachespeicher abgelegt und schlussendlich die Tag-RAM Einträg aktualisiert.*

Speicherart	Latenz (l)	Bandbreite (b_c)
RDRAM, 800 MHz, 16 Bit	50 ns (7)	1,6 GB/s (12)
DDRAM, 133 MHz, 32 Bit	60 ns (8)	1 GB/s (8)
SDRAM, 133 MHz, 32 Bit	60 ns (8)	1 GB/s (4)
PCI-X, 133 MHz, 64 Bit	>120 ns (16)	4,3 GB/s (32)
PCI, 33 MHz, 32 Bit	>500 ns (64)	133 MB/s (1)

Tabelle 3.1: *Kenngrößen der verwendeten Speicher- und Bustypen. Die auf die Taktrate der Referenzpipeline genormt Werte sind in Klammern angegeben.*

peline im Falle einer Cacheanfrage, die nicht bedient werden kann, in die Statistik eingehen ließen. Dieses Verhalten wurde durch Leerzyklen p_m beschrieben, die auf die Gesamtzyklen der Pipeline hinzugerechnet werden. p_m besteht dabei aus der Summe der Latenzzyklen l und der Anzahl an Zyklen, die benötigt werden um einen L-Block aus dem externen Speicher in den Cachespeicher zu transferieren. L-Blöcke, die spekulativ in den Cache geholt werden, werden nicht zu den Gesamtzyklen gezählt. Der externe Speicher ist während deren Transferzeit als nicht verfügbar vermerkt und steht somit für etwaige reguläre L-Blocktransfers nicht zur Verfügung. Die Zyklen, die der VoxelCache warten muss, bis seine Fehlanfrage behandelt wird, werden auf die Gesamtzyklenzahl angerechnet. Des Weiteren wird angenommen, dass alle Speicher- und Busarten über Burst Mechanismen verfügen und somit neue Speicherzugriffe noch während des aktuellen Datentransfers aufgesetzt werden können.

Zur Messung der Performanz wurde ein völlig transparenter Datensatz mit einer Größe von 136^3 Voxel benutzt. Diese Größe des Datensatzes wurde gewählt, da das Tag-RAM des VoxelCache in der Hauptachsenrichtung mit 5 Bit adressiert wird und daher in diese Richtung 2^5 verschiedene L-Blöcke, respektive 128 Voxel adressiert werden können. Bei einem Datensatz mit weniger als 128 Voxel existiert daher die Möglichkeit, dass ein Sehstrahl, der neu in das Volumen eintritt, bereits seine Daten

im VoxelCache verfügbar hat. Ab einer Volumengröße von mehr als 128 Voxel kann dieser Effekt nicht mehr auftreten. Die gewählte Transparenz des Volumens soll verhindern, dass Optimierungseffekte, wie „Early-Ray-Termination“, das Ergebnis verfälschen.

Für die Generierung der Statistik wurden folgende Größen während der Simulation erhoben: Die Anzahl der erfolgreichen Cacheanfragen (*Cache Hits*) c_h, die Anzahl der nicht bedienbaren Cacheanfragen (*Cache Misses*) c_m, die Gesamtzahl der in den Cachespeicher geladenen Voxel v_f, die Anzahl der spekulativ geladenen Voxel v_p, die verwendeten Bits pro Voxel v_b, sowie die Gesamtzahl der Taktzyklen c_p, die zum Visualisieren eines Bildes benötigt werden. Die Kenngrößen l und b_c der Speicher- bzw. Busarten wurden hierbei aus Tabelle 3.1 entnommen und der Simulation als Startparameter mitgegeben. Anhand dieser Statistik können folgende, aussagekräftige Kennzahlen über den VoxelCache bestimmt werde.

- Der Anteil der erfolgreichen Anfragen an der Gesamtzahl der Anfragen:

$$r_h = \frac{c_h}{c_h + c_m} \; .$$

- Die durchschnittliche Zugriffszeit in Taktzyklen:

$$t_a = \frac{c_h + c_m \cdot l}{c_h + c_m} \; .$$

- Die Auslastung des Speicherbusses im Verhältnis zur maximalen Bandbreite:

$$u_{mb} = \frac{v_f \cdot v_b}{c_p \cdot b_c} \; .$$

- Die Auslastung der Visualisierungspipeline:

$$u_p = \frac{c_h}{c_p} \; .$$

Speichertyp	r_h / %	t_a / *Zyklen*	u_{mb} / %	u_p / %
RDRAM	97,9 (99,3)	2,1 (1,08)	20,3 (31,8)	47,6 (91,8)
DDRAM	97,9 (99,3)	2,1 (1,11)	30,4 (45,2)	47,6 (90,6)
SDRAM	97,7 (98,8)	2,6 (1,27)	49,5 (76,8)	38,7 (80,2)
PCIX	97,8 (99,3)	2,3 (1,15)	7,1 (44,9)	44,5 (89,0)
PCI	97,5 (97,8)	6,9 (3,84)	67,9 (77,4)	14,5 (32,5)

Tabelle 3.2: *Simulationsergebnisse der VHDL-Implementierung in Bezug auf erfolgreiche Anfragen (r_h), durchschnittliche Zugriffszeit (t_a), Busauslastung (u_{mb}) und Pipelineauslastung (u_p). Zum Vergleich sind in Klammern die jeweiligen Ergebnisse der C++-Simulation angegeben.*

- Die Anzahl an transferierten Voxel pro Abtastpunkt:

$$v_s = \frac{c_h + c_m}{v_f} .$$

Anhand vier repräsentativer Bildkonfigurationen wurde die Cachestatistik ermittelt. Mit jeweils Parallelprojektion und perspektivischer Projektion wurde ein Bild entlang der z-Richtung dargestellt, wobei die Bildebene direkt auf der x-y-Grundfläche des Volumens liegt. Daneben wurde eine diagonale Abbildung mit einem Winkel von jeweils 45° zwischen Blickrichtung und Hauptachsen des Volumens verwendet, ebenfalls in beiden Projektionsarten. Dabei wurde sichergestellt, dass alle Sehstrahlen des Bildes in das Volumen eintreten. Obwohl man durch die divergierenden Sehstrahlen der perspektivischen Projektion und der damit einhergehenden schlechteren Kohärenz ein ungünstigeres Cacheverhalten erwarten würde, wurde dieser Effekt durch die an der Seite austretenden kürzeren Sehstrahlen und die daraus folgenden selteneren Ersetzungen der L-Blöcke kompensiert. Zwischen perspektivischer und paralleler Projektion konnte daher kein signifikanter Unterschied in der Performanz festgestellt werden. Auch zwischen den orthogonalen und diagonalen Blickrichtungen lagen die Unterschiede der Ergebnisse im Prozentbereich, so dass im Folgenden immer nur die Resultate der orthogonalen Blickrichtung mit perspektivischer Projektion aufgeführt werden.

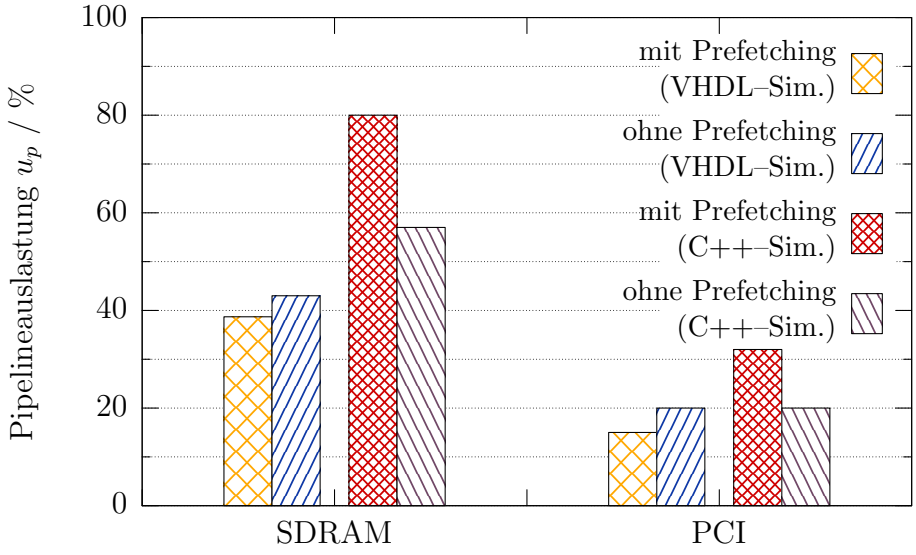


Abbildung 3.14: *Gegenüberstellung der per VHDL-Simulation ermittelten und per C++-Simulation prognostizierten Pipelineauslastung für zwei verschiedene Speicherarten mit und ohne spekulativem Prefetching von L-Blöcken.*

In Tabelle 3.2 sind für alle untersuchten Speicher- und Bustypen die Ergebnisse der zyklenakkuraten VHDL-Simulation aufgeführt. In Klammern sind zum Vergleich die Ergebnisse der C++-Simulation angeben. Deutlich ist zu erkennen, dass die Trefferrate r_h nur minimal unter der C++-Simulation liegt und für alle Typen

Abtastrate x-, y-Richtung	*Abtastrate z-Richtung*	u_p / %	r_h / %	$u_m b$ / %	t_a / *Zyklen*	v_s / *Voxel*
0,5	1	18,8	91,3	17,7	5,3	7,5
0,5	2	31,6	95,4	14,9	3,2	3,8
1	1	31,3	95,6	15,4	3,2	3,9
1	2	47,7	97,8	11,7	2,1	2,0
2	1	50,7	98,11	11,4	2,0	1,8
2	2	67,3	99,0	7,6	1,5	0,9
2	4	80,5	99,5	4,5	1,2	0,5
2	10	91,2	99,8	2,0	1,1	0,2

Tabelle 3.3: *Vergleich von Unter- und Überabtastung in x-, y-, z-Richtung anhand des DDRAM Speichers.*

fast identische Werte erreicht. Auffällig hingegen ist der deutliche Unterschied in der Bus- und Pipelineauslastung. Hier bleiben die Werte signifikant hinter den vorhergesagten Werten der C++-Simulation zurück. Um dieses Phänomen eingehender zu untersuchen, wurde für jeweils ein Speicher und Bustyp die Pipelineauslastung mit und ohne Prefetching durchgeführt (vgl. Abb. 3.14). Zu erkennen ist, dass im Fall der C++-Simulation das spekulative Holen der L-Blöcke die Pipelineauslastung erheblich steigert, während es diese bei der VHDL-Simulation unerwartet reduziert. Ohne Prefetching liegen die Werte deutlich weniger auseinander oder sind für den PCI-Bus sogar identisch. Dies bedeutet, dass das Prefetching-Schema nur in der C++-Simulation eine Steigerung bringt. Zu erklären ist dies mit dem Fakt, dass in der C++-Simulation zwar die Latenz des Speicherzugriffs berücksichtigt wurde, die inhärente Latenz des VoxelCaches, da nicht bekannt, nicht eingerechnet wurde. Die tatsächliche Latenz des VoxelCache liegt aber bei der VHDL-Implementierung in derselben Größenordnung wie der Speicherzugriff. Dies führt dazu, dass das Holen der spekulativen L-Blöcke häufig in den Zeitraum eines Cachefehlers fällt und die Pipeline angehalten bleibt bis der Transfer des spekulativen L-Blocks abgeschlossen ist und die notwendigen Daten für den Cachemiss geholt werden können. Als weiterer Effekt ist zu beachten, dass die ersten Ergebnisse der C++-Simulation mit einer niedrigen Strahlabtastrate vorgenommen wurdne, da eine höhere Abtastrate keine signifikante Steigerung erbrachte, und auf diese Weise erheblich Simulationszeit gespart wurde.

Da eine Hardwareimplementierung einer Volumenvisualisierung nur im Bereich der hochqualitativen Bildgebung Sinn macht, ist eine hohe Abtastrate erforderlich. Nach dem Abtasttheorem von Nyquist [65] ist mindestens eine zweifache Überabtastung erforderlich um ein Signal korrekt rekonstruieren zu können. Daher wurden in Tabelle 3.3 für einen ausgewählten Speichertyp die Auswirkungen der Abtastrate auf die Ergebnisse untersucht. Wie für einen Cache zu erwartet, werden die Ergebnisse mit höherer Abtastrate deutlich besser. Ab einer 2-fachen Überabtastung in alle Raumrichtungen steigt die Pipelineauslastung über 67 %. Gut ist zu erkennen

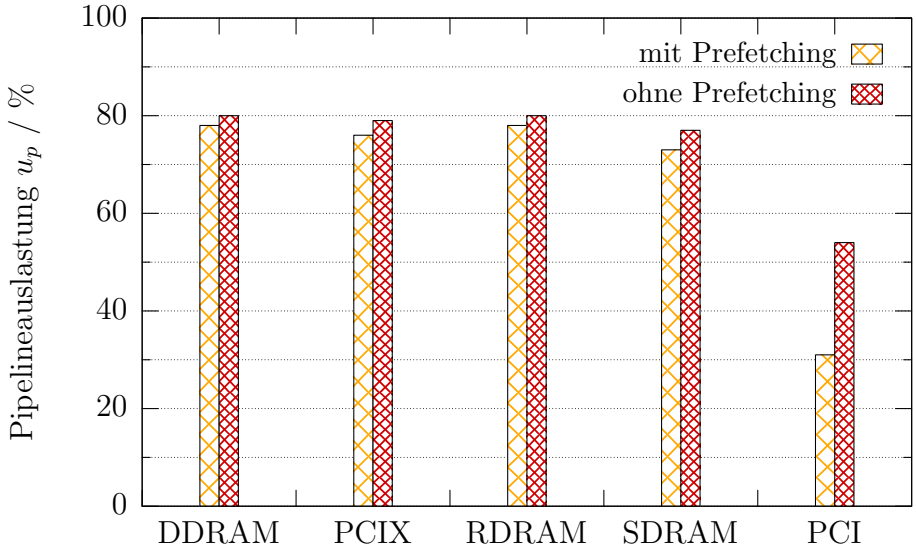


Abbildung 3.15: *Pipelineauslastung u_p für die untersuchten Speicherarten bei zweifacher Überabtastung in x- und y-Richtung und vierfacher Überabtastung in z-Richtung.*

wie minimale Änderungen der Trefferrate die Pipelineauslastung deutlich erhöhen.

Die VHDL-Simulation wurde somit für alle Typen mit einer 2-fachen Überabtastung in x-, y-Richtung und einer 4-fachen Überabtastung in z-Richtung ohne Prefetching wiederholt. Die Ergebnisse sind in Abbildung 3.15 aufgetragen. Mit einer Pipelineauslastung um die 80 % liegen die Werte im erwarteten Rahmen und zeigen, dass der VoxelCache die Trennung von Visualisierungspipeline und Speicher erfüllen kann ohne die Pipelineauslastung des Systems übermäßig zu reduzieren.

Um einen Überblick über die benötigten Hardware-Ressourcen des VoxelCache zu gewinnen, wurde der vollständige VoxelCache synthetisiert (vgl. Tab. 3.4). Als Zielplattform wurden zwei Virtex-FPGA gewählt, wobei der Virtex-II 4000 momentan als Basis für die nächste Generation des VIZARD II genutzt wird. Der VoxelCache erreicht mit einer maximalen Taktfrequenz von 134 MHz die anvisierte Geschwindigkeit der Referenzpipeline. Die benötigten Ressourcen von ca. 15 % sind so gering, dass neben der VIZARD II-Pipeline der VoxelCache leicht auf den FPGA-Chip passt.

Der VoxelCache trennt somit effektiv die Visualisierungspipeline von der verwendeten Speicherart. Der damit einhergehende Performanzverlust von ca. 20 % der Pipelineauslastung kann durch die höhere Taktfrequenz der FPGA-Komponenten, die nun ohne aufwendige Eigenentwicklungen nutzbar sind, leicht ausgeglichen werden. Eine Synthese für den Virtex-II 4000 der neuen VIZARD II-Plattform erbrachte eine maximale Taktfrequenz der Visualisierungspipeline von 133 MHz, ohne irgendwelche Modifikationen vornehmen zu müssen. Im Vergleich dazu erreicht die aktuelle Pipeline des VIZARD II-Prototypen 70 MHz auf einem Virtex-E 1000.

Durch den on-chip Cache konnte zusätzlich die erforderliche Bandbreite zum on-board Voxelspeicher auf unter 5 % der verfügbaren maximalen Bandbreite gesenkt werden, da für jeden Abtastpunkt im Schnitt nur noch 0,5 Voxel in den Cache

FPGA Nutzung für:	*Virtex-II 2V4000ff1152*			*Virtex-II 2V8000ff1517*		
Ressource	*Benutzt*	*Verfügbar*	*Nutzung*	*Benutzt*	*Verfügbar*	*Nutzung*
IOs	0	0	0,00 %	0	0	0,00 %
Global Buffers	1	16	6,25 %	1	16	6,25 %
Function Generators	6 461	46 080	14,02 %	4 105	93 184	4,41 %
CLB Slices	3 231	23 040	14,02 %	2 053	46 592	4,41 %
Dffs or Latches	4 024	48 552	8,29 %	2 944	96 508	3,05 %
Block RAMs	16	120	13,33 %	16	168	9,52 %
Block Multipliers	0	120	0,00 %	0	168	0,00 %
Block Multiplier Dffs	0	4 320	0,00 %	0	6 048	0,00 %

Tabelle 3.4: *Syntheseergebnisse des VoxelCaches für zwei FPGA-Typen. Gegenübergestellt ist der momentan verwendete Virtex-II 4000 und der aktuelle Virtex-II 8000.*

nachgeladen werden müssen (vgl. Tab. 3.3). Dies ist von besonderer Bedeutung im Hinblick auf die in den nächsten Kapiteln erfolgende Integration der dynamischen Speicherverwaltung des Multiskalenmodells. Noch während der Berechnung des Bildes steht somit genügend Bandbreite zur Verfügung um Daten im Voxelspeicher auszutauschen oder nachzuladen.

Kapitel 4

Wavelettransformation in Hardware

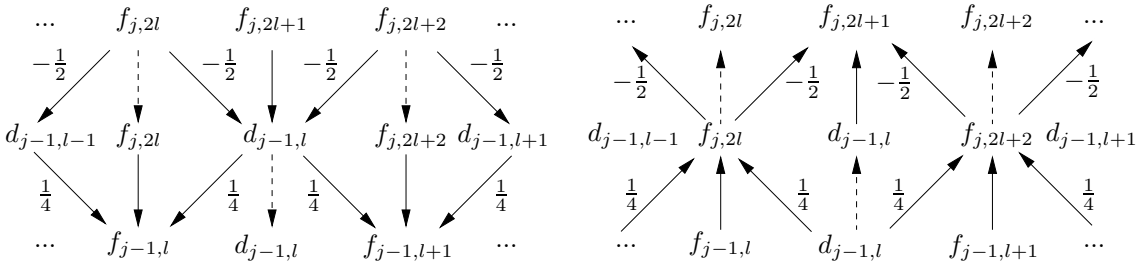
Durch die im vorangegangenen Kapitel vorgestellte Entkopplung des Voxelspeichers von der Bildgebungspipeline bietet sich die Möglichkeit neuartige Repräsentationen des Volumens im Speicher ablegen zu können. Dies ermöglicht auch sehr große Volumendatensätze darstellen zu können. In diesem Kapitel sollen daher die theoretischen Grundlagen der Wavelets erläutert werden, die dem Multiskalenmodell der Visualisierungspipeline zugrunde liegt. Des Weiteren wird eine Hardwareimplementierung der Transformation vorgestellt.

Wavelets sind eine seit den 1980er Jahren in der Computergraphik bekannte mathematische Methode zur Signalanalyse. Als Alternative zur Fouriertransformation haben sie in der allgemeinen Signaltheorie der Mathematik und Physik eine viel ältere Geschichte, die bis zum Ende des 19. Jahrhunderts zurückreicht [83]. Diese allgemeine Theorie zur hierarchischen Zerlegung von Funktionen findet in der Computergraphik in allen Bereichen Anwendung, die sich mit der Analyse, Visualisierung und Manipulation von großen Datenmengen beschäftigen. Die Volumengraphik ist daher ein Teilgebiet, das sich ebenfalls gut für diese Methodik eignet.

Der Grundgedanke aller hierarchischen Methoden ist, Funktionen durch eine Anzahl von Koeffizienten zu beschreiben, die jede eine gewisse Information über die Ursprungsfunktion beitragen. Diese Information bezieht sich sowohl auf ihre Position wie auch auf ihre Frequenz. Im Gegensatz dazu bieten z. B. die Koeffizienten der Fourieranalyse ausschließlich Information über eine spezielle Frequenz, ohne irgendwelche Information über andere Frequenzen zu vermitteln. Die Wavelets bieten darüber hinaus noch weitere Vorzüge: Sie besitzen meist eine lineare Komplexität, eignen sich durch ihre meist dünne Besetzung der Koeffizienten hervorragend für Kompression und lassen sie sehr einfach an eine Vielzahl von Funktionen anpassen [83].

4.1 Wavelettransformation

Beim Waveletschema wird das Signal in eine Anzahl von Basisfunktionen zerlegt, die alle eine translierte und skalierte Variante eines so genannten Mother-Wavelets darstellen. Ein einfaches Mother-Wavelet ist z. B. durch das Haar-Wavelet für einen



(a) Transformation *(b) Rücktransformation*

Abbildung 4.1: *Schema des Linearen Wavelet Lifting: Die Stufe j stellt die Ausgangsmenge dar. Das Schema kann in den nächsten Stufen mit einer Schrittweite von 2, 4, 8, ... wiederholt werden. Alle d-Koeffizienten werden dabei ausgelassen. Die Transformation benötigt keinen zusätzlichen Speicher, alle Koeffizienten werden im Originalspeicherbereich gelesen und geschrieben.*

Vektorraum V^j gegeben:

$$\phi_i^j := \phi(2^j x - 1) \qquad i = 0, ..., 2^j - 1$$

mit

$$\phi(x) = \begin{cases} 1 & : \quad x < 0 \leq \frac{1}{2} \\ -1 & : \quad \frac{1}{2} \leq x < 1 \\ 0 & : \quad \text{sonst} \end{cases} .$$

In dieser Arbeit werden als Basis biorthogonale Spline-Wavelets [16] verwendet. Da das Gebiet der Wavelets zu umfangreich ist um an dieser Stelle ausführlich behandelt zu werden, wird im Folgenden ausschließlich auf die unmittelbar verwendeten Methoden eingegangen. Für eine ausführliche Erläuterung von Wavelets und deren Anwendung wird auf Stollnitz et al. [83] und Graps [29] verwiesen.

4.1.1 Wavelet-Lifting-Schema

Um die Transformation unter Umgehung des Frequenzraums rein im Ortsraum vornehmen zu können, wird eine spezielle Variante der Wavelets, das Lifting, verwendet. Hierbei handelt es sich um eine Transformation, die aus mehreren Gründen für eine Hardware Implementierung hervorragend geeignet ist. Das Lifting kann ohne zusätzlichen Speicherbedarf im Speicherraum der Originaldaten erfolgen und die inhärente Parallelität erlaubt einen SIMD-Ansatz. Des Weiteren ist die Inverstransformation ein einfaches Abarbeiten der ursprünglichen Transformationsschritte in umgekehrter Reihenfolge [86].

Gegeben sei ein Signal f_j mit j Punkten. Dieses Signal soll im Folgenden in ein gröberes Signal und ein detailliertes Signal transformiert werden:

$$\begin{aligned} f_{j-1} &:= \text{Durchschnitts-Koeffizienten,} \\ d_{j-1} &:= \text{Detail-Koeffizienten.} \end{aligned}$$

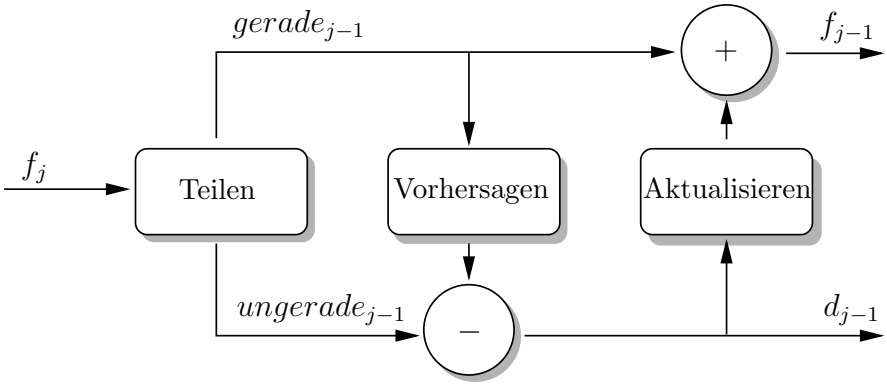


Abbildung 4.2: *Wavelettransformation des Lifting-Schemas: Nach dem Teilen (T) der Ausgangsmenge wird die ungerade Menge durch die geraden Partner vorhergesagt (V). Das gröbere Signal wird durch die Aktualisierung (A) erzeugt.*

Dies erfolgt in drei Schritten. Der erste Schritt besteht darin, die Ausgangsmenge durch eine Operation T in zwei neue, gleichgroße Mengen zu teilen. Der erste Teil enthält alle Punkte mit geradem Index j und der zweite Teil alle mit ungeradem Index j:

$$(gerade_{j-1}, ungerade_{j-1}) := T(f_j) \ .$$

Im zweiten Schritt wird versucht eine Korrelation zwischen der geraden und der ungeraden Teilmenge herzustellen. Hierzu wird versucht aus den geraden die ungeraden Koeffizienten anzunähern. Da die Punkte der Teilmenge abwechselnd in der Ausgangsmenge liegen, sollte an jedem Punkt eine starke Korrelation zwischen den Teilmengen bestehen, für den Fall, dass das Ausgangssignal eine gewisse lokale Korrelation besitzt. Anders ausgedrückt; jeder Punkt der einen Teilmenge sollte sich in diesem Fall aus seinen Nachbarn der anderen Teilmenge gut vorhersagen lassen:

$$d_{j-1,l} \ = \ f_{j,2l+1} - \frac{1}{2}(f_{j,2l} + f_{j,2l+2}) \ .$$

Der zweite Term kann hierbei als Vorhersage V interpretiert werden:

$$d_{j-1} \ = \ ungerade_{j-1} - V(gerade_{j-1}) \ .$$

Der dritte Schritt besteht in der Aktualisierung A der neuen Punkte f_{j-1}, der sicherstellen soll, dass die Grundbedingung des Tiefpass erhalten bleibt. Diese besagt, dass das gröbere Signal denselben Mittelwert F wie das feinere Signal besitzen soll:

$$F = 2^{-j} \sum_{l=0}^{2^j - l} f_{j,l} \ .$$

Diese Bedingung wird sichergestellt durch:

$$f_{j-1,l} = f_{j,2l} + \frac{1}{4}(d_{j-1,l-1} + d_{j-1,l}) \ .$$

Wird der zweite Term als Aktualisierung A ersetzt, ergibt sich:

$$f_{j-1,l} = gerade_{j-1} + A(d_{j-1}) \ .$$

Der Ablauf des Lifting-Schemas kann daher in die Schritte *Teilen*, *Vorhersagen* und *Aktualisieren* aufgeteilt werden (vgl. Abb. 4.2).

$$(gerade_{j-1}, ungerade_{j-1}) := T(f_j) \tag{4.1}$$
$$ungerade_{j-1} -= V(gerade_{j-1}) \tag{4.2}$$
$$gerade_{j-1} += A(d_{j-1}) \tag{4.3}$$

Rücktransformation

Die Rücktransformation erfolgt mit denselben Schritten wie die Transformation. Sie werden allerdings in umgekehrter Reihenfolge durchgeführt (vgl. Abb. 4.3). Als erstes wird die Aktualisierung rückgängig gemacht:

$$gerade_{j-1} = f_j - A(d_{j-1}) \ .$$

Danach wird die Vorhersage zurück genommen:

$$ungerade_{j-1} = d_{j-1} + A(gerade_{j-1}) \ .$$

Zum Schluss werden die Teilmengen wieder in abwechselnder Reihenfolge zu einer Menge zusammengefasst (Z). Wenn nun die geraden Punkte die Durchschnitte und die Ungeraden die Differenzen enthalten, kann die Rücktransformation in folgender kompakter Darstellung geschrieben werden:

$$gerade_{j-1} -= A(d_{j-1}) \tag{4.4}$$
$$ungerade_{j-1} += V(gerade_{j-1}) \tag{4.5}$$
$$f_j := Z(gerade_{j-1}, ungerade_{j-1}) \ . \tag{4.6}$$

Der kompakte Träger von 3 und 5 Koeffizienten für die geraden bzw. ungeraden Koeffizienten kann in Abbildung 4.1(b) abgelesen werden. So sind z. B. für die Rekonstruktion des Koeffizienten $f_{j,2l}$ die Koeffizienten $f_{j-1,2}$,$d_{j-1,l}$ und $d_{j-1,l-1}$ erforderlich.

4.1.2 Ganzzahl-Lifting-Schema

Die bisherigen Berechnungen waren Abbildungen $\mathbb{R} \to \mathbb{R}$. Für die Umsetzung der Algorithmen in Hardware ist dieser Wertebereich aber problematisch und auch unnötig, da die Ausgangsdaten im vorliegenden Fall natürliche Zahlen sind. Es wird daher eine Variation, das Ganzzahl-Lifting-Schema, verwendet, welches eine Abbildung $\mathbb{N} \to \mathbb{Z}$ erlaubt [14]. Die Verwendung von Ganzzahlen vereinfacht eine Implementierung erheblich, ohne die algorithmische Komplexität zu steigern.

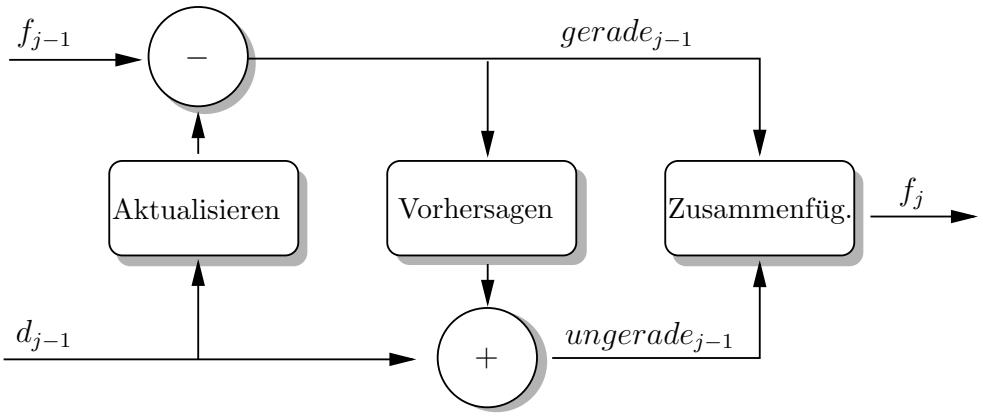


Abbildung 4.3: *Rücktransformation des Lifting-Schemas: Zuerst muss die Aktualisierung rückgängig gemacht werden, um die geraden Punkte zu bekommen. Danach werden die Vorhersagen abgezogen, um die ungeraden Punkte zu erhalten. Abschließend werden die Teilmenge im Reißverschlußverfahren zusammen gefügt.*

Die f- und d-Koeffizienten werden hierbei durch folgende, leicht modifizierte Transformation berechnet:

$$d_{j-1,l} = f_{j,2l+1} - \lfloor \frac{f_{j,2l} + f_{j,2l+2}}{2} + \frac{1}{2} \rfloor \tag{4.7}$$

$$f_{j-1,l} = f_{j,2l} + \lfloor \frac{d_{j-1,l-1} + d_{j-1,l}}{4} + \frac{1}{2} \rfloor \ . \tag{4.8}$$

Die Rücktransformation ist gegeben durch:

$$f_{j,2l+1} = d_{j-1,l} + \lfloor \frac{f_{j,2l} + f_{j,2l+2}}{2} + \frac{1}{2} \rfloor \tag{4.9}$$

$$f_{j,2l} = f_{j-1,l} - \lfloor \frac{d_{j-1,l-1} + d_{d_j-1,l}}{4} + \frac{1}{2} \rfloor \ . \tag{4.10}$$

Diese modifizierten Transformationen liefern trotz der Abrundungsoperation nach der Rücktransformation wieder exakt dieselben Koeffizienten und können daher zur verlustfreien Transformation verwendet werden.

Die bisher vorgestellte Wavelettransformation ist eine eindimensionale Transformation. Sie kann aber ohne weiteres auf drei Dimensionen erweitert werden. Hierfür gibt es zwei Verfahren: die Standard-Zerlegung und die Nichtstandard-Zerlegung. Der Unterschied besteht in der Reihenfolge, in der die eindimensionalen Zerlegungen in x-, y-, z-Richtung durchgeführt werden.

Bei der Standard-Zerlegung wird zuerst die eindimensionale Wavelettransformation in x-Richtung angewendet, bis in diese Richtung der abschließende Durchschnittswert und alle Detail-Koeffizienten bestimmt sind. Dies wird danach für die y-Richtung und im Folgenden für die z-Richtung durchgeführt.

Im Fall der Nichtstandard-Zerlegung werden die einzelnen Transformationen in den Raumrichtungen verschränkt. Dies bedeutet, nach der Zerlegung in x-Richtung,

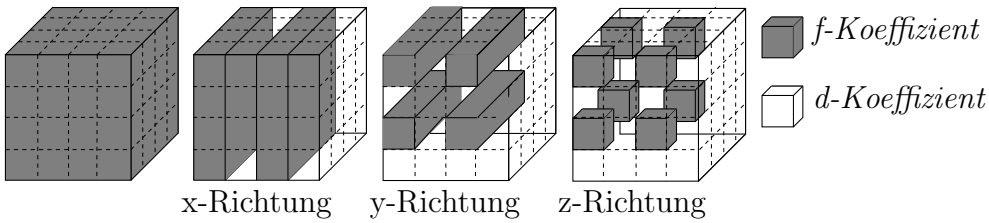


Abbildung 4.4: *Nichtstandard-Zerlegung der dreidimensionalen Wavelettransformation.*

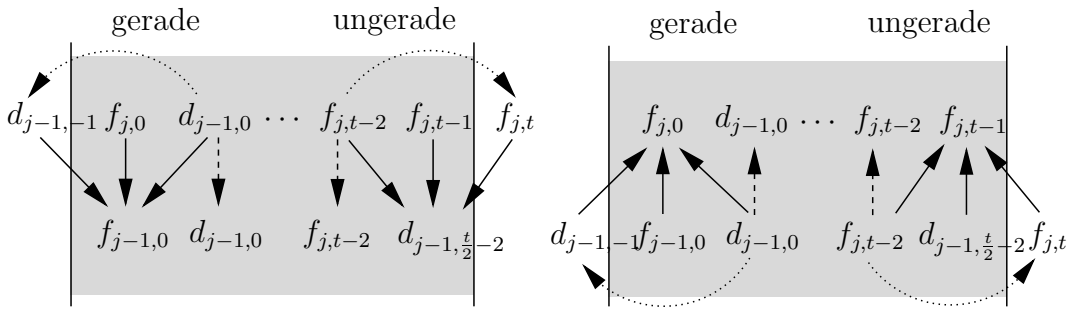
folgt eine Zerlegung in y-Richtung, gefolgt von einer in z-Richtung. Dies wird solange wiederholt, bis in alle Raumrichtungen die vollständigen Transformationen durchgeführt wurden. Die Schrittweite der Transformation wird dabei immer verdoppelt, um die Detail-Koeffizienten zu überspringen und ausschließlich auf den Durchschnitts-Koeffizienten zu arbeiten (vgl. Abb. 4.4).

Hierarchische Zerlegung

Das Waveletschema ermöglicht eine hierarchische Darstellung der Volumendaten. Hierzu wird die Nichtstandard-Zerlegung in einer Variante verwendet, um eine einfache Darstellung der verschiedenen Hierarchieebenen zu gewährleisten. Die Ausgangsdaten werden in Subblöcke der Größe t^3 zerteilt und auf jeden dieser Subblöcke einmalig die Nichtstandard-Zerlegung angewendet. Als Ergebnis entstehen so $(\frac{t}{2})^3$ Durchschnitts-Koeffizienten und $t^3 - (\frac{t}{2})^3$ Detail-Koeffizienten. Die Detail-Koeffizienten werden gegebenenfalls komprimiert und abgespeichert.
In der nächsten Hierarchiestufe werden acht benachbarte $(\frac{t}{2})^3$ Durchschnitts-Koeffizienten wieder zu einem t^3 Subblock zusammengefasst und erneut transformiert. Dieser Vorgang wird solange wiederholt, bis ein finaler t^3 Subblock übrig bleibt. Dieser Subblock stellt das tiefpassgefilterte Äquivalent des gesamten Datensatzes dar. Zusammen mit allen gespeicherten Detail-Koeffizienten enthält er alle Informationen die benötigt werden, um das Ausgangssignal sowie alle Subblöcke jeder Hierarchiestufen wieder herzustellen. Die Transformation ist bisher verlustfrei und benötigt exakt die gleiche Speichermenge, wie die Ausgangsdaten.

Die Unterteilung in t^3 Blöcke erlaubt die schnelle, lokal begrenzte Rücktransformation einzelner Blöcke. Es entstehen aber Diskontinuitäten durch die willkürliche Einführung von Grenzflächen an den Subblöcken. An den Grenzflächen müssten eigentlich die Nachbarn innerhalb der anliegenden Subblöcken berücksichtigt werden. Dies würde aber die strikte Trennung der Subblöcke aufheben und zu zusätzlichen Abhängigkeiten beim Rücktransformieren führen. Die Grenzflächen werden daher durch ein symmetrische Erweiterung speziell behandelt. Hierdurch können zwar Artefakte in die Bildgebung eingeführt werden, diese sind aber erst bei starker Kompression der Detail-Koeffizienten zu beobachten und im Fall der hier durchgängig verwendeten verlustfreien Kompression nicht vorhanden [31].

Für alle ungeraden Punkte ist nur für die obere Grenze des Subblocks kein benötigter Punkt vorhanden. Hierzu wird für den fehlenden Punkt $f_{j,t} = f_{j,t-2}$ genommen. Der verwendete Koeffizient braucht nicht gespeichert zu werden, da er



(a) Transformation *(b) Rücktransformation*

Abbildung 4.5: *Symmetrische Erweiterung der Grenzflächen bei der Wavelettransformation und -rücktransformation bei einer Subblockgröße von* $t = 16$ *Voxel.*

durch die symmetrische Waveletbasis implizit aus den gespeicherten Koeffizienten bei der Rücktransformation gewonnen wird. Bei der Transformation der geraden Punkte fehlt für die untere Grenze der benötigte Koeffizient. Hierfür wird der Punkt $d_{j-1,-1} = d_{j-1,0}$ herangezogen (vgl. Abb. 4.5).

Verwaltung der Multiskalendaten

Die Multiskalenhierarchie wird auf dem Host-Rechner in einem Baum mit jeweils acht Kindern (Octree) verwaltet. Der Wurzelknoten besteht aus dem t^3 Koeffizienten großen, tiefpassgefilterten Datensatz. Die Kinder sind jeweils die $t^3 - \frac{t}{2}^3$ Detail-Koeffizienten, die erforderlich sind um die nächste Hierarchiestufe rückzutransformieren. Etwaig komprimiert abgespeicherte d-Koeffizienten müssen vor der Rücktransformation dekomprimiert werden. Dies ermöglicht die Verwendung von lokal verschiedenen Hierarchiestufen.

4.2 Implementierung

Die vorgestellte Wavelettransformation wurde aufgrund ihrer Charakteristik gewählt, die sich sehr gut für eine Implementierung in Hardware eignet. Der arithmetische Aufwand ist gering, der Träger ist kompakt, die Transformation lässt sich vollständig als Pipeline abbilden und die Speicherzugriffe für die Koeffizienten haben eine regelmäßige Struktur. Die grundlegende Struktur der Einheit zur 3D-Wavelet-Rücktransformation wird in Abbildung 4.6 aufgezeigt. Neben dem Wavelettransformationsmodul ist eine Anzahl an Puffern notwendig, welche die einzelnen Module voneinander unabhängig machen. Die übliche Reihenfolge der Transformation in x-, y- und z-Richtung wurde explizit umgekehrt. Da die Rücktransformation in der umgekehrten Reihenfolge wie die Transformation erfolgen muss, würden schon bei der ersten Rücktransformation Puffer in der Größe eines Blocks erforderlich sein. Durch die Umkehr ist der erste Schritt in x-Richtung und alle Koeffizienten liegen vollständig

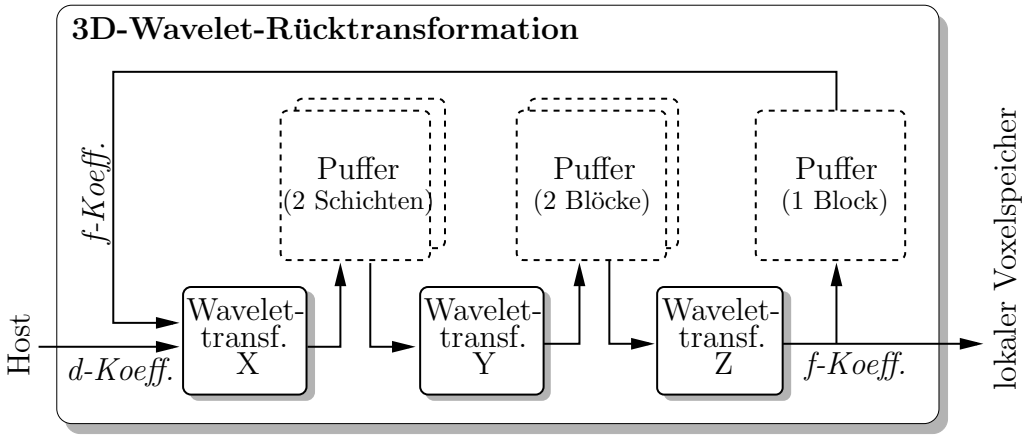


Abbildung 4.6: *Aufbau der Einheit zur Waveletrücktransformation. Die Wavelettransformationseinheiten unterscheiden sich nur in der Reihenfolge, in der die Eingangs- und Ausgangswerte adressiert werden. Um die Rücktransformation voll in einer Pipeline verarbeiten zu können, müssen die Puffer zwischen den Transformationseinheiten doppelt vorhanden sein, um die Abhängigkeiten zwischen den Einheiten aufzulösen.*

in einer Ebene. Somit reicht für die Ergebnisse der ersten Rücktransformation eine Schicht als Zwischenpuffer.

Für die erste Stufe werden abwechselnd d-Koeffizienten vom Host und f-Koeffizienten aus dem internen Speicher genommen. Das Ergebnis der Rücktransformation wird dann in dem ersten Puffer schichtweise abgelegt. Der Schichtpuffer ist redundant ausgelegt, da eine fertig gestellte Schicht als Eingabe für die nächste Stufe dient und erst nach der vollständigen Abarbeitung durch die zweite Stufe verändert werden darf.

Die zweite Transformationseinheit nimmt die Koeffizienten aus dem Schichtpuffer und transformiert sie in die nächste Raumrichtung. Da die nächste Stufe die dritte Raumrichtung behandelt und somit den schichtweisen Ablauf verlässt, muss die zweite Stufe die Ergebnisse in einen Zwischenspeicher ablegen, der Platz für einen kompletten Block bietet. Um wieder eine vollständige Pipelineverarbeitung zu gewährleisten ist dieser Blockspeicher wiederum doppelt ausgelegt.

Die letzte Waveleteinheit nimmt die Rücktransformation in die dritte Raumrichtung vor und speichert das Ergebnis in einen separaten Puffer. Dieser kann genau einen Block fassen und dient entweder als Eingabe für die nächste Wavelettransformation eines Blocks oder speichert den Block zwischen, bevor er in den lokalen Speicher der Bildgebungspipeline geschrieben wird. Da die Wavelettransformationen immer die gleichen Berechnungen ausführen, sind alle drei Waveleteinheiten identisch. Sie unterscheiden sich nur in der Art des Zugriffsmusters beim Holen der Eingangskoeffizienten und beim Ablegen der Ausgangskoeffizienten.

Adressierungsschema Wavelettransformation X

```
1: for z = 0; z < BlockgrößeZ; z += 1 do
2:   for y = 0; y < BlockgrößeY; y += 1 do
3:     for x = 0; x < BlockgrößeX; x += 1 do
4:       if (x mod 2 == 0) && (y mod 2 == 0) && (z mod 2 == 0) then
5:         Addr_Ein = f-Koeffizient-Eingang
6:       else
7:         Addr_Ein = d-Koeffizient-Eingang
8:       end if
9:       Addr_Aus = x + y · BlockgrößeX
10:    end for
11:  end for
12:  Ausgangspuffer tauschen
13: end for
```

Adressierungsschema Wavelettransformation Y

```
1: for z = 0; z < BlockgrößeZ; z += 1 do
2:   for x = 0; x < BlockgrößeX; x += 1 do
3:     for y = 0; y < BlockgrößeY; y += 1 do
4:       Addr_Ein = x + y · BlockgrößeX
5:       Addr_Aus = x + y · BlockgrößeX + z · BlockgrößeX · BlockgrößeY
6:     end for
7:   end for
8:   Eingangspuffer tauschen
9: end for
10: Ausgangspuffer tauschen
```

Adressierungsschema Wavelettransformation Z

```
1: for y = 0; y < BlockgrößeY; y += 1 do
2:   for x = 0; x < BlockgrößeX; x += 1 do
3:     for z = 0; z < BlockgrößeZ; z += 1 do
4:       Addr_Ein = x + y · BlockgrößeX + z · BlockgrößeX · BlockgrößeY
5:       Addr_Aus = x + y · BlockgrößeX + z · BlockgrößeX · BlockgrößeY
6:     end for
7:   end for
8: end for
9: Eingangspuffer tauschen
```

Abbildung 4.7 zeigt den Aufbau der Wavelettransformationseinheit. Die eigentlichen Berechnungen (vgl. Formel 4.9 bzw. 4.10) werden in zwei ALUs vorgenommen, die sich nur im Detail unterscheiden. Die ALUs werden von einer Kontrolleinheit mit den Koeffizienten versorgt. Dabei muss die Kontrolleinheit darauf achten, dass die ALU, welche die ungeraden Koeffizienten berechnet, immer erst auf zwei Ergebnisse aus der anderen ALU warten muss. Darüber hinaus sind Zähler implementiert, die dafür sorgen, dass für die Grenzkoeffizienten die entsprechenden replizierten Koeffizienten an den Eingängen der ALUs anliegen. Pro zu berechnenden Koeffizienten werden zwei neue Eingabewerte aus dem Puffer benötigt. Um diese zwei Anfragen

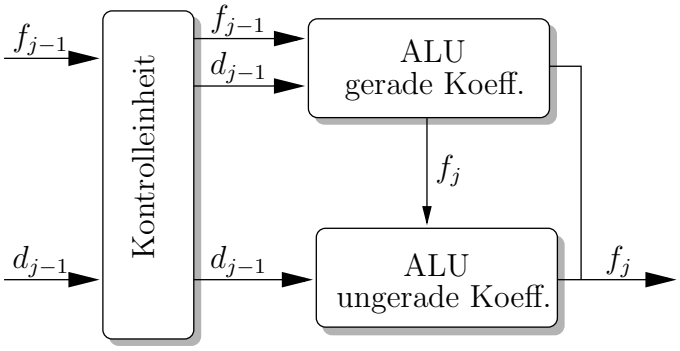


Abbildung 4.7: *Zusammenspiel der einzelnen ALUs zur Rücktransformation der geraden und ungeraden Waveletkoeffizienten. Verzögerungsglieder wurden übersichtlichkeitshalber ausgelassen.*

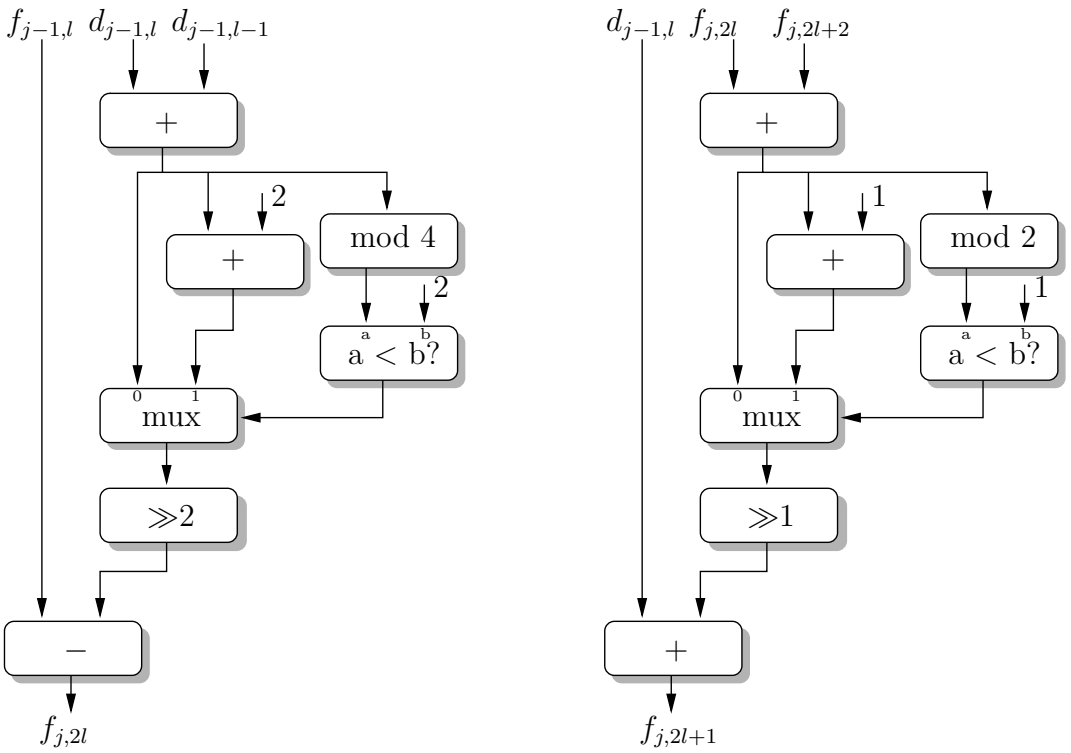
an den Speicher nicht nacheinander ausführen zu müssen, sollte der Speicher zwei Lesezugriffe unabhängig parallel ausführen können. Die als Zielplattform verwendeten Xilinx FPGAs verfügen über eine Anzahl „Dual-Port SelectRAMs“ [92], die diese Bedingung erfüllen. Der schematische Aufbau der ALUs ist in Abbildung 4.8(a) für die Bestimmung der geraden Koeffizienten $f_{j,2k}$ und in Abbildung 4.8(b) für die Bestimmung der ungeraden Koeffizienten $f_{j,2k+1}$ dargestellt.

Die Blocktransformationseinheit könnte auch mit einer einzigen Wavelettransformationseinheit implementiert werden, welche auch nur mit einem Puffer in der Größe eines Blocks auskäme. Die Rücktransformation erfolgt dann sequentiell in drei Durchgängen, was die Verarbeitungszeit aber auch mehr als verdreifachen würde.

4.3 Performanz und Ressourcenverbrauch

Die Transformationseinheit wurde in VHDL implementiert und für einen Virtex-II 4000 und Virtex-II 8000 synthetisiert. Die detaillierten Ergebnisse hinsichtlich der benötigten Ressourcen sind in Tabelle 4.1 zusammengefasst. Die verwendeten logischen Ressourcen sind mit 4 % relativ gering. Allerdings ist die Wavalettransformationseinheit mit 12 % Ausnutzung der Block RAMs recht hoch. Diese Zahl ergibt sich nicht nur durch die Größe der Puffer, sondern auch durch die ineffiziente Ausnutzung der einzelnen Block RAMs. Jeder Puffer muss, wegen der Anforderung zwei unabhängige Werte pro Zyklus lesen oder schreiben zu können, sein eigenes Block RAM besitzen, auch wenn er es nicht ausfüllt. So verwendet ein Schichtpuffer nur $16^2 \cdot$ 16 Bit = 4 kBit der 18 kBit, die im Block RAM verfügbar sind. Für die maximale Taktfrequenz der Einheit wurde nach der Synthese 165 MHz ermittelt.

Der maximale Decodierdurchsatz der Einheit beträgt dank der Pipelinestruktur ein Koeffizienten pro Taktzyklus. Die gesamte Einheit kann somit ca. 40 000 Blöcke pro Sekunde rücktransformieren. Dies entspricht einem Datendurchsatz von ca. 300 MB/s, bei einer Blockgröße von 16^3 Koeffizienten und 16 Bit pro Koeffizienten. Die schnellste Waveletdekompression in Software erreicht hingegen maximal 100 MB/s [30] Durchsatz.



(a) ALU gerade Koeff. ($f_{j,2l}$).

(b) ALU ungerade Koeff. ($f_{j,2l+1}$).

Abbildung 4.8: *Prozessablauf der ALUs der geraden und ungerade Waveletkoeffizienten (ohne Verzögerungsglieder).*

FPGA Nutzung für:	*Virtex-II 2V4000ff1152*			*Virtex-II 2V8000ff1517*		
Ressource	*Benutzt*	*Verfügbar*	*Nutzung*	*Benutzt*	*Verfügbar*	*Nutzung*
IOs	0	0	0,00 %	0	0	0,00 %
Global Buffers	1	16	6,25 %	1	16	6,25 %
Function Generators	1 785	46 080	3,87 %	1 794	93 184	1,93 %
CLB Slices	894	23 040	3,88 %	897	46 592	1,93 %
Dffs or Latches	1 334	48 552	2,75 %	1 341	96 508	1,39 %
Block RAMs	14	120	11,67 %	14	168	8,33 %
Block Multipliers	0	120	0,00 %	0	168	0,00 %
Block Multiplier Dffs	0	4 320	0,00 %	0	6 048	0,00 %

Tabelle 4.1: *Syntheseergebnisse der Waveletrücktransformation für zwei FPGA-Typen. Gegenübergestellt ist der momentan verwendete Virtex-II 4000 und der aktuelle Virtex-II 8000.*

Kapitel 5

Visualisierungspipeline für Multiskalendaten

Mit der kontinuierlich fortschreitenden Verbesserung medizinischer 3D-Scanner, sowie der steigenden Leistungsfähigkeit wissenschaftlicher Simulationsmodelle besteht der Bedarf zu der Visualisierung immer größerer Volumendatensätze. Um die damit einhergehende Datenmenge handhaben zu können, sind Techniken erforderlich, die über bisher bekannte Verfahren der hardwarebeschleunigten Volumenvisualisierung hinausgehen. In diesem Kapitel werden die notwendigen Änderungen erläutert, die an der Visualisierungspipeline des VIZARD II vorgenommen werden müssen, um Datensätze, die um eine Größenordnung umfangreicher als bisher sind darstellen zu können. Hierfür wird eine generische Multiskalenunterstützung integriert, die nicht ausschließlich auf die bisher thematisierte Wavelettransformation beschränkt ist.

In Abbildung 5.1 sind alle relevanten Komponenten der VIZARD II Pipeline sowie deren Änderungen und Ergänzungen aufgeführt. In den folgenden Abschnitten wird auf die Änderungen im Einzelnen eingegangen.

5.1 Raycaster

Die Vorzüge der Multiskalenvisualisierung bestehen nicht nur in der effizienten Komprimierbarkeit der Volumendaten, sondern auch in Bezug auf die Darstellungsgeschwindigkeit. Da in Bereichen mit niedrigem Informationsgehalt höhere Hierarchiestufen des Volumens gewählt werden können, müssen diese Bereiche auch nur mit einer deutlich niedrigeren Rate abgetastet werden. Die Gesamtzahl der Abtastpunkte, die für die Berechnung eines Bildes notwendig sind, kann somit deutlich reduziert werden.

Die Abtastpunkte entlang eines Sehstrahls werden vom *Raycaster* bestimmt. Dieser nimmt entweder den initialen Eintrittspunkt des Strahls in das Volumen oder den jeweils vorherigen Abtastpunkt plus ein Strahlinkrement i. Anhand der Länge des Strahlinkrements kann die Abtastrate gesteuert werden. Der Raycaster bekommt für jedes Pixel im Bild einen Strahleintrittspunkt sowie das zugehörige Inkrement übergeben. Dieses Inkrement entspricht der vom Benutzer gewünschten Abtastrate

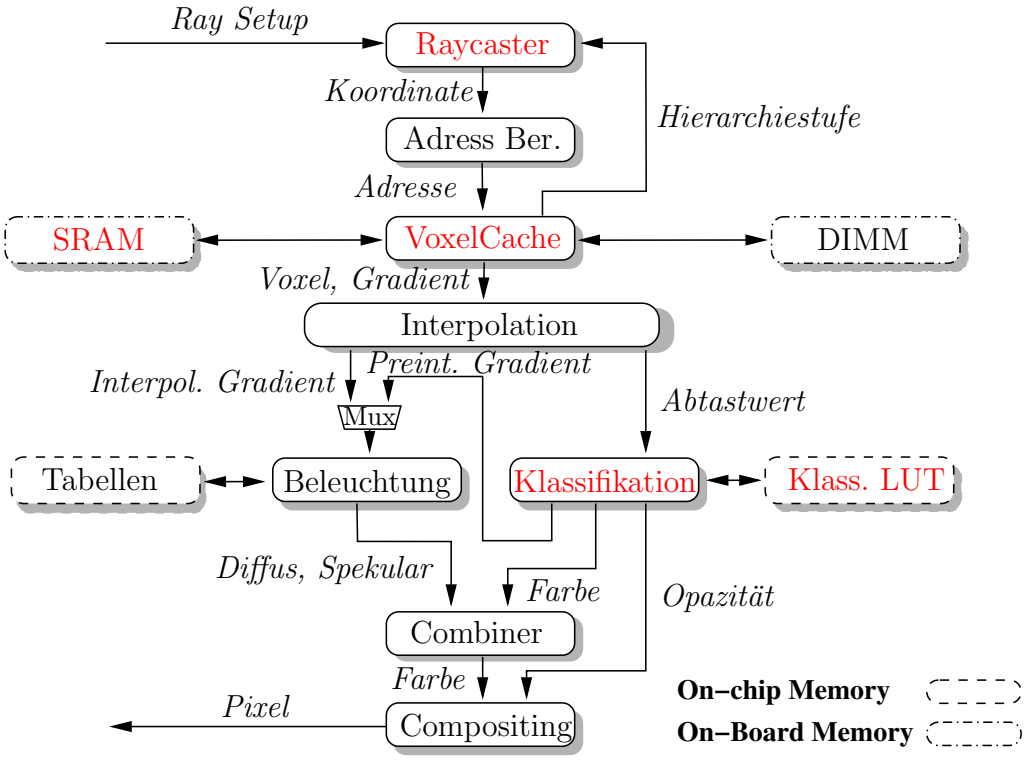


Abbildung 5.1: *Visualisierungspipeline des VIZARD II mit Erweiterung zur Unterstützung des Multiskalenmodells. In Rot sind die neuen sowie die modifizierten Komponenten hervorgehoben.*

bezogen auf das Weltkoordinatensystem und der höchsten Auflösungsstufe. Wechselt der Strahl in eine Region in der nur eine niedrigere Auflösungsstufe zur Verfügung steht, würde ein Beibehalten des Inkrements einer effektiv höheren Abtastrate entsprechen. Um diese unnötige Erhöhung zu vermeiden und die Berechnung des Bildes zu beschleunigen, wird dem Raycaster für jeden Strahl die aktuelle Hierarchiestufe h_n mitgeteilt, für die er den nächsten Abtastpunkt p_{j+1} berechnen soll:

$$\vec{p}_{j+1} = \vec{p}_j + 2^{h_n} \cdot \vec{i}\,.$$

Die Multiplikation mit der Zweierpotenz der Hierarchiestufe kann in Hardware elegant als Linksshift implementiert werden. Weitere Änderungen sind am Raycaster nicht erforderlich.

Die Hierarchiestufe h_n, die aktuell am Raycaster anliegt, bezieht sich auf einen Punkt der gerade aus dem VoxelCache bezogen wurde und spiegelt somit nicht die Hierarchiestufe des neuen, sondern die eines vorhergehenden Punktes wieder. Dies führt bei einem Wechsel von einer höheren zu einer niedrigeren Auflösungsstufe zu einer Überabtastung, beim umgekehrten Wechsel zu einer Unterabtastung. Die Anzahl der Punkte mit Über- oder Unterabtastung ergibt sich aus der Latenz der Bildgebungspipeline zwischen Raycaster und VoxelCache. Die Ermittlung der Hierarchiestufe zum Zeitpunkt der Berechnung der Abtastpunkte würde zu einem

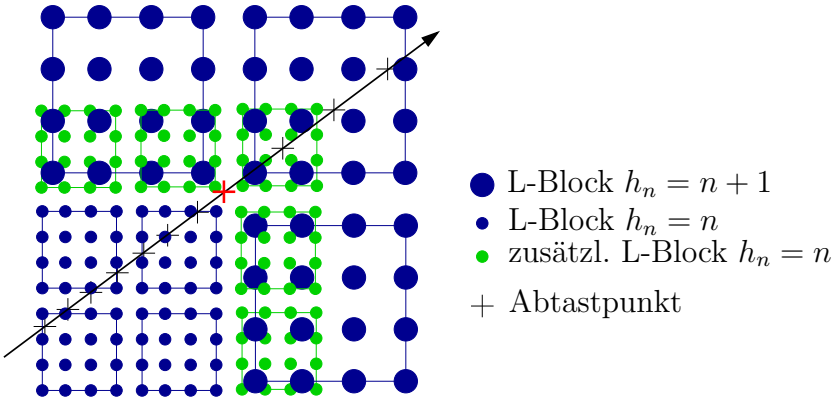


Abbildung 5.2: *Sehstrahl durch verschiedene Hierarchiestufen (h_n) mit adaptiven Abtastabständen. Die zusätzlichen Blöcke (grün) werden nicht direkt adressiert, sondern nur als Ergänzung für eine Achterumgebung geladen, um die Interpolation in derselben Hierarchiestufe zu ermöglichen. Sind diese Blöcke nicht vorhanden muss zwischen verschiedenen Hierarchiestufen interpoliert werden (rot markierter Abtastpunkt). Für alle anderen Abtastpunkte liegen die Daten zur Interpolation in derselben Hierarchiestufe.*

Aufheben der Pipelinestruktur führen und einen signifikanten Leistungseinbruch zur Folge haben.

Durch die Verwendung von Strahlgruppen, bei denen die Strahlpunkte in abwechselnder Reihenfolge für einen Strahl berechnet werden, verteilt sich die störende Latenz auf mehrere Strahlen. Bei einer Größe der Strahlgruppen von $4 \cdot 4$ Strahlen, verteilt sich die Latenz von 20 Zyklen zwischen Raycaster und VoxelCache auf 16 Strahlen. Dies bedeutet, für maximal zwei Strahlpunkte pro Strahl wird bei einem Wechsel der Hierarchiestufe die Abtastung nicht entsprechend des vorgegebenen Wertes vorgenommen. Für die temporäre Überabtastung führt dies zu keiner Verschlechterung des Bildes; bei einer Unterabtastung kann dies aber theoretisch zu Bildartefakten führen. Soll diese minimale lokale Unterabtastung verhindert werden, dürfen nur Wechsel von niedriger in höher aufgelöste Hierarchiestufen erfolgen. Dies kann leicht bei der Traversierung der Octree-Hierarchie sichergestellt werden, führt aber abhängig vom Datensatz zu Einbußen bei der Performanz, da nun mehr Blöcke in höherer Auflösung nötig sind.

Da die Abtastrate nicht nur für die Bestimmung der Abtastpunkte, sondern auch für die Integration des Volume Rendering Integrals von Bedeutung ist, muss die jeweilig verwendete Hierarchiestufe in einem Kontrollvektor für jeden Strahlpunkt mitgeführt werden.

5.2 Interpolation

Da die Wavelettransformation gleichmäßig in alle Raumrichtungen wirkt, ändert sich für die trilineare Interpolation nichts, solange die, für die Interpolation verwendete, Nachbarschaft vollständig in derselben Hierarchiestufe liegt (vgl. Abb. 5.2).

Dies muss bei der Traversierung der Hierarchie auf dem Host-Rechner sichergestellt werden und kann über eine geeignete Traversierung der Octree-Hierarchie erfolgen. Da der VoxelCache zu jedem Abtastpunkt die optimale Hierarchiestufe bestimmen kann, werden alle notwendigen Voxel der Achterumgebung konsistent mit derselben Hierarchiestufe adressiert, auch wenn sie in verschiedenen L-Blöcken liegen. Sollte sich dennoch ein L-Block nicht in der notwendigen Hierarchiestufe im lokalen Speicher befinden, können zwei unterschiedliche Strategien gewählt werden um den Konflikt aufzulösen.

Die korrekte Variante erfordert ein explizites Holen des L-Blocks vom Host-Rechner und zieht eine dementsprechend hohe Totzeit der Pipeline nach sich. Soll diese Verzögerung vermieden werden, kann an Stelle des angeforderten L-Blocks sein nächst niedrigerer aufgelöster Repräsentant aus dem Speicher in den Cache geladen werden. Diese Strategie ist immer erfolgreich, da der Block des Wurzelknotens sich immer im Speicher befindet und auch alle Voxel (wenn auch tiefpassgefiltert) repräsentiert. Die mit diesem L-Block vollzogene Interpolation ist nicht vollständig korrekt, verursacht aber keine zusätzliche Totzeit der Pipeline.

5.3 VoxelCache

Die Einholung einer Achterumgebung durch den VoxelCache muss wegen der zusätzlichen Hierarchiestufen modifiziert werden. Da jeder Punkt im Volumen einen Repräsentanten in jeder Hierarchiestufe besitzt, ist die Spezifikation des Abtastpunkts nur durch seine Position in Weltkoordinaten nicht eindeutig. Die Bildgebungspipeline hat zwar im Kontrollvektor die Hierarchiestufe, die der Raycaster spekulativ für diesen Punkt vorhergesagt hat, es ist aber nicht sichergestellt, dass der zum Punkt gehörende Block im Speicher verfügbar ist.

Für die weitere Berechnung des Bildes ist die Hierarchiestufe aus der der Abtastpunkt generiert wurde auch unerheblich. Dennoch sollte, um eine optimale Bildqualität zu erreichen, immer die maximal verfügbare Auflösungsstufe genutzt werden um den Abtastpunkt zu repräsentieren. Da sich die Blöcke der Hierarchiestufen dynamisch ändern können, muss ein Mechanismus installiert werden anhand dessen der Cache bestimmen kann, welcher momentan im Speicher befindliche L-Block die optimale Repräsentation darstellt.

Dies wird über eine eindimensionale Hierarchie-Tabelle realisiert, die über den Abtastpunkt indiziert wird. Da auf der untersten Hierarchiestufe die Blöcke eine Kantenlänge von 16 Voxel besitzen, sind die untersten 4 Bit der Komponenten des Abtastpunkts nicht relevant. Die Tabelle reduziert sich somit auf eine Größe von $\frac{D_x}{16} \cdot \frac{D_y}{16} \cdot \frac{D_z}{16}$, wobei D_x, D_y und D_z die Dimensionen des Datensatzes in die entsprechende Raumrichtung sind.

Um weiterhin einen Datendurchsatz von einer Achterumgebung pro Taktzyklus zu erreichen, muss die Hierarchie-Tabelle SRAM-basiert eingerichtet sein. Eine Änderung in der Zusammensetzung der Blöcke im externen Speicher muss sich im SRAM widerspiegeln. Da sich aber Änderungen der Hierarchiestufe meist nur in lokal begrenzten Bereichen ereignen, sind die damit verbundenen Speicherbandbreitean-

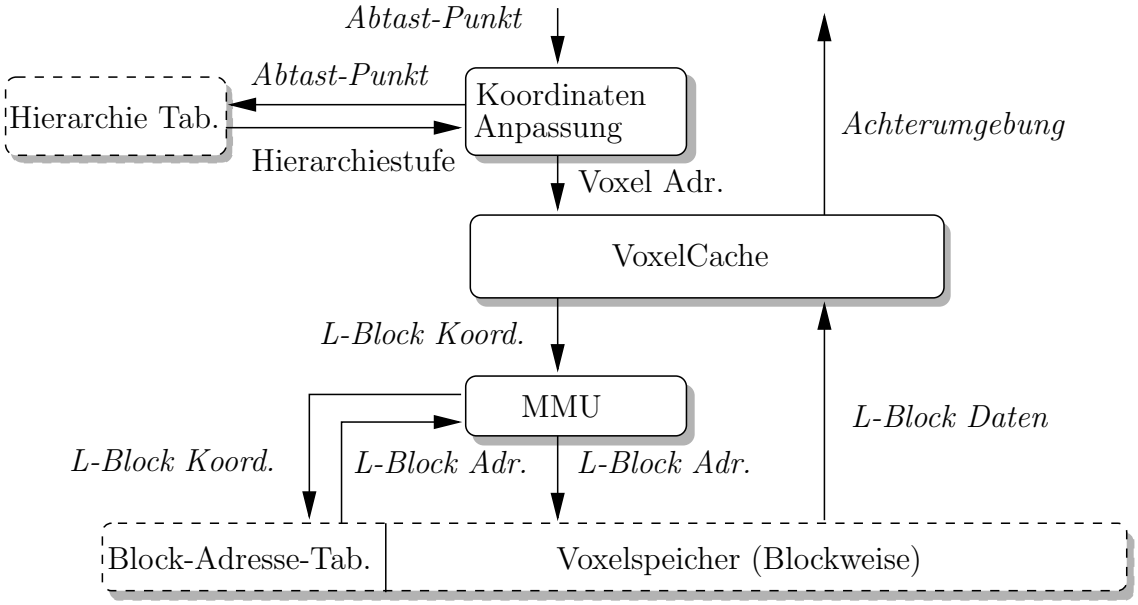


Abbildung 5.3: *Erweiterung des VoxelCache zur Unterstützung der Multiskalenhierarchie.*

forderungen gering. Im vorliegenden Fall wird ein „Xilinx Virtex-II™ Development Kit“[10] der Firma Avnet verwendet. Dies verfügt zusätzlich zu einem SDRAM Interface über eine gesonderte AvBus-Verbindung, über die eine Speichererweiterungskarte[9] mit 1 MB angeschlossen werden kann. Bei einer Datenbusbreite von 32 Bit und einer Busfrequenz von 100 MHz ergibt sich so eine Bandbreite von 400 MB/s. Selbst der komplette Austausch der Hierarchie-Tabelle von 1 MB stellt somit keine Problem dar.

Nach dem Ermitteln der optimalen Hierarchiestufe wird aus ihr und der Koordinate des Abtastpunkts eine neue virtuelle Adresse für das Finden der Achterumgebung im VoxelCache generiert. Der virtuelle Adressraum ist dabei so angeordnet, dass an das Volumen mit der höchsten Auflösungsstufe, welches der Hierarchiestufe 0 entspricht, das Volumen der nächsten Hierarchiestufe in x-Richtung angefügt wird. Alle weiteren Hierarchiestufen werden in gleicher Weise abwechselnd in x- und y-Richtung angefügt (vgl. Abb. 5.4). Durch diese Anordnung wird der virtuelle Adressraum in eine Raumrichtung um den Faktor zwei größer als das originale Volumen. Die virtuellen Adressen im VoxelCache werden daher mit einem Bit mehr als die Abtastpunkte ausgestattet. Den Versatz $(T_{x,y}(h_n))$ für die jeweilige Hierarchiestufe innerhalb des Adressraums wird in einer Tabelle abgespeichert, die für jeden Datensatz beim Laden der statischen Datensatzinformationen initialisiert wird. Neben dem Versatz der höheren Hierarchiestufen muss auch eine Skalierung der Abtastpunkte vorgenommen werden, da sich die Anzahl der verfügbaren Stützstellen in jede Raumrichtung pro Hierarchiestufe h_n halbiert.

Ist eine errechnete Umgebung nicht im VoxelCache vorhanden, muss die Memory Management Unit (MMU) den Ort dieses Blockes im on-board Speicher kennen. Da sich die Zusammensetzung der Blocks im Speicher dynamisch ändert, muss in

$$
\begin{aligned}
x_v &= \frac{x}{2^{h_n}} + T_x(h_n) \\
y_v &= \frac{y}{2^{h_n}} + T_y(h_n) \\
z_v &= \frac{z}{2^{h_n}}
\end{aligned}
$$

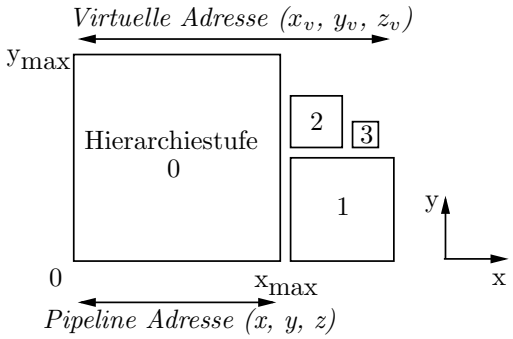


Abbildung 5.4: *Generierung der virtuellen Adresse aus der Pipelineadresse (x,y,z) und dem Versatz $T_{x,y}(h_n)$ der jeweiligen Hierarchiestufe h_n, sowie Aufbau des virtuellen Adressraums.*

einer separaten Block-Adress-Tabelle die Startadresse eines Blocks im Speicher abgelegt werden. Da in dieser Tabelle für jeden Block jeder Hierarchiestufe ein Eintrag vorhanden sein muss, auch wenn sich dieser Block momentan nicht im Speicher befindet, ist diese Tabelle zu groß um sie on-chip zu speichern und muss somit in den on-board Speicher gelegt werden. Der notwendige Speicherplatz S_T berechnet sich aus der Anzahl an Blocks b in der gesamten Hierarchie und somit aus der Dimension des Datensatzes $D_{x,y,z}$ und der Kantenlänge eines Blocks t:

$$
\begin{aligned}
S_T &= (\log_2 b) \cdot b \,\text{Bit} \quad , mit \\
b &= \frac{D_x \cdot D_y \cdot D_z}{t^3} \cdot \sum_{i=0}^{h_{max}-1} \frac{1}{2^{3 \cdot i}} \; .
\end{aligned} \tag{5.1}
$$

Wobei sich die maximale Hierarchiestufe des Datensatzes h_{max} aus

$$
h_{max} = \left\lceil \log_2 \frac{max(D_x, D_y, D_z)}{t} \right\rceil
$$

berechnet.

Da im Allgemeinen der Speicherplatz für die Blöcke begrenzt ist und sich nicht alle Blöcke im Speicher befinden können, ist die Anzahl der im Speicher zu referenzierenden Blöcke b_S begrenzt. Somit ändert sich die Formel (5.1) zu:

$$
S_T = (\log_2 b_S) \cdot b \,\text{Bit} \; . \tag{5.2}
$$

Insgesamt muss die Größe des lokalen on-board Speichers S_{ges} sowohl für die Tabelle und die Blöcke Platz bieten und ergibt sich zu:

$$
S_{ges} = S_T + 2 \cdot b_S \cdot t^3 \; . \tag{5.3}
$$

5.4 Klassifikation

Die Integration des DVRI (vgl. Formel 2.9) sollte zum selben Ergebnis führen, unabhängig davon, wie viele und in welchem Abstand Abtastpunkte gewählt werden.

Dichtewert	α_0	$\cdots$	$\alpha_{h_{max}}$	Rot	Grün	Blau	Ambiente	Diffus	Spekular
1									
$\vdots$									
$2^{\text{voxelbits}}$									

Tabelle 5.1: *Erweiterte Klassifikationstabelle. Zusätzlich zu den Materialparametern und dem Alpha-Wert α_0 für die höchste Auflösungsstufe sind noch alle korrigierten Alpha-Werte für die weiteren Hierarchiestufen abgelegt.*

Um dies zu gewährleisten muss die Opazität, die sich stets auf die vom Benutzer vorgegebene Abtastrate der höchsten Auflösungsstufe bezieht, bei sich ändernden Abständen der Punkte entsprechend angepasst werden. Da das Compositing eine nichtlineare Operation ist, muss der Opazitätswert gemäß folgender Formel [52] angepasst werden:

$$\alpha = 1 - \sqrt[N]{1 - \alpha_0}\,.$$

Der Opazitätswert α_0 ist dabei der vorgegebene Wert und N der Faktor der Überabtastung.

Da sich die Abstände nur in einem Faktor von 2^{h_n} unterscheiden können, werden die korrigierten Opazitätswerte nicht in der Pipeline berechnet, sondern in einem Vorverarbeitungsschritt in einer Tabelle abgespeichert (vgl. Tab. 5.1). Der zusätzliche Speicheraufwand gegenüber einer Speicherung eines einzelnen Alpha-Werts ist linear von der Anzahl an Hierarchiestufen abhängig. Die Materialparameter (Farb- und Beleuchtungswerte) und der Opazitätswert für einen Abtastpunkt wird weiter über den Voxelwert adressiert. Anhand der Hierarchiestufe h_n für den aktuellen Abtastpunkt wird dann die korrekte Spalte des Opazitätswerts gewählt.

Da der Aufbau der Klassifikationstabelle nicht dynamisch zur Laufzeit geändert werden kann, wird für die maximale Hierarchiestufe $\mathrm{h_{max}}$ eine künstliche Grenze von acht Stufen gesetzt. Dies erlaubt, bei einer Grundgröße der Blocks von 16 Voxel, Datensätze mit einer Auflösung von $16 \cdot 2^8$ Voxel $= 4\,096$ Voxel in jeder Dimension.

5.5 Bildgebung

Die verwendeten Ausgangsdaten bestehen aus natürlichen Zahlen eines begrenzten Wertebereichs $[0, n]$. Durch die verwendete Wavelettransformation können die zur Bildgebung herangezogenen Durchschnitts-Koeffizienten aber den gültigen Wertebereich verlassen. Im Extremfall kann das Intervall nach einer Transformation auf $[-\frac{n}{2}, \frac{3n}{2}]$ anwachsen. Um die Werte für die Bildgebungspipeline wieder auf den ursprünglichen Wertebereich zu bringen, werden die f-Koeffizienten (x) auf das Intervall $[0, n]$ abgebildet:

$$x = \begin{cases} 0 & : \quad x < 0 \\ x & : \quad 0 \leq x \leq n \\ n & : \quad n < x \end{cases}\,.$$

Um bei der Rücktransformation keinen Verlust an Genauigkeit zu erhalten, werden die f-Koeffizienten mit voller Genauigkeit gespeichert und erst bei der Übertragung an die Bildgebungspipeline abgeschnitten.

5.6 Ergebnisse

Um die Möglichkeiten der Multiskalenpipeline evaluieren zu können, müssen mehrere Rahmengrößen vorgegeben werden. Diese wurden der Zielplattform, dem FPGA-Entwicklungsboard der Firma Avnet[10], entnommen. Die im Zusammenhang dieses Kapitels relevanten Daten sind hierbei der verwendete FPGA (Virtex-II 2V4000), der on-board DDRAM-Speicher (512 MB) sowie der verfügbare on-board SRAM-Speicher (1 MB). Die Größe des, für die Hierarchie-Tabelle verfügbaren, SRAM-Speichers limitiert die Datensatzgröße auf $2\,254^3$ Voxel; dies entspricht aber immer noch zwei Instanzen des „Visible Human“-Datensatzes. Aus dieser Datensatzgröße berechnet sich eine maximale Hierarchiestufe $h_{max} = 8$ und somit ein Platzbedarf der Tabelle für Block-Adressen von $S_T = 5,33$ MB. Der DDRAM-Speicher von 512 MB erlaubt somit noch das gleichzeitige Speichern von $b_S = 64\,000$ Blöcken.

Als konkrete Umsetzung der Multiskalenhierarchie wird die Waveletttransformation zugrunde gelegt. Um für die Waveletdekompression Kenngrößen zu ermitteln, wurde mit diesen Rahmendaten eine C++-Simulation der Multiskalenpipeline durchgeführt. Hierzu wurden für zwei Datensätze ein Kamerapfad abgelaufen und für jedes Bild, die auf die Karte transferierten Blöcke von d-Koeffizienten, sowie die Gesamtzahl der Blöcke im Speicher ermittelt. Die Kamera beginnt mit einem Bild, das den vollständigen Datensatz zeigt, zoomt dann auf einen ausgewählten Ausschnitt des Datensatzes und dreht sich im Folgenden einmal in einer 360° Rotation um den Datensatz. Durch diese Kamerafahrt werden alle Hierarchiestufen durchlaufen.

Um die L-Blöcke zu bestimmen, die für die Bildgebung herangezogen werden, wird der Octree der Wavelethierarchie vom Wurzelknoten zu den Blättern traversiert. Für jeden Knoten innerhalb der Traversierung wird die projizierte Fläche dieses L-Blocks auf der Bildebene bestimmt. Unterschreitet diese Fläche einen Schwellwert, wird der L-Block nicht weiter aufgelöst und die Traversierung an dieser Stelle abgebrochen. Um eine maximale Bildqualität zu erreichen und den ungünstigsten Fall zu ermitteln, wurde als Abbruchkriterium der Hierarchietraversierung ein Pixel als projizierte Fläche eingestellt.

Das Diagramm in Abbildung 5.5 zeigt die ermittelten Größen für den „Visible Human“-Datensatz. Abbildung 5.6 zeigt vier Positionen entlang des hierfür verwendet Kamerapfads. Deutlich ist zu erkennen, dass zu Beginn nur wenige Blöcke ausreichen, um das Abbruchkriterium zu erfüllen. Erst wenn deutlich näher an den Datensatz gezoomt wird, werden sprungartig Blöcke nachgeladen. Maximal sind dabei 18 000 Blöcke zu transferieren, was ca. 140 MB an Daten entspricht. Dieser Effekt schwächt sich ab, sobald ein Großteil des Datensatzes komplett außerhalb der Sichtpyramide liegt. Ab diesem Zeitpunkt müssen nur noch wenige Blöcke ausgetauscht werden. Ein ähnliches Bild ergibt sich für den Weihnachtsbaum-Datensatz [38] (vgl. Abb. 5.7 und 5.8). Aufgrund der Größe wird aber die maximale Zahl an Blöcken nur

kurzfristig benötigt und auch die Anzahl an transferierten Blöcken liegt mit maximal 10 000 Blöcken (78 MB) deutlich niedriger. In diesem Zusammenhang ist darauf hinzuweisen, dass nicht alle transferierten Blöcke im on-board Speicher abgelegt werden. Einige Blöcke werden nur als Zwischenergebnis für die Rücktransformation weiterer Blöcke benötigt, die in der Hierarchie tiefer liegen. Bei Spitzenlasten kann somit die im vorhergehenden Kapitel präsentierte Wavelettransformationseinheit immer noch genug Blöcke für 2–4 Bilder pro Sekunden verarbeiten. Sollen auch diese Spitzenlasten mit höherer Bildrate dargestellt werden, so kann die Wavelettransformationseinheit, bei vorhandenen Ressourcen, auch parallel instantiiert werden, da keine Abhängigkeit zwischen den einzelnen Blöcken besteht.

Anstatt der projizierten Fläche eines Blockes können noch andere Kriterien zur Auswahl der richtigen Hierarchiestufe herangezogen werden. So kann man auch zusätzlich alle Blöcke nach dem Block durchsuchen, der den größten Fehler im Bildraum verursacht. Dieser Block wird dann durch seine höher aufgelösten Blöcke ausgetauscht. Die Berechnung des Bildfehlers erfolgt hierbei großteils als Vorverarbeitungsschritt [30]. Die an dieser Stelle präsentierten Ergebnisse und Statistiken stellen somit eine untere Grenze dar, die durch die Wahl der Hierarchiestufen und des Abbruchkriteriums der Hierarchietraversierung weiter verbessert werden können. Auf die Implementierung der Multiskalenpipeline und die vorgestellten Komponenten hat dies aber keinerlei Einfluss.

Anhand der vorgestellten Architektur konnte somit gezeigt werden, dass sich das waveletbasierte Multiskalenmodell in eine dedizierte Visualisierungspipeline für Volumendaten integrieren lässt. Durch die Nutzung des VoxelCaches lässt sich die notwendige Virtualisierung der Adressberechnung durchführen, ohne größere Modifikationen an der Visualisierungspipeline vornehmen zu müssen.

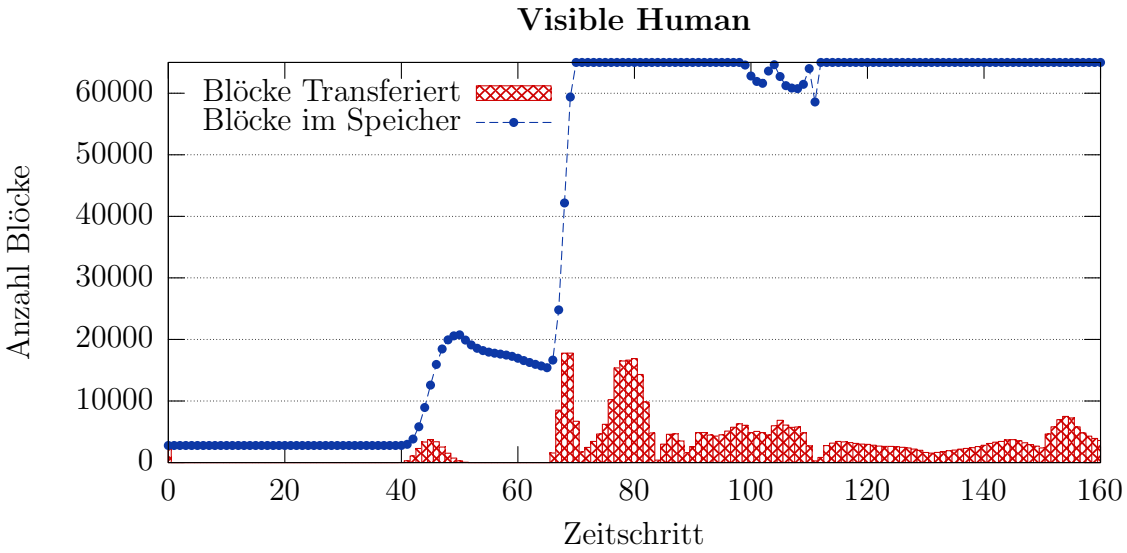


Abbildung 5.5: *Statistik über transferierte und gespeicherte Blöcke für einen Kameraschwenk am Beispiel des „Visible Human“-Datensatzes. Vier Bilder des Kamerapfades sind in Abb. 5.6 abgebildet. Der Datensatz besitzt eine Auflösung von 2 048 · 1 216 · 1 874 Voxel.*

Abbildung 5.6: *Vier Zeitschritte des Kameraschwenk aus Abb. 5.5.(a) – (c) Zoom sowie (d) anschließende Rotation um den Kopf. Als Bildeinlage ist in jedem Bild eine Visualisierung der verwendeten Hierarchie eingebettet. Es ist deutlich zu erkennen, wie im Verlauf des Kamerapfads der Bereich des Kopfes in deutlich steigender Auflösung abgebildet wird und nicht sichtbare Bereiche aus dem Speicher verworfen werden.*

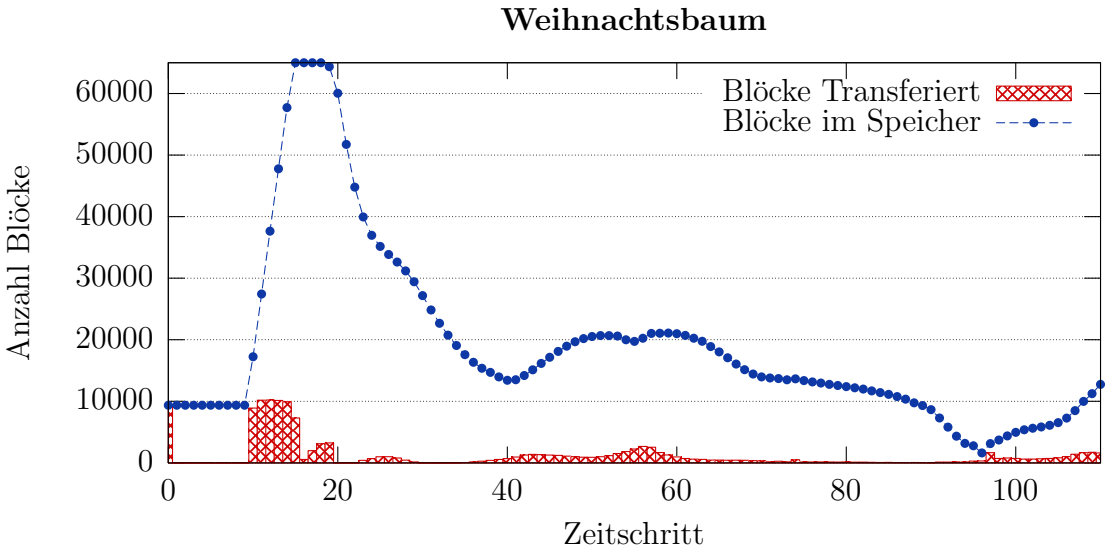
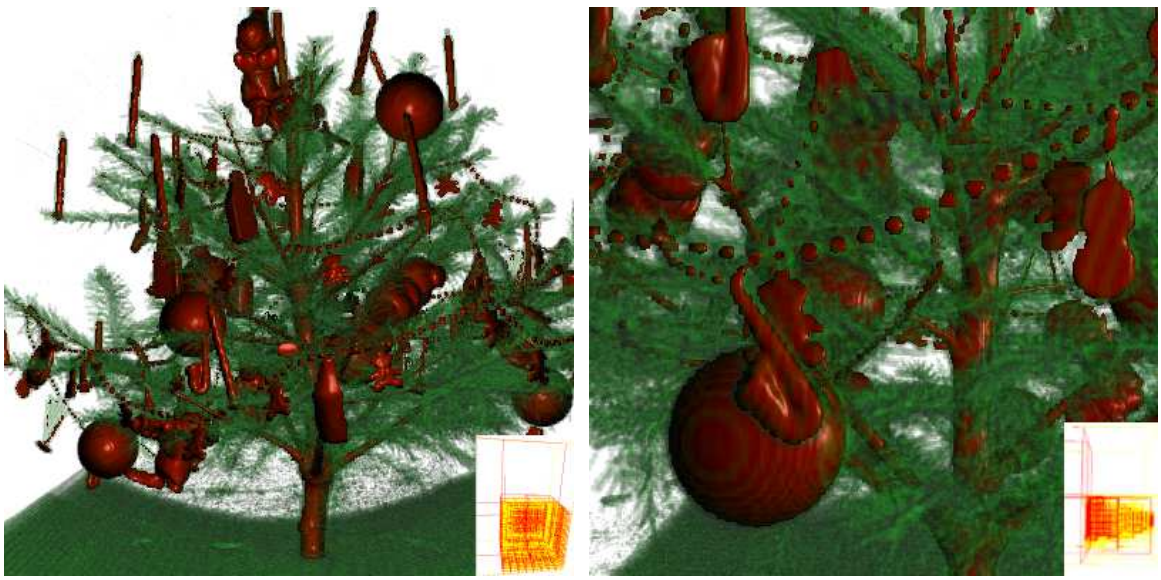


Abbildung 5.7: *Statistik über transferierte und gespeicherte Blöcke für einen Kameraschwenk am Beispiel des Weihnachtsbaum-Datensatzes. Zwei Bilder des Kamerapfades sind in Abb. 5.8 abgebildet. Der Datensatz besitzt eine Auflösung von* $512 \cdot 512 \cdot 999$ *Voxel.*



(a) *(b)*

Abbildung 5.8: *Erster und letzter Zeitschritt aus dem Kameraschwenk der in Abb. 5.7 evaluiert wurde. Als Einlage ist in jedem Bild eine Visualisierung der jeweilig verwendeten Hierarchie eingebettet.*

Kapitel 6

Verfahren zur Volumenkompression in Hardware

Das in den vorhergehenden Kapiteln vorgestellte Multiskalenmodell ermöglicht durch die richtige Wahl der Auflösungsstufe, die zur Darstellung eines Bildes benötigten Volumendaten signifikant zu reduzieren. Ein weiterer Vorteil der Wavelettransformation blieb dennoch bisher ungenutzt. Für lokal korrelierte Funktionen liefert die Wavelettransformation überwiegend Detail-Koeffizienten die nahe oder gleich Null sind. Diese Eigenschaft kann in zweifacher Weise genutzt werden. Die hohe Anzahl an Null-Koeffizienten eignet sich durch die inhärent geringe Entropie gut zur Kompression. Zusätzlich kann durch eine Quantisierung der Detail-Koeffizienten eine noch höhere Kompressionsrate erreicht werden. Durch die Quantisierung wird die Kompression allerdings verlustbehaftet. Im Folgenden werden nur Daten betrachtet, die ohne Quantifizierung komprimiert wurden, da eine zusätzliche Quantisierung beim Codieren trivial ist und kein Änderung in der Implementierung des Decoders erfordert. Des Weiteren stellen die ermittelten Werte für die verlustfreie Kompression den ungünstigsten Grenzfall dar, womit die Leistungsmerkmale mit Quantifizierung nach oben abgeschätzt sind.

Zur Kompression der Waveletkoeffizienten wird in diesem Kapitel eine Implementierung eines Fixed-Huffmandecodierers vorgestellt. Dieses Verfahren eignet sich in mehrfacher Weise zur Umsetzung in Hardware. Es erreicht für Waveletkoeffizienten sehr gute Kompressionsraten, benötigt minimale logische Ressourcen und erfordert keinen Speicherplatz für Codebücher. Die Codierung kann als einmaliger Vorverarbeitungsschritt in Software erfolgen, und ermöglicht, dass nur die komprimierten Waveletkoeffizienten gespeichert werden müssen.

Auch wenn die Multiskalendarstellung für sehr große Volumendatensätze hervorragend geeignet ist, so besitzt diese Methode doch einen schwerwiegenden Nachteil, der die Verwendung für in Echtzeit generierte animierte Volumendaten (4D-Daten) ungeeignet macht. Die Waveletkompression ist sehr rechen- und speicherintensiv und erfordert entweder eine zeitintensive Vorverarbeitung oder dedizierte Hardware. In vielen Fällen ist aber eine aufwendige Vorverarbeitung und die daraus resultierende Latenz nicht akzeptabel sowie eine Integration von dedizierter Hardware in das

den Datensatz erzeugende System nicht immer möglich. Daher wird im zweiten Teil dieses Kapitels ein 3D-Huffmandecoder vorgestellt der speziell für diese Szenario entwickelt wurde.

6.1 Huffmancodierung

Eine einfache Methode zur universellen und verlustfreien Quellcodierung stellt die Huffmancodierung [34, 78] dar. Alle M möglichen Buchstaben (Werte) α der Quelle $\mathcal{U}$ bilden ein Alphabet $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots, \alpha_M\}$. Jeder Buchstabe wird durch einen Repräsentanten (Codewort) κ dargestellt und besitzt eine Wahrscheinlichkeit von $p(\alpha_j)$. Alle Repräsentanten bilden zusammen das Codebuch $\mathcal{K}$. Die Länge eines Codeworts κ_j sei l_j. Die durchschnittliche Länge der binären Codewörter der Quellbuchstaben beträgt somit:

$$\bar{l} = \sum_{j=1}^{M} p_j l_j \,.$$

Für eine gegebene Wahrscheinlichkeitsverteilung der Quellwörter $\{p_j\}$ ergibt der Huffmancode eine minimale Länge $\bar{l}$ für einen eindeutig dechiffrierbaren Code [28].

Unter einem eindeutig dechiffrierbaren Code ϕ wird verstanden, dass für zwei Folgen von Quellwörtern $\underline{u} = \{u_1, u_2, \ldots, u_n\}$ und $\underline{\hat{u}} = \{\hat{u}_1, \hat{u}_2, \ldots, \hat{u}_{\hat{n}}\}$ mit möglicherweise ungleicher Länge n und $\hat{n}$ eine Abbildung $\phi : \mathcal{U} \rightarrow \hat{\mathcal{A}}$ existiert, so dass eine Aneinanderreihung von $\phi(u_1), \phi(u_2), \ldots, \phi(u_n)$ nur dann gleich $\phi(\hat{u}_1), \phi(\hat{u}_2), \ldots, \phi(\hat{u}_{\hat{n}})$ ist, wenn $\underline{u}$ gleich $\underline{\hat{u}}$ ist.

Um die Qualität der Codierung zu bestimmen, muss noch der Informationsgehalt I einer Nachricht x eingeführt werden:

$$I_a(\alpha) = \log_a \frac{1}{p(\alpha)} \,.$$

Für den binären Fall wird mit $a = 2$ der Informationsgehalt in Bit gemessen. Summiert man den Informationsgehalt aller Nachrichten der Quelle, gewichtet mit der Wahrscheinlichkeit der Nachricht, ergibt sich die Entropie:

$$H_a(\mathcal{U}) = \sum_{\alpha \,\epsilon\, \mathcal{U}} p(\alpha) \log_a \frac{1}{p(\alpha)} \,.$$

6.2 Fixed-Huffmandecoder

Die bisher vorgestellte Wavelettransformation war eine reine Überführung in ein anderes Basissystem und hat neben der hierarchischen Darstellung noch keinen zusätzlichen Gewinn. Allerdings bestehen die Detail-Koeffizienten bei einem lokal korellierten Ausgangssignal hauptsächlich aus Nullen und sind daher ausgezeichnet für eine Kompression geeignet. Mit dem Ziel der Implementierung in Hardware wurde ein Kompressionsschema gewählt, das ohne aufwendige arithmetische Operationen auskommt und auch keinen großen Speicherbedarf für ein etwaiges Code-Buch benötigt. Der verwendete Fixed-Huffmancode basiert dabei auf der Arbeit

von Guthe [31]. Für das Kompressionsschema wird angenommen, dass die Detail-Koeffizienten hauptsächlich aus Null-Koeffizienten bestehen. Es wird daher versucht eine Folge von Null-Koeffizienten möglichst kompakt zu repräsentieren.

Zuerst wird die Anzahl der aufeinander folgenden Nullen gezählt, wobei eine Folge aus mindestens einer Null und aus maximal 128 Nullen besteht. Folgen mit mehr als 128 Nullen werden in mehrere Teilfolgen zerlegt. Für jede Folge von Nullen wird im Kompressions-Bitstrom ein Byte reserviert, wobei das nullte Bit immer auf Null gesetzt wird und als Trennzeichen der Folge dient. In die restlichen 7 Bit wird die Anzahl der Nullen minus Eins in der Folge abgespeichert. Somit können bis zu 128 Koeffizienten in einem Byte komprimiert werden.
Folge von n Null-Koeffizienten ($0 < n \leq 128$):

Bit:	7	6	5	4	3	2	1	0
	$n-1$							0

Alle Koeffizienten, die nicht Null sind, werden einzeln abgespeichert. Als erstes wird die Anzahl an Bits n bestimmt, die notwendig ist um den Betrag des Koeffizienten zu speichern. Dann werden n Einsen an den Bitstrom angefügt und nach einer trennenden Null wird das Vorzeichen des d-Koeffizienten in ein Bit codiert. Abschließend wird der Betrag des d-Koeffizienten in n Bit an den Bitstrom angefügt. Diese Codierung führt zu einem Bitstrom aus dem sich eindeutig die Ausgangsdaten rekonstruieren lassen.
Einzelner Nichtnull-Koeffizient:

Bit:	2n + 2	$\cdots$	n + 2	n + 1	n	n - 1	$\cdots$	0
	$\|\|Koeff\|\|$			$sgn(Koeff)$	0	1	1	1

Die Kompressionsrate kann deutlich erhöht werden, indem Waveletkoeffizienten unterhalb eines Schwellwerts auf Null gesetzt werden, wodurch längere Nullfolgen entstehen. Diese Quantisierung führt natürlich zu einer verlustbehafteten Kompression bei der die Kompressionsrate gegen Bildqualität abgewogen werden muss. Eine qualitative Betrachtung zur Quantisierung und den daraus folgenden Bildfehlern kann in der Arbeit von Guthe [30, 31] gefunden werden. Typischerweise kann mit Quantisierung eine Kompressionsrate von 20–50 erreicht werden ohne größere sichtbare Artefakte in das Bild einzuführen. Bei verlustfreier Kompression wird typischerweise ein Kompressionsfaktor von 2–4 erreicht.

Durch die unterschiedliche Länge der codierten Koeffizienten ist eine parallele Verarbeitung des Eingangsstrom nicht möglich. Zu Beginn der Decodierung eines Codeworts ist die Länge nicht bekannt und somit die Startposition des nächsten Koeffizienten nicht vorhersagbar. Der Bitstrom wird daher Bit für Bit sequentiell abgearbeitet. Dies bedeutet für die Decodierzeit für einen kompletten Block, dass sie stark schwankt. Die Decodierung von Nicht-Null Koeffizienten ist dabei besonders zeitaufwendig. Für einen Koeffizienten dessen Betrag in n Bits codiert ist, benötigt der Decoder c_{NN} Zyklen:

$$c_{NN} = t_N + t_S + t_V + t_{SV} \,.$$

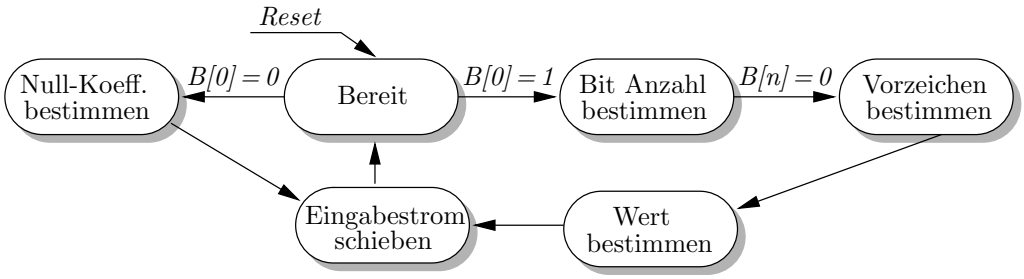


Abbildung 6.1: *Zustandsdiagramm für die Dekompression der Fixed-Huffman-Einheit.*

Wobei t_N die Anzahl der Zyklen, die benötigt werden um die Anzahl der direkt aufeinander folgenden Einsen und die trennende Null zu zählen, welche die Länge des Codeworts angeben, t_S die Anzahl des Zyklen für die Erkennung des Vorzeichenbit, t_V die Anzahl an Zyklen für das Lesen des Wertes und t_{SV} die Anzahl an Zyklen ist, um den Eingangsstrom um n-Bit zum Anfang des nächsten Koeffizienten zu schieben. Die Decodierung einer Folge von Nullen hat hingegen eine konstante Zeit. Sie benötigt:

$$c_N = t_L + t_{SN}$$

Hierbei steht t_L für die Anzahl an Zyklen, die für die Zählung der Folge benötigt wird, und t_{SN} gibt die Zyklen an, die benötigt werden um den Eingabestrom bis zum nächsten Koeffizienten weiter zu schieben. Die Verarbeitung von langen Nullfolgen kann dazu führen, dass der Decoder warten muss, bis alle Null-Koeffizienten am Ausgang weitergegeben wurden.

Eine deutliche Steigerung des Decodierdurchsatzes kann nur durch die Parallelisierung auf höherer Ebene geschehen. Durch die einfache Instantiierung von mehreren Einheiten können Blöcke parallel bearbeitetet werden. Allerdings benötigt dieser Schritt eine enge Abstimmung mit der Wavelettransformationseinheit, da diese die Daten mit der entsprechenden Geschwindigkeit weiterverarbeiten können muss.

6.2.1 Performanz und Ressourcenverbrauch

Der Decoder wurde in VHDL implementiert (vgl. Abb. 6.1) und anhand zweier Datensätze aus Kapitel 3 evaluiert. Gemessen wurde die Gesamtzeit in Zyklen, die benötigt wurden um einen Block d-Koeffizienten zu decodieren. Neben diesen zwei Datensätzen wurden noch zwei pathologische Datensätze verwendet, welche die minimalen und maximalen Zyklen pro Koeffizienten bestimmen sollten. Die minimale Decodierzeit ergibt sich, wenn alle Koeffizienten Null sind und somit der gesamte Block nur aus Nullfolgen besteht. Die maximale Decodierzeit ergibt sich, wenn alle Koeffizienten aus einem 16 Bit Wert bestehen. Die resultierenden Codewörter haben dann die maximale Länge von 34 Bit. Der eigentliche Wert des Koeffizienten ist dabei bedeutungslos.

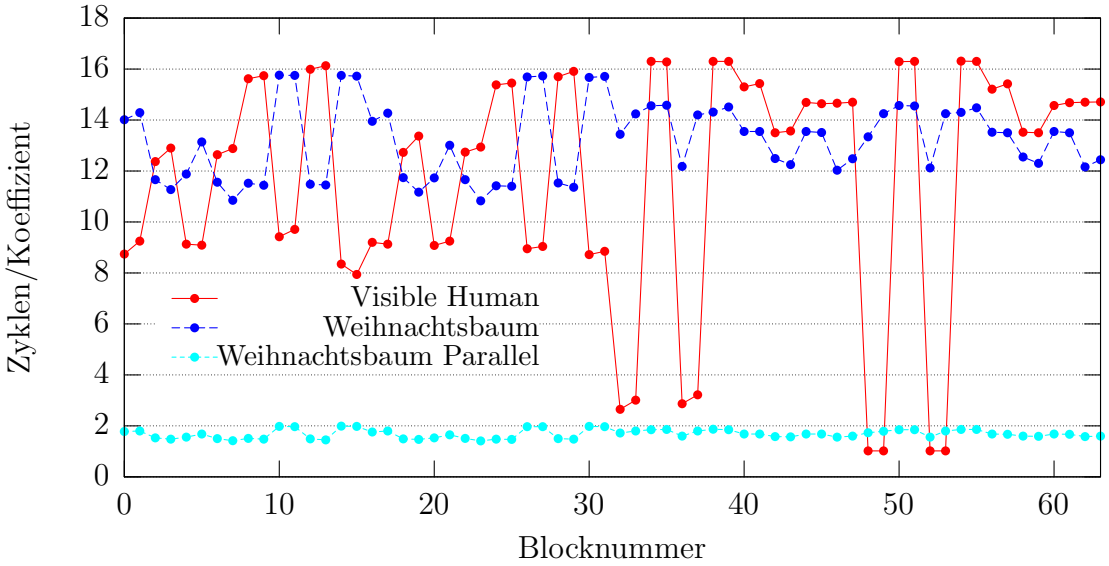


Abbildung 6.2: *Durchschnittliche Decodierzeit für die d-Koeffizienten von 64 zufällig ausgewählten Blöcken des Visible-Human und Weihnachtsbaum-Datensatzes, ermittelt anhand einer zyklenakkuraten VHDL-Simulation. Ebenfalls angegeben ist die Messung der parallelen Decodierung mit acht Instanzen der Decodiereinheit für die Blöcke des Weihnachtsbaums.*

Datensatz	c_{NN} *Zyklen / Koeff.*	$\sigma_{c_{NN}}$ *Zyklen / Koeff.*
Nullen	1,02	–
16 Bit-Koeff.	38,00	–
Visible Human	11,74	4,66
Weihnachtsbaum	13,21	1,48
Weihnachtsbaum 8x parallel	1,68	0,17

Für den „Visible Human“ und den „Weihnachtsbaum“-Datensatz wurden 64 zufällige 16^3 Blöcke extrahiert, wavelettransformiert und komprimiert. In Abbildung 6.2 ist die durchschnittliche Decodierzeit in Zyklen/Koeffizient pro Block angegeben. Um ein konservatives Ergebnis zu erreichen, wurde wie bisher auch auf die Quantisierung von Koeffizienten vor der Kompression verzichtet. Bei einer maximalen Taktfrequenz der Einheit von 160 MHz können somit zwischen 43 676 und 1 174 Blöcke pro Sekunde dekomprimiert werden. Dies entspricht einer Bandbreite zwischen minimal 10 MB/s und maximal 340 MB/s. Für die evaluierten Datensätze wird im Schnitt eine Bandbreite von ca. 3 700 Blöcken pro Sekunde (25 MB/s) erreicht. Da der durchschnittliche Durchsatz der Decodiereinheit deutlich unter dem der Transformationseinheit ist, bedeutet ein eins zu eins Verhältnis von Transformationseinheit und Decodiereinheit ein weitgehendes Ausbremsen der Transformationseinheit. Um einen maximalen Durchsatz an Blöcken zu erhalten, müssen daher mehrere Decodiereinheiten vor eine Transformationseinheit geschaltet werden. Die Parallelisierung der Decodiereinheit kann auf zwei Ebenen erfolgen. Da die L-Blöcke unabhängig voneinander sind, könnten mehrere Blöcke parallel decodiert werden.

FPGA Nutzung für:	*Virtex-II 2V4000ff1152*			*Virtex-II 2V8000ff1517*		
Ressource	*Benutzt*	*Verfügbar*	*Nutzung*	*Benutzt*	*Verfügbar*	*Nutzung*
IOs	0	0	0,00 %	0	0	0,00 %
Global Buffers	1	16	6,25 %	1	16	6,25 %
Function Generators	221	46 080	0,48 %	225	93 184	0,24 %
CLB Slices	111	23 040	0,48 %	113	46 592	0,24 %
Dffs or Latches	217	48 132	0,45 %	215	96 508	0,22 %
Block RAMs	0	120	0,00 %	0	168	0,00 %
Block Multipliers	0	120	0,00 %	0	168	0,00 %
Block Multiplier Dffs	0	4 320	0,00 %	0	6 048	0,00 %

Tabelle 6.1: *Syntheseergebnisse des Huffmandecoders für zwei FPGA-Typen. Gegenübergestellt ist der momentan verwendete Virtex-II 4000 und der aktuell Virtex-II 8000. Die maximale Taktfrequenz der Einheit beträgt 160 MHz.*

Dies erfordert aber große Zwischenspeicher für die L-Blöcke vor der Transformationseinheit. Dieser Ansatz ist daher zwar elegant, aber nicht effektiv. Somit bleibt nur die Decodierung innerhalb eines Blockes zu parallelisieren. Da die Codewörter unabhängig voneinander sind, ist dies auch leicht möglich, solange der jeweilige Startpunkt des nächsten Codeworts bekannt ist. Um die Effizienz dieser Parallelisierung zu testen, wurden die Blöcke des Weihnachtsbaums bei der Codierung nicht in einem Block, sondern in acht Subblöcken mit jeweils 448 Koeffizienten gespeichert. Die Decodierzeit für den gesamten Block ist somit das Maximum der Decodierzeiten der jeweiligen Subblöcke. Die Verwendung von acht Instanzen ist hinsichtlich der Ressourcen unproblematisch, da die benötigten Ressourcen einer Einheit unter 0,5 % der vorhandenen FPGA-Ressourcen und somit minimal sind (vgl. Tab. 6.1).

Der Fixed-Huffmancode stellt folglich eine effiziente Möglichkeit dar, die Decodierung der Detail-Koeffizienten in Hardware durchzuführen. Somit können alle Schritte, die zur Verwendung des Multiskalenmodells erforderlich sind, vollständig in Hardware erfolgen. Der Host-Computer muss dadurch nur die in einem Vorverarbeitungsschritt notwendige Berechnung der Detail-Koeffizienten vornehmen, zur Laufzeit die Hierarchie verwalten und die komprimierten Koeffizienten an die Hardware schicken.

6.3 3D-Huffmandecoder

Für die Kompression in Echtzeit generierter dynamischer Volumendatensätze ist die vorhergehenden Abschnitt beschrieben Kompression nicht geeignet. Häufig ist die damit verbundene aufwendige Vorverarbeitung auf bestehenden Datengenerierungssystemen nicht machbar, und auch keine Integration dedizierter Hardware zur Kompression möglich. So wurden z. B. im EU-Projekt DynCT [23, 6] die Volumendaten in einem Computertomograph aufgenommen und von einem integrierten Server in Echtzeit auf den Viusalisierungsclient übertragen. Eine geringstmögliche Latenz zwischen Aufnahme und Visualisierung der Daten ist bei einem solchen intraoperati-

ven Szenario von höchster Bedeutung. Die Datensätze, mit einer Größe von 3,2 MB, wurden mit 12 Hz von dem CT-Scanner geliefert und mit der VIZARD II-Karte visualisiert. Dies resultiert in einer Bandbreite von 58 MB/s. Um diese Datenrate deutlich zu reduzieren wird im Folgenden ein Codierverfahren präsentiert, das eine gute Kompressionsrate bei minimaler Latenz bietet und im Bezug auf serverseitige Ressourcen geringe Anforderungen stellt [5].

Überraschenderweise gibt es so gut wie keine Arbeiten, die sich speziell mit der verlustfreien Komprimierung von Volumendaten beschäftigen. So wird zwar in der Arbeit von Fowler et al. [24] ein solches Kompressionsschema vorgestellt, der Schwerpunkt liegt aber auf der Maximierung der Kompressionsrate ohne dabei die Laufzeit zu berücksichtigen. So werden insgesamt vier Durchläufe über den Datensatz benötigt, was die Latenz dramatisch erhöht. Des Weiteren wurde ein Ansatz publiziert, der den Datensatz in Subblöcke unterteilt und diese in den Frequenzraum transformiert [15]. Die Subblöcke werden dann aber direkt aus ihrer komprimierten Darstellung visualisiert, was dieses Verfahren für die, dieser Arbeit zugrunde liegenden Rahmenbedingungen unmöglich macht.

Als Basis der im Folgenden vorgestellten Kompression wurde ebenfalls das Huffmanschema, wie in Abschnitt 6.1 vorgestellt, benutzt. Der präfixfreie Code erlaubt eine geordnete Suche der Codewörter ohne aufwendige Arithmetik und erleichtert so die Implementierung in Hardware. Um die Kompressionsrate weiter zu erhöhen, wird noch Zusatzinformation zur Codierung verwendet. Da die Quellwörter nicht 1D sondern 3D vorliegen, wird die lokale räumliche Korrelation der Daten verwendet, wie sie auch in anderen „Differential pulse-code modulation“ (DPCM) Methoden [28] Anwendung findet. Es wird ein einfaches Differenz-Modell angenommen, bei dem der Vorhersagewert (Predictor) ein bereits verarbeitetes Nachbarvoxel ist:

$$\hat{\alpha}_n = \begin{cases} \alpha_n & : \quad n = 0 \\ \alpha_n - \alpha_{n-t^2} & : \quad n = i \cdot t^2, i \,\epsilon\, \mathbb{N} \\ \alpha_n - \alpha_{n-t} & : \quad n = i \cdot t, \;\; i \,\epsilon\, \mathbb{N} \\ \alpha_n - \alpha_{n-1} & : \quad \text{sonst} \; . \end{cases}$$

Die Reihenfolge der Bearbeitung wurde in der üblichen x-, y- und z-Richtung vorgenommen, um das sequentielle Übertragungsschema der Datensätze beizubehalten und keine Umsortierung vornehmen zu müssen. Allerdings müssen so jeweils ein Referenzvoxel der letzten Reihe und der letzten Schicht zwischengespeichert werden. Um eine einfache Parallelisierung der Decodierung zu ermöglichen, wurde der Datensatz in unabhängige Subblöcke der Kantenlänge $t = 4$ Voxel zerlegt.

Da die Wahrscheinlichkeitsverteilung eines Datensatzes per se nicht bekannt ist, besteht die Generierung des Huffmancodes aus zwei Schritten. Zuerst wird die Anzahl eines jeden Buchstaben α_i im Datensatz gezählt und so die Wahrscheinlichkeitsverteilung bestimmt. Danach wird mit dieser Verteilung die Codierung vorgenommen. Für statische Datensätze stellt dieses Verfahren kein Problem dar, aber für dynamische Daten erzeugt dieser Vorverarbeitungsschritt zusätzliche Latenz. Eine Lösung wäre ein adaptiver Huffmancode [78], der seinen Binärbaum zur Decodierzeit verändert. Da dies aber eine erhebliche Komplexität für den Hardwaredecoder bedeuten würde, wurde von dieser Möglichkeit Abstand genommen.

Stattdessen wurde zusätzlich die zeitliche Kohärenz der Daten genutzt, indem nicht die aktuelle Wahrscheinlichkeitsverteilung des Zeitschritts n zur Codierung benutzt wird, sondern die des vorhergehenden Zeitschritt $n-1$. Die Erstellung der Wahrscheinlichkeitsverteilung und die Codierung können somit gleichzeitig erfolgen. Die Kompressionsrate ist aber nicht mehr optimal, da die Codierung auf einer partiell inkorrekten Wahrscheinlichkeitsverteilung beruht. Anhand dynamischer CT-Daten wird gezeigt, dass die Zeitschritte eines kontinuierlichen Stroms genügend Kohärenz besitzen um diesen Nachteil nicht ins Gewicht fallen zu lassen.

6.3.1 Implementierung

Die Decodierung des Bitstroms lässt sich prinzipiell auf zwei Arten durchführen. Entweder können simultan alle Codewörter des Codebuchs gesucht werden oder das Codewort wird durch bitweises Testen des bei der Huffmancodierung entstehenden Binärbaums ermittelt. Die erste Methode hat den Vorteil, dass die Suche nach dem entsprechenden Codewort immer innerhalb eines Zyklus erfolgreich ist. Somit benötigt ein Bitstrom $C_k = (k+1) \cdot n$ Zyklen, wobei k die Anzahl an Zyklen ist, die pro gefundenem Codewort benötigt werden um das Codewort aus dem Bitstrom zu entfernen. Der Aufwand an Komparatoren, die jeweils ein Codewort mit dem Bitstrom vergleichen, steigt aber linear mit der Anzahl der Codewörter im Codebuch und liegt daher schon bei einem 8-Bit Datensatz bei $2^{8+1}-1$ Differenzwerten und somit bei 511 Instanzen. Die zweite Art des Durchwanderns des Binärbaums benötigt dagegen nur einen Komparator und eine feste Größe an Speicher, in dem der Binärbaum abgelegt wird. Die Anzahl der Taktzyklen zum Finden eines Codeworts kann aber für jedes Codewort unterschiedlich ausfallen. Die Anzahl der Zyklen zur Bestimmung eines Quellworts ist gleich $C_i = (m \cdot l_i) + k$, wobei m die konstante Anzahl an Zyklen ist, die benötigt wird um im Baum eine Ebene tiefer zu steigen. Die durchschnittliche Anzahl der Zyklen $\overline{C}_d$ der Decodierung lässt sich somit für den kompletten Bitstrom berechnen: $\overline{C}_d = (m \cdot \bar{l}) + k$.

Der schematische Aufbau des Huffmandecoders ist in Abbildung 6.3 dargestellt. Die Ein- und Ausgangsfifo dient zur Entkoppelung des Decoders von den vorhergehenden und folgenden Einheiten. Kernkomponenten sind das Schieberegister und die Komparatormodule. Das Schieberegister ermöglicht das Extrahieren von Codewörtern bis zu einer Länge von 32 Bit. Nach dem Finden des Codeworts durch die Komparatormodule wird der Wert des Schieberegisters um die l_C Bit des Codeworts nach rechts geschoben. Ist die Summe S_l der Codewortlängen seit dem letzten Lesen aus der Eingangsfifo gleich 32 Bit, wird ein neuer 32 Bit Wert aus der Eingangsfifo in die oberen 32 Bit des Schieberegisters geladen. Danach wird die Schiebeoperation durchgeführt. Ist die S_l größer als 32 Bit, muss das Schieben in zwei Schritten erfolgen. Zuerst wird um die Anzahl an Bit geschoben, die nötig ist um auf eine Summe von 32 Bit zu kommen. Danach werden die oberen 32 Bit geladen und der Rest der Schiebeoperation ausgeführt. Die untersten 32 Bit des Schieberegisters stehen als Eingabe den Komparatormodulen zur Verfügung. Nachdem die Komparatormodule ein Codewort gefunden haben, wird das zugehörige Quellwort in eine Ausgangsfifo

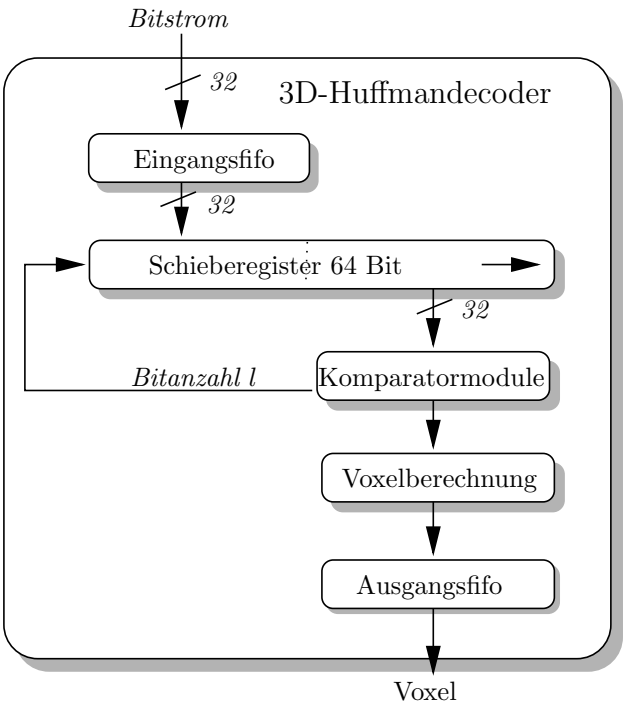


Abbildung 6.3: *Schematischer Aufbau des 3D-Huffmandecoders. Das Schieberegister ermöglicht einen garantierten Zugriff auf konstante 32 Bit für die Komparatormodule.*

geschrieben. Für die Erkennung von Codewörtern stehen zwei verschiedene Module in Form eines Paralleldecoders und eines Sequentielldecoders zur Verfügung.

Paralleldecoder

Der Paralleldecoder vergleicht alle unteren 32 Bit des Schieberegisters simultan. Hierfür enthält er ein Codewort C, dessen Länge l_C, eine Maske M sowie das passende Quellwort. Codewort, Maske und Quellwort werden zur Initialisierungsphase geladen und sind für den Verlauf des Decodierens fix. In der Maske sind ausschließlich die Bits b_0–b_{l_C-1} gesetzt. Der Paralleldecoder ermittelt anhand folgender Logik sein Ergebnis: $E = (C \wedge M) \oplus B$.
Eine Null als Ergebnis bedeutet dabei eine Übereinstimmung zwischen Codewort und Bitstrom B. Mehrere Paralleldecoder können auf einmal instantiiert werden.

Sequentielldecoder

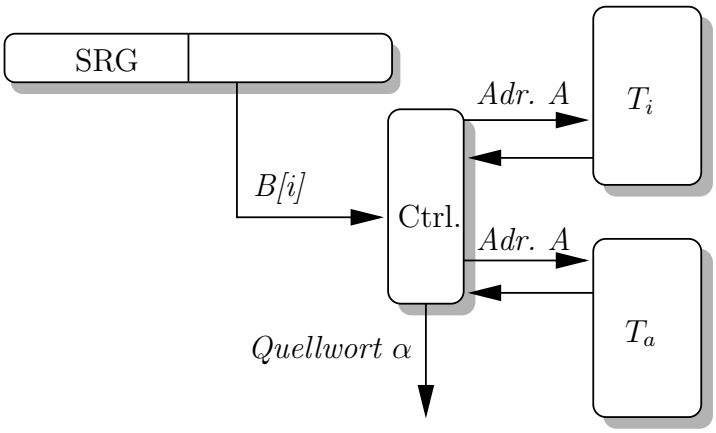
Der Sequentielldecoder versucht Codewörter im Bitstrom durch das sequentielle Testen von einzelnen Bit zu finden (vgl. Abb. 6.4). Er benutzt hierzu die Baumstruktur die zur Erzeugung des Huffmancodes benutzt wurde.

Aufbau der Baumstruktur

Um den Baum effizient abspeichern zu können, wird die Baumstruktur in zwei Tabellen für innere und äußere Knoten abgelegt [79]. Die erste Tabelle mit inneren Knoten des Binärbaums dient zur Traversierung. Sie enthält für jeden Knoten zwei Einträge, die angeben an welcher Adresse sich die Kindknoten in der Tabelle befinden. Im Folgenden ist ein Tabelleneintrag dargestellt.

G_l	Adresse linkes Kind A_l		G_r	Adresse rechtes Kind A_r	
31	30	$\cdots$ 16	15	14	$\cdots$ 0

Die Tabellenbits G_l und G_r geben an, ob es sich bei dem Kindknoten um einen inneren oder äußeren Knoten handelt und somit auf welche Tabelle sich die Adresse bezieht. Ein gesetztes Bit kennzeichnet einen äußeren Knoten, ein nicht gesetztes Bit einen inneren Knoten. Für jeden Knoten werden 32 Bit alloziert. Die zweite Tabelle enthält alle äußeren Knoten und somit die Quellwörter. Die Länge l_C des dazu gehörenden Codeworts wird nicht extra abgespeichert, sondern zur Laufzeit bei der Traversierung mitgezählt. Die Traversierung der Baumstruktur erfolgt nur von der Wurzel zu den Blättern und ist somit nur einfach verkettet.




```
1:  i := 0
2:  A := 0
3:  E := T_i[A]
4:  if B[i] = 0 then
5:      A := E[A_l]
6:      TB := E[A_l]
7:  else
8:      A := E[A_r]
9:      TB := E[A_r]
10: end if
11: i := (i + 1) % 32
12: if TB = 0 then
13:     goto 3:
14: else
15:     α := T_a[A]
16:     goto 2:
17: end if
```

Abbildung 6.4: *Aufbau des Sequentielldecoders mit den zwei SRAM-basierten Tabellen für innere (T_i) und äußere Knoten (T_a), sowie Ablauf der Traversierung zum Finden der Codewörter anhand eines Bits ($B[i]$) an der Position i aus dem Schieberegister (SRG).*

Laden der Tabellen

Die Tabellen werden vor dem Decodieren für jeden Datensatz einmal geladen. Die ersten n Codewörter mit der höchsten Wahrscheinlichkeit $p(\alpha_j)$ werden automatisch

während des Ladens in die n zur Verfügung stehenden Paralleldecoder geladen. Die notwendigen Masken werden dabei dynamisch erzeugt. Die restlichen Codewörter werden dann in die SRAM-basierten Tabellen geladen. Die Größe der Tabellen kann hierbei variieren. Für statische 3D-Datensätze müssen sich in der Tabelle nur die tatsächlich vorkommenden Differenzwerte widerspiegeln. Bei dynamischen Datensätzen können im abhängigen Modus aber auch Differenzwerte auftreten, die bei der Erzeugung der Tabelle nicht vorhanden waren, da die Tabelle nicht mit dem aktuell verwendeten Datensatz erzeugt wurde. Des Weiteren ist die Tabelle natürlich von der Anzahl der Bits eines Quellworts (Differenzwerts) l_Q abhängig. Die maximale Länge der Codewörter l_C ist auf 32 Bit beschränkt, womit die l_C in $i = 5$ Bit abgelegt werden kann.

Quellwort	Codewort	l_C
$i + l_C + l_Q - 1 \cdots i + l_C$	$i + l_C - 1 \cdots i$	$i - 1 \cdots 0$

Bei den üblichen 8 Bit Datensätzen ergibt sich so eine maximale Tabellengröße für die äußeren Knoten von 2,88 kB, bei $l_C = 32$, $i = 5$, $l_Q = 8 + 1$ und 2^{8+1} möglichen Tabelleneinträgen. Für 12 Bit Datensätzen vergrößert sich die Tabelle auf maximale 50 kB, mit $l_C = 32$, $i = 5$, $l_Q = 12 + 1$ und 2^{12+1} möglichen Tabelleneinträgen. Die Größe der Tabelle für die inneren Knoten berechnet sich ebenfalls aus der Anzahl an Quellworten. Für jeden der maximal $2^{l_Q+1} - 2$ inneren Knoten wird einen Eintrag von 4 Byte benötigt. Dies ergibt maximal 2 kB und 32 kB für 8 Bit bzw. 12 Bit Datensätze.

6.3.2 Performanz und Ressourcenverbrauch

Im folgenden Abschnitt soll eine Evaluierung des eben vorgestellten Codierungsverfahrens vorgestellt werden. Hierfür wird die Kompressionsrate für verschiedene 3D- und 4D-Datensätze aus den wichtigsten Anwendungsbereichen der Volumenvisualisierung (Medizin, Materialforschung und wissenschaftliche Simulation) bestimmt, sowie der Einfluß der Anzahl an Parallel- und Sequentielldecoder auf die Geschwindigkeit und den Ressourcenverbrauch ermittelt.

Auch wenn der 3D-Huffmancode nicht in erster Linie für statische 3D-Daten gedacht ist, soll dennoch zunächst die Kompressionsrate für einen Satz bekannter 3D-Datensätze ermittelt werden. In Tabelle 6.2 ist für die in Abbildung 6.5 dargestellten Datensätze die durchschnittliche Länge der Codewörter $\bar{l}$ sowie der Kompressionsfaktor $k = l_{orig}/\bar{l}$ angegeben, wobei l_{orig} die konstante Länge eines Datensatz in Bit pro Voxel ist. Zusätzlich ist zur Orientierung noch $\bar{l}$ der Kompression mit „gzip“ [25] als Referenzwert angegeben, das auf dem Verfahren von Lempel und Ziv (LZ) [94] basiert. In allen Ergebnissen ist die Größe der Tabelle zum Decodieren mit eingeschlossen, um ein vergleichbares Ergebnis zu erhalten.

Die Datensätze erreichen sehr gute Kompressionsraten, die fast die Ergebnisse von gzip erreichen. Für sehr homogene Datensätze (Aneurisma und Motorblock) bleiben die Kompressionsraten aber zurück. Dies rührt daher, dass es eine untere Grenze für die Kompression des 3D-Huffmancodes gibt. Ein Block besteht im besten

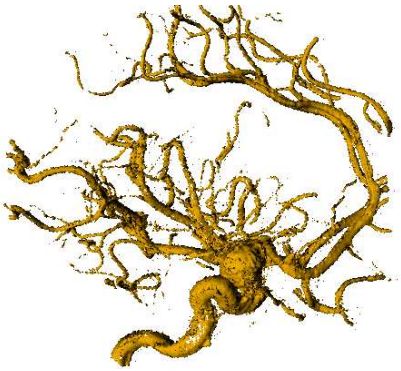
Datensatz	$\bar{l}$ / $\frac{Bit}{Voxel}$	k	$\bar{l}_{gzip}$ / $\frac{Bit}{Voxel}$	h_{PD} / %		t_{aa} / *Zyklen*			
				5 PD	*10 PD*	*0 PD*	*5 PD*	*10 PD*	*511 PD*
Aneurisma	1,18	6,78	0,15	97,32	97,41	5,12	2,19	2,18	1,98
Thorax	5,04	1,59	4,83	39,25	55,59	14,20	11,30	9,42	1,98
Motorblock	3,51	2,28	3,02	78,78	87,44	9,84	5,64	4,44	1,98
Fuß	2,80	2,86	2,15	73,12	77,23	8,41	5,89	5,36	1,98
Diesel	1,39	5,76	0,16	94,97	95,88	5,49	2,53	2,41	1,98
Neghip	3,01	2,65	2,40	77,69	85,68	8,63	5,24	4,29	1,98

Tabelle 6.2: *Durchschnittliche Länge eines Codeworts $\bar{l}$, Kompressionsrate k und durchschnittliche Länge eines Codeworts mit gzip $\bar{l}_{gzip}$ für verschieden statische 3D-Volumendatensätze (vgl. Abb. 6.5) nach Kompression mit dem 3D-Huffmancodierer. Zusätzlich ist der Anteil der durch die Paralleldecoder (PD) gefundenen Codewörter unter Verwendung von 5 bzw. 10 Paralleldecodern aufgeführt, sowie die durchschnittliche Anzahl an Zyklen t_{aa} angegeben, die benötigt wird um ein Quellwort zu einem Codewort zu finden bei der Benutzung von 0, 5, 10 und 511 Paralleldecodern.*

Fall aus einem 8 Bit Referenzvoxel sowie 63 Differenzwerten, die in 1 Bit Codewörtern codiert wurden. Somit ist die kleinste durchschnittliche Länge der Codewörter für einen Block: $\bar{l}_{min} = \frac{(8+63)\,Bit}{64\,Voxel} = 1,11\frac{Bit}{Voxel}$. Somit erreichen diese homogenen Datensätze fast das theoretische Limit.

Das primäre Ziel ist aber nicht, eine möglichst hohe Kompressionsrate zu erzielen. Vielmehr ist die Frage des hohen Durchsatzes und der geringen Latenz ausschlaggebend. Hierzu wurden mehrere zyklenakkurate VHDL-Simulationen mit einer unterschiedlichen Anzahl an schnellen Paralleldecodern, die ihr Codewort in einem Zyklus finden können, durchgeführt und der Anteil der Codewörter bestimmt, der durch die Paralleldecoder gefunden wurde (h_{PD}) (vgl. Tab. 6.2). Obwohl 5–10 Paralleldecoder nur 1–2 % der möglichen Codewörter eines 8 Bit Datensatzes abdecken, ist hier eine typische Eigenschaft des Huffmancodes zu beobachten. Die fünf häufigsten Codewörter decken bereits 39–97 % der vorkommenden Codewörter ab; bei 10 Paralleldecodern steigt dieser Wert auf 55–97 %. Deutliche Unterschiede erkennt man auch in der durchschnittlichen Zeit, die es dauerte um ein Codewort zu finden (t_a). Werden keine Paralleldecoder instantiiert und nur die sequentielle Baumsuche verwendet, braucht es durchschnittlich 5–14 Zyklen um ein Codewort zu ermitteln und das dazugehörige Quellwort zu bestimmen. Werden alle 511 möglichen Codewörter mittels Paralleldecoder gesucht, ergibt sich ein konstante Suchzeit von 1,98 Zyklen pro Codewort. Dies ergibt sich aus der konstanten Suchzeit von einem Zyklus des Paralleldecoders und einem Zyklus der Notwendig ist, um den Bitstrom weiter zu schieben. Beide Operationen können auch innerhalb eines Zyklus erfolgen. Die maximal erreichbare Taktung der Einheit wird dann aber um mehr als die Hälfte reduziert und würde somit zu einer effektiven Reduktion des Datendurchsatzes führen. Bei 5 und 10 Paralleldecodern steigt diese Zahl auf 2–9 Zyklen, abhängig vom Datensatz.

Der 3D-Huffmancode wurde im Hinblick auf kontinuierlich gelieferte Daten („Streaming Data“) entwickelt. Um die Effizienz hinsichtlich dieses Bereichs zu testen,



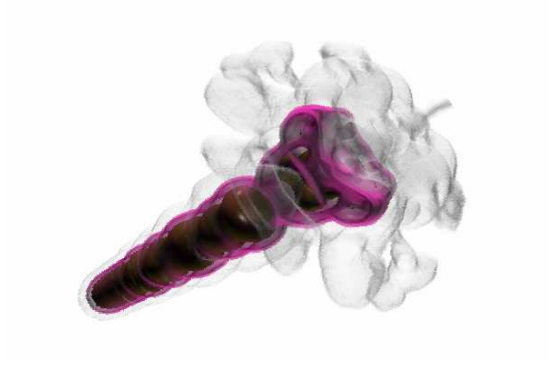
(a) Mittels Kontrastmittel sichtbar gemachte Arterie der rechten Gehirnhälfte in der ein Aneurisma erkennbar ist [69] (256^3).

(b) CT-Scan eines Thorax [71]. Der vordere Teil des Oberkörpers wurde mit einer Schnittebene entfernt ($512^2 \cdot 259$).

(c) Rotations Röntgenaufnahme eines Fußes [70] (256^3).

(d) CT-Scan eines Motorblocks mit zwei Zylindern [27] ($256^2 \cdot 128$).

(e) Wissenschaftliche Simulation der Aufenthaltswahrscheinlichkeit der Elektronen eines Proteinmoleküls [85] (64^3).



(f) Simulation einer Dieseleinspritzung in eine Brennkammer [80] (64^3).

Abbildung 6.5: *Statische 3D-Datensätze aus den Bereichen Medizin, Materialforschung und wissenschaftliche Simulation, die zur Evaluierung des Kompressionsverfahrens in Tabelle 6.2 verwendet wurden. In Klammern ist die Datensatzgröße angegeben. Alle Bilder wurden auf der VIZARD II-Karte berechnet.*

Datensatz	$\bar{l}$ / $\frac{Bit}{Voxel}$	σ / $\frac{Bit}{Voxel}$	k	$\bar{l}_{gzip}$ / $\frac{Bit}{Voxel}$	σ_{gzip} / %
Diesel (unabh.)	1,186	0,01	6,75	0,15	0,00
Diesel (abh.)	1,189	0,01	6,73	–	–
DynCT (unabh.)	2,067	0,05	3,87	2,37	0,07
DynCT (abh.)	2,067	0,05	3,87	–	–

(a) Durchschnittliche Länge der Codewörter $\bar{l}$ für zwei 4D-Datensätze, die zugehörige Standardabweichung σ und der Kompressionfaktor k der Zeitserie, sowie die entsprechenden Werte ermittelt mit gzip.

Datensatz	h_{PD} / %		t_{aa} / *Zyklen*			
	5 PD	*10 PD*	*0 PD*	*5 PD*	*10 PD*	*511 PD*
Diesel	99,58	99,72	8,31	2,18	2,18	1,98
DynCT	95,93	98,22	8,00	2,87	2,51	1,98

(b) Anteil der durch Paralleldecoder gefundenen Codewörter h_{PD} und durchschnittlichen Suchzeit t_{aa} in Abhängigkeit der Anzahl der verwendeten Paralleldecoder.

Tabelle 6.3: *Simulationsergebnisse der VHDL-Implementierung des Fixed-Huffmandecoders für zwei 4D-Datensätze (vgl. Abb. 6.7 und 6.9).*

wurden zwei 4D-Datensatzreihen verwendet. Die erste Animation stammt aus dem Forschungsgebiet der „Smoothed Particle Hydrodynamics“ (SPH) und stellt in 64 Zeitschritten die Einspritzung eines Dieselgemischs in eine Brennkammer dar [26]. In Abbildung 6.7 sind drei Zeitschritte aus dieser Animation dargestellt, die auf der VIZARD II Beschleunigerkarte berechnet wurde. In der SPH-Simulation werden verschiedene physikalische Größen, wie Temperatur, Dichte, Druck und Geschwindigkeit berechnet. In der Animation ist nur die Dichteverteilung und deren Entwicklung über die Zeit abgebildet.

Der zweite Datensatz entstammt der klinischen Erprobungsphase des DynCT-Projekts und stellt eine Testaufnahme einer Biopsienadel mit einem neuartigen 4D-CT Scanner im Flouroscopy Modus dar. Die Animation umfasst 100 Zeitschritte und wurde ebenfalls auf dem VIZARD II berechnet (vgl. Abb. 6.9). In ihr ist das Einschieben einer Biopsienadel in ein Fleischstück, das auf ein Brett mittels eines Nagels befestigt ist, zu sehen.

Um die Performanz zu messen, wurde der Kompressionsfaktor jedes einzelnen Zeitschritts in zwei Modi ermittelt. Im ersten Modus sind alle Zeitschritte unabhängig von einander, d.h. es wird wie im statischen Fall die Wahrscheinlichkeitsverteilung des jeweiligen Datensatzes zur Generierung des Huffmancodes benutzt. Dies hat den Nachteil, dass die Wahrscheinlichkeitsverteilung in einem Vorverar-

FPGA Nutzung für:	*Virtex-II 2V4000ff1152*			*Virtex-II 2V8000ff1517*		
Ressource	*Benutzt*	*Verfügbar*	*Nutzung*	*Benutzt*	*Verfügbar*	*Nutzung*
IOs	0	0	0,00 %	0	0	0,00 %
Global Buffers	1	16	6,25 %	1	16	6,25 %
Function Generators	3 187	46 080	6,92 %	3 187	93 184	3,42 %
CLB Slices	1 594	23 040	6,92 %	1 594	46 592	3,42 %
Dffs or Latches	1 411	48 552	2,91 %	1 411	96 508	1,46 %
Block RAMs	3	120	2,50 %	3	160	1,86 %
Block Multipliers	0	120	0,00 %	0	168	0,00 %
Block Multiplier Dffs	0	4 320	0,00 %	0	6 048	0,00 %

Tabelle 6.4: *Syntheseergebnisse des Huffmandecoders mit zehn Paralleldecodern für zwei FPGA-Typen. Gegenübergestellt ist der momentan verwendete Virtex-II 4000 und der aktuelle Virtex-II 8000. Die Einheit erreicht eine maximale Taktrate von 169 MHZ.*

beitungsschritt vor der Codierung ermittelt werden muss, was die Latenz deutlich erhöht. Dieses Problem wird im abhängigen Modus dadurch umgangen, dass die Wahrscheinlichkeitsverteilung des vorhergehenden Datensatzes zur Codierung verwendet wird. Die Wahrscheinlichkeitsverteilung kann parallel mit der Decodierung erfolgen und verursacht somit keine zusätzliche Latenz. Da die einzelnen Zeitschritte eine hohe räumliche Kohärenz besitzen, sollten sie sich in ihrer Wahrscheinlichkeitsverteilung und somit in ihrem Kompressionsfaktor kaum unterscheiden.

Um diese Annahme zu überprüfen, wurde für jeden Zeitschritt der 4D-Animation die Kompressionsrate für den unabhängigen und abhängigen Modus berechnet. Wie man in Abbildung 6.6 und Abbildung 6.8 erkennt, wird im abhängigen Modus nahezu dieselbe Kompressionsrate wie im unabhängigen Modus erreicht. Dies wird auch durch die nahezu identische durchschnittliche Codierungslänge $\bar{l}$ und die Standardabweichung σ über die Zeitschritte bestätigt (vgl. Tabelle 6.3).

Mittels eines Synthese-Programms [61] wurde der Ressourcenverbrauch und die maximale Taktfrequenz eines 3D-Huffmandecoders mit 7 Paralleldecodern ermittelt. Wie aus der Tabelle 6.4 abzulesen, ist der Ressourcenverbrauch der Einheit mit bis zu 7 % gering, so dass auch eine parallele Decodierung in Bereichen in denen ein höherer Datendurchsatz verlangt wird durchaus möglich ist. Durch das blockbasierte Schema wären keinerlei Änderungen an der Decodiereinheit von Nöten. Des Weiteren würde sich die Anzahl der verwendeten Block RAMs nur alle zwei Einheiten ändern, da die Tabellen durch ihre „dual-Ports“ von jeweils zwei Sequentielldecodern parallel verwendet werden können. Bei einer maximalen Taktfrequenz des 3D-Huffmandecoders von 169 MHz können mit nur einer Instanz des 3D-Huffmandecoders der Diesel- sowie der DynCT-Datensatz mit 50 Hz bzw. 12 Hz decodiert werden.

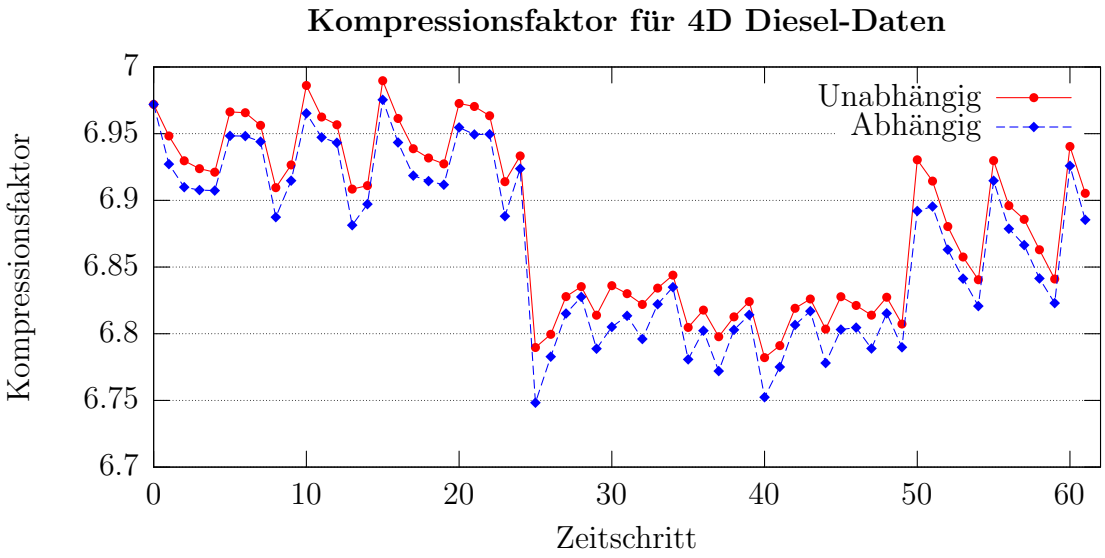
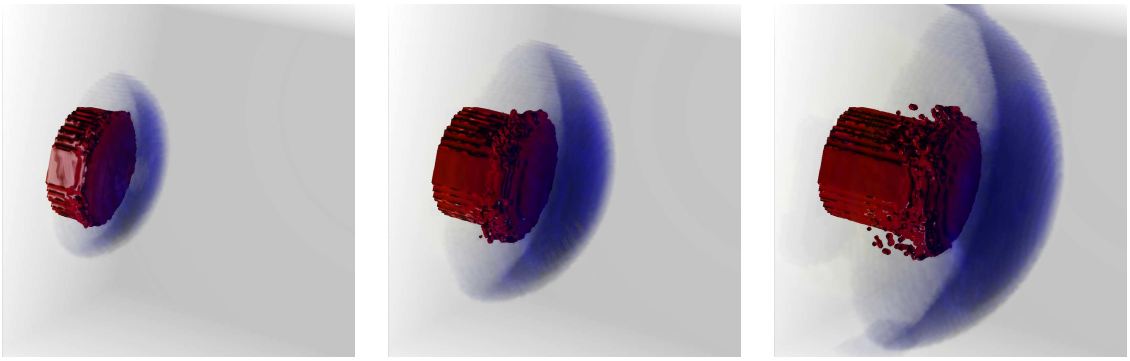


Abbildung 6.6: *Kompressionsfaktor für die zeitabhängigen Diesel-Daten aus Abb. 6.7. Bei der abhängigen Zeitreihe sind für jeden Zeitschritt n die Huffmantabelle des Zeitschritt $n+1$ verwendet. Bei der unabhängigen Zeitreihe wurde die Tabelle des jeweiligen Zeitschritts selbst zur Kompression verwendet.*



(a) ZS 20 *(b) ZS 40* *(c) ZS 63*

Abbildung 6.7: *Drei Zeitschritte (ZS) eines animierten Volumendatensatzes. Per SPH-Verfahren wird die Einspritzung eines Dieselgemisches simuliert. Dargestellt ist die Dichteverteilung des eingespritzten Dieselgemisches. In rot ist der Primärstrahl und in blau die sich bildende Druckwelle farbcodiert (100^3 Voxel).*

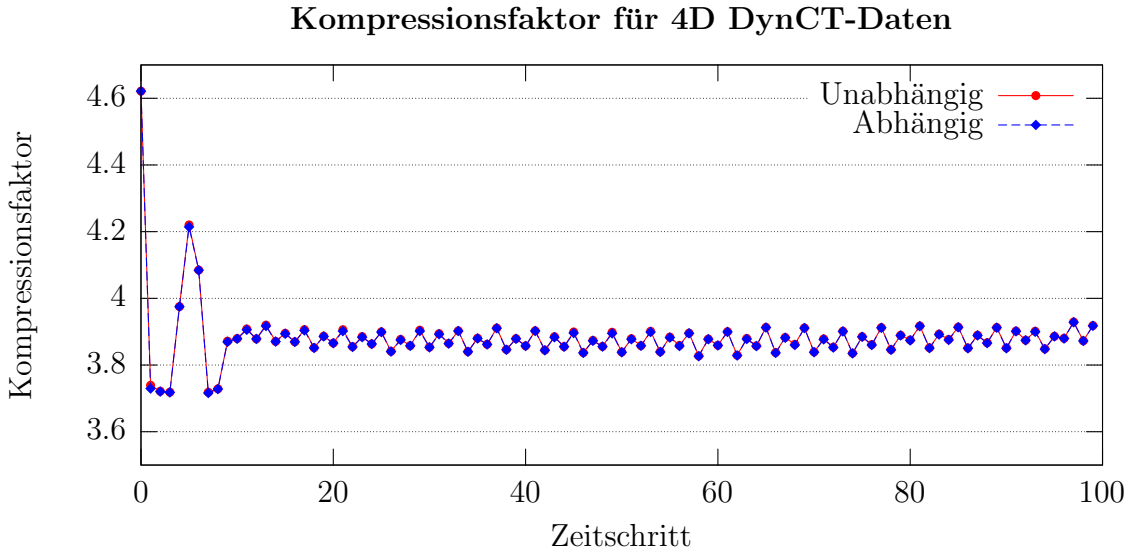


Abbildung 6.8: *Kompressionsfaktor für die zeitabhängigen DynCT-Daten aus Abb. 6.9. Bei der abhängigen Zeitreihe wird für jeden Zeitschritt n die Huffmantabelle des Zeitschritts $n+1$ verwendet. Bei der unabhängigen Zeitreihe wurde die Tabelle des jeweiligen Zeitschritts selbst zur Kompression verwendet, was zu fast identischen Ergebnissen führt. Die anfänglich starke Variation des Kompressionsfaktors ist auf das Rekonstruktionsverfahren der CT-Daten zurückzuführen, das mehrere Umläufe des Scanners benötigt um einen Datensatz zu generieren.*

(a) ZS 10 *(b) ZS 50* *(c) ZS 99*

Abbildung 6.9: *Testaufnahmen eines animierten Datensatzes des DynCT-Projekts. Mithilfe eines Fleischstücks das auf ein Holzbrett geschraubt ist, wurde in vorklinischen Studien das Einführen einer Biopsienadel mittels CT-Flouroskopie aufgenommen* ($512^2 \cdot 20$ *Voxel).*

Kapitel 7

Zusammenfassung

Diese Arbeit stellt erstmals die Integration eines Multiskalenmodells für volumetrische Daten in eine hardwarebasierte Visualisierungspipeline vor. Die präsentierten Verfahren erlauben Datensätze auf hochoptimierter dedizierter Hardware zu visualisieren, die um eine Größenordnung umfangreicher sind als bisher. Hierfür wurden exemplarisch an der hardwarebeschleunigten Visualisierungspipeline der VIZARD II-Karte alle nötigen Änderungen und neuen Komponenten vorgestellt, die erforderlich sind um ein waveletbasiertes Multiskalenmodell zu verwenden. Die zugehörige Waveletdekompression dieses hierarchischen Modells wurde ebenfalls in Hardware implementiert. Hierdurch konnte eine Steigerung der Bandbreite zwischen Host-Computer und Visualisierungshardware auf das Dreifache bestehender optimierter sofwarebasierter Dekompression erzielt werden.

Um das zur Bildgebung benötigte, sich dynamisch ändernde, Working Set im lokalen Volumenspeicher der Beschleunigerkarte verwalten zu können, wurde eine Virtualisierung der Speicherschnittstelle der Visualisierungspipeline erforderlich. Daher wurde eine neuartige cachebasierte Speicherschnittstelle, der VoxelCache, entworfen und in VHDL implementiert. Als weiteres Ergebnis wurde dadurch eine effektive Entkopplung der Visualisierungspipeline von den Details der verwendeten Speichertechnologie erzielt. Die Trennung der Pipeline vom verwendeten Volumenspeicher durch den VoxelCache erlaubt eine kontinuierliche Anpassung an neue Hardware. Somit können handelsübliche FPGA-Karten verwendet und zeitaufwendige Eigenentwicklungen, wie sie die hochoptimierte Speicherschnittstelle des VIZARD II erforderlich machte, vermieden werden. Die notwendigen Änderungen für die Verwendung einer neuen Speicherart sind auf den Bereich der Speicherschnittstelle beschränkt und erfordern nun keine Modifikationen an den Komponenten der Visualisierungspipeline mehr.

Die Auswirkung der neuen Speicherschnittstelle auf die Visualisierungspipeline, sowie die benötigten Ressourcen wurden anhand einer vollständigen VHDL-Implementierung des VoxelCaches ermittelt. In einer zyklenakkuraten Simulation wurde für mehrere Speicherarten die Auswirkungen auf die Leistungsfähigkeit der Visualisierungspipeline gemessen und den Ergebnissen einer früher erstellten C++-Simulation gegenübergestellt. Die erhaltenen Ergebnisse zeigen, dass der VoxelCa-

che die ihn gestellten Erwartungen, besonders hinsichtlich dem hochqualitativen Visualisieren mit hohen Abtastraten, erfüllt. In diesem Bereich kann ein von der Visualisierungspipeline benötigtes Voxel zu über 99 % im on-chip Cache vorgefunden werden und benötigt keinen Datentransfer vom lokalen on-board Speicher der Visualisierungskarte. Trotz der zusätzlichen Indirektion durch den VoxelCache wird die Visualisierungspipeline noch mit bis zu 80 % ausgelastet. Nicht bestätigt haben sich die Erwartungen hinsichtlich einer zusätzlichen Optimierung durch ein spekulatives Füllen des VoxelCache, die in der vorangegangenen C++-Simulation gute Performanzgewinne erzielte. Die zusätzliche Latenz des VoxelCache führte zu einer effektiven Leistungseinbuße durch dieses Prefetching-Schema.

Die notwendige Leistung der Waveletttransformation- und der Dekompressionseinheit wurde in einer C++-Simulation der Multiskalenpipeline mittels einer Kamerafahrt für zwei sehr große Datensätze (8,7 GB und 500 MB) abgeschätzt und die erforderliche Bandbreite an transferierten Waveletdaten pro Bild ermittelt. Um eine sehr gute Bildqualität zu garantieren und eine konservative Abschätzung zu erhalten wurde zur Bildgenerierung als Fehlermaß für die Auswahl der Hierarchiestufen des Multiskalenmodells eine maximale projizierte Fläche von 1 Pixel pro Voxel zugrunde gelegt. Da die Simulationsergebnisse stark von den vorgegebenen Rahmenbedingungen abhängen, wurde exemplarisch eine FPGA-Karte mit 512 MB DDRAM-Speicher und 1 Mib SRAM und einem Virtex-II 4000 als Zielplattform benutzt.

Um alle Komponenten der Waveletdekompression in Hardware durchzuführen, wurde die Rücktransformation der Waveletkoeffizienten ebenfalls in VHDL implementiert. Als Multiskalenmodell wurde gezielt ein Waveletschema gewählt, welches geringen arithmetischen Aufwand und ein regelmässiges Speicherzugriffsschema besitzt, sich in einer Pipelinestruktur umsetzen lässt und sich somit sehr gut für eine Implementierung in Hardware eignet. Die erforderlichen Ressourcen sind gering und bestehen hauptsächlich aus on-chip RAM als Puffer zwischen den einzelnen Transformationsmodulen. Mit einer Instanz dieser Einheit kann ein Decodierdurchsatz von bis zu 300 MB/s erreicht werden. Eine parallele Instantiierung der Module ist leicht möglich und nur von den verfügbaren Ressourcen abhängig.

Die hohen Bandbreitenanforderungen an die Beschleunigerkarte wurden reduziert, indem ein Fixed-Huffmankompressionsverfahren verwendet wurde, das sich explizit für die Kompression von Waveletkoeffizienten eignet. Dieses Verfahren wurde in VHDL implementiert und in einer zyklenakkuraten Simulation die Performanz in Hinblick auf die, bei der Evaluierung der Visualisierungspipeline benutzten, Datensätze ermittelt. Die Dekompressionsleistung ist stark vom Kompressionsgrad der Daten abhängig, beläuft sich aber pro Einheit in einer Bandbreite von minimal 10 MB/s und maximal 340 MB/s.

Neben der Handhabung sehr großer Datensätze wurde noch die Unterstützung von dynamischen Datensätzen im Bereich der Echtzeitvisualisierung untersucht. Hierbei lag der Schwerpunkt vor allem auf der Reduzierung der Bandbreite zwischen Volumendatenserver und Visualisierungsclient, ohne zusätzliche größere Latenz in die Datenübertragung einzuführen. Dies ist besonders im Bereich der medizinischen Online-Visualisierung ein wichtiges Kriterium. Der entwickelte 3D-Huffmancode erweist sich dabei als gut geeignet, da er gute Kompressionsraten erreicht und durch

den minimalen arithmetischen Aufwand sowie die Nutzung zeitlicher und räumlicher Kohärenz zwischen den einzelnen Zeitschritten nur geringe Latenz verursacht. Dies wurde anhand zweier Datensätze aus dem Bereich der wissenschaftlichen Simulation und der Medizin bestätigt.

Die hier vorgestellten Verfahren zur Speicherung und Kompression von Volumendaten erweitern die bisherigen Möglichkeiten der Volumenvisualisierungshardware konsequent. Die Visualisierung von sehr großen Datensätzen sowie die Verwendung aktuellster Technologien für die Beschleunigerkarte ist möglich, ohne aufwendige Änderungen an den bestehenden Komponenten vornehmen zu müssen. Somit bietet sich die Möglichkeit diese Visualisierungshardware als zukunftsfähige Basis für weitere Entwicklungen zu nutzen.

Eigene Veröffentlichungen

[1] Kanus, U., G. Wetekam und J. Hirche: *VoxelCache: A Cache-Based Memory Architecture for Volume Graphics*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 76–83, San Diego, California, USA, Juli 2003.

[2] Kanus, U., G. Wetekam, J. Hirche und M. Meissner: *VIZARDII: An FPGA-based Interactive Volume Rendering System*. In: *Field-Programmable Logic and Applications*, Proc. of the 12th International Conference on Field-Programmable Logic, Seiten 1114–1117, Montpellier, France, September 2002.

[3] Meissner, M., U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Strasser, M. Doggett, P. Forthmann und R. Proksa: *VIZARD II: A Reconfigurable Interactive Volume Rendering System*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 137–146, Saarbrücken, Deutschland, September 2002.

[4] Meissner, M., G. Wetekam, J. Hirche und U. Kanus: *Accomodating Pipeline Latency for High Clock Frequency Image Order Volume Rendering Architectures*. Technischer Bericht WSI-2001-13, Wilhelm-Schickard-Institut für Informatik, Graphisch-Interaktive Systeme (WSI/GRIS), Universität Tübingen, 2001.

[5] Wetekam, G. und B. Lutz: *Hardware-Implementierung einer 3D-Huffman-Decodierung für dynamische Volumendaten*. In: *Hardware for Visual Computing Workshop*, Tübingen, Deutschland, April 2005. Erhältlich von http://hvc.gris.uni-tuebingen.de.

[6] Wetekam, G. und Á. del Rio: *Visualization and User Interface for Multi-Slice CT Flouroscopy*. In: *Tagungsband Fachtagung Biomedizinische Technik*, 38. DGBMT Jahrestagung, Seiten 214–215, Ilmenau, Deutschland, September 2004.

[7] Wetekam, G., D. Staneker, U. Kanus und M. Wand: *A Hardware Architecture for Multi-Resolution Volume Rendering*. Akzeptiert bei: Eurographics/SIGGRAPH Workshop on Graphics Hardware, 2005.

Literaturverzeichnis

[8] Ashenden, P. J.: *The Designer's Guide to VHDL*, Kapitel D, Seiten 629–635. Morgan Kaufmann, 1995.

[9] Avnet, Inc: *Communications/Memory Module, User's Guide*. Erhältlich von http://www.avnet.com, Januar 2005.

[10] Avnet, Inc: *Xilinx® Virtex-II Development Kit, Datasheet*. Erhältlich von http://www.avnet.com, Januar 2005.

[11] Bitter, I. und A. Kaufman: *A Ray-Slice-Sweep Volume Rendering Engine*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Los Angeles, California, USA, August 1997.

[12] Boer, M. de, A. Gröpl, J. Hesser und R. Männer: *Latency and Hazard-free Memory Architecture for Direct Volume Rendering*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 109–119, Poitiers, France, August 1996.

[13] Cabral, B., N. Cam und J. Foran: *Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware*. In: *Proc. of Symposium on Volume Visualization*, Seiten 91–98, Washington, DC, Oktober 1994.

[14] Calderbank, R., I. Daubechies, W. Sweldens und B. Yeo: *Wavelet Transforms that Map Integers to Integers*. Technischer Bericht, Department of Mathematics, Princeton University, 1996.

[15] Chiueh, T., C. Yang, T. He, H. Pfister und A. Kaufman: *Integrated Volume Compression and Visualization*. In: Yagel, Roni und Hans Hagen (Herausgeber): *IEEE Visualization '97*, Seiten 329–336, 1997.

[16] Cohen, A., I. Daubechies und J.C. Feauveau: *Biorthogonal bases of compactly supported wavelets*. In: *Comm. Pure & Appl. Math 45*, Seiten 485–560, 1992.

[17] Cullip, T. J. und U. Neumann: *Accelerating Volume Reconstruction with 3D Texture Mapping Hardware*. Technischer Bericht TR93-027, Department of Computer Science at the University of North Carolina, Chapel Hill, 1993.

[18] Dachille IX, F. und A. Kaufman: *GI-Cube: An Architecture for Volumetric Global Illumination and Rendering*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 119–128, Interlaken, Schweiz, August 2000.

[19] Doggett, M.: *An Array Based Design for Real-Time Volume Rendering*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 93–101, Maastricht, Netherlands, August 1995.

[20] Doggett, M., M. Meissner und U. Kanus: *A Low-Cost memory architecture for PCI-Based Interactive Ray Casting*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 7–14, August 1999.

[21] Engel, K., M. Kraus und T. Ertl: *High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading*. In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Los Angeles, California, USA, August 2001.

[22] Ertl, T., R. Westermann und R. Grosso: *Multiresolution and Hierarchical Methods for the Visualization of Volume Data*. Future Generation Computer Systems, Special issue on scientific visualization, 15(1), Februar 1999. ISSN:0167-739X.

[23] European Commission: *DynCT, Real Time Motion Compensated Reconstruction and Visualisation for Dynamic Computed Tomography*. Information Society Technologies, IST-1999-10515, erhältlich von http://www.cordis.lu/ist/home.html, Dezember 2004.

[24] Fowler, J. und R. Yagel: *Lossless Compression of Volume Data*. In: *The 1994 Symposium on Volume Visualization*, Seiten 43–50, 1994.

[25] Gailly, J. und M. Adler: *gzip*. Erhältlich von http://www.gzip.org, Februar 2005.

[26] Ganzenmüller, S.: *Analyse und Design einer objektorientierten SPH-Bibliothek mit Entwurfsmustern unter dem Aspekt der Parallelisierung*. Diplomarbeit, Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen, Tübingen, Deutschland, 2001.

[27] General Electric Corporation: *Engine Block*. Erhältlich von http://www.volvis.org, 2004.

[28] Gibson, J., T. Berger, T. Lookabaugh und D. Lindbergh R. Baker: *Digital Compression for Multimedia*. Morgan Kaufmann Publishers, San Francisco, California, USA, 1998.

[29] Graps, A.: *An Introduction to Wavelets*. IEEE Computational Sciences and Engineering, 2(2):50–61, 1995.

[30] Guthe, S. und W. Strasser: *Advanced Techniques for High-Quality Multi-Resolution Volume Rendering.* Computers & Graphics, 28(1):51–58, Februar 2004.

[31] Guthe, S., M. Wand, J. Gonser und W. Strasser: *Interactive rendering of large volume data sets.* In: *Proc. of IEEE Visualization*, Boston, Massachusetts, USA, Oct. 2002.

[32] Günther, T., C. Poliwoda, C. Reinhard, J. Hesser, R. Männer, H.-P. Meinzer und H.-J. Baur: *VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicine.* In: *Proc. of the 9th Eurographics Hardware Workshop*, Seiten 103–108, Oslo, Norway, September 1994.

[33] Hirche, J.: *VHDL-Design und Synthese eines Phong-Shaders für die Volumenvisualisierung.* Diplomarbeit, Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen, Tübingen, Deutschland, 1999.

[34] Huffman, D.: *A Method for the Construction of Minimum Redundancy Codes.* In: *Proc. of IRE*, Seiten 1098–1101, 1951.

[35] Ihm, I. und S. Park: *Wavelet-based 3D compression scheme for very large volume data.* In: *Proc. of Graphics Interface*, Seiten 107–116. ACM Press, Juni 1998.

[36] Intel Corporation: *PC SDRAM Specification, Revision 1.7.* Erhältlich von http://www.intel.com, März 2005.

[37] JEDEC Solid State Technology Association: *Double Data Rate (DDR) SDRAM Specification, JESD79D* . Erhältlich von http://www.rambus.com, März 2005.

[38] Kanitsar, A., T. Theussl, L. Mroz, M. Srámek, A. V. Bartrolí, B. Csébfalvi, J. Hladůvka, D. Fleischmann, M. Knapp, R.Wegenkittl, P. Felkel, S. Röttger, S. Guthe, W. Purgathofer und E. Gröller: *Christmas Tree Case Study: Computed Tomography as a Tool for Mastering Real World Objects with Applications in Computer Graphics.* In: *IEEE Visualization*, Seiten 489–492, Boston, Massachusetts, USA, Oktober 2002.

[39] Kniss, J., G. Kindlmann und C. Hansen: *Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets.* In: *Proc. of IEEE Visualization*, Seiten 255–262, San Diego, California, USA, Oktober 2001. IEEE Computer Society Press.

[40] Knittel, G.: *VERVE: Voxel Engine for Real-time Visualization and Examination.* Computer Graphics Forum, 3(12):37–48, 1993.

[41] Knittel, G.: *A PCI-based Volume Rendering Accelerator.* In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 73–82, Maastricht, The Netherlands, August 1995.

[42] Knittel, G.: *The UltraVis System.* In: *Proc. of IEEE Symposium on Volume Visualization*, Salt Lake City, Utah, USA, Oktober 2000.

[43] Knittel, G. und W. Strasser: *An Accelerator for Voxel Graphics.* In: *Proc. of the 2. ITG/GI Workshop on Workstations*, Seiten 69–78, Hagen, Deutschland, Mai 1993.

[44] Knittel, G. und W. Strasser: *A Compact Volume Rendering Accelerator.* In: *Proc. of ACM/IEEE Symposium on Volume Visualization*, Seiten 67–74, Washington, D.C., USA, Oktober 1994.

[45] Knittel, G. und W. Strasser: *VIZARD - Visualization Accelerator for Realtime Displays.* In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 139–146, Los Angeles, USA, August 1997.

[46] Kreeger, K. und A. Kaufman: *PAVLOV: A Programmable Architecture for Volume Processing.* In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 77–86, Lisboa, Portugal, August 1998.

[47] Lacroute, P.: *Real-Time Volume Rendering on Shared Memory Multiprocessors Using the Shear-Warp Factorization.* In: *Proceedings IEEE/ACM Parallel Rendering Symposium*, Seiten 15–22, 1995.

[48] Lacroute, P. und M. Levoy: *Fast Volume Rendering Using a Shear-Warp factorization of the Viewing Transform.* In: *Computer Graphics*, Proc. of ACM SIGGRAPH, Seiten 451–457, Juli 1994.

[49] Levoy, M.: *Display of Surfaces From Volume Data.* IEEE Computer Graphics & Applications, 8(5):29–37, Mai 1988.

[50] Levoy, M: *Efficient Ray Tracing of Volume Data.* In: *ACM Transactions on Graphics, 9(3)*, Seiten 245–261, Juli 1990.

[51] Lichtenbelt, B.: *Design of a High Performance Volume Visualization System.* In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Los Angeles, California, USA, August 1997.

[52] Lichtenbelt, B., R. Crane und S. Naqvi: *Introduction to Volume Rendering.* Hewlett-Packard Professional Books, Prentice-Hall, Los Angeles, California, USA, 1998.

[53] Lichtermann, J.: *Design of a Fast Voxel Processor for Parallel Volume Visualization.* In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 83–92, Maastricht, The Netherlands, August 1995.

[54] Li, P. Peggy, Scott Whitman, Roberto Mendoza und James Tsiao: *ParVox - A Parallel Splatting Volume Rendering System for Distributed Visualization*. In: Painter, James, Gordon Stoll und Kwan-Liu Ma (Herausgeber): *IEEE Parallel Rendering Symposium*, Seiten 7–14, Phoenix, Arizona, USA, Oktober 1997.

[55] Lorensen, W. E. und H. E. Cline: *Marching–Cubes: A High Resolution 3D Surface Construction Algorithm*. In: *Computer Graphics*, Proc. of ACM SIGGRAPH, Seiten 163–169, 1987.

[56] Max, N.: *Optical Models for Direct Volume Rendering*. IEEE Transactions on Visualization and Computer Graphics, 1(2):99–108, 1995.

[57] Meissner, M.: *Occlusion Culling and Hardware Accelerated Volume Rendering*. Dissertation, Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen, Tübingen, Deutschland, 2000.

[58] Meissner, M., M. Doggett, J. Hirche und U. Kanus: *Efficient Space Leaping for Ray Casting Architectures*. In: *Volume Graphics*, Workshop on Volume Graphics, Seiten 149–161, Stony Brook, NY, USA, Juni 2001.

[59] Meissner, M., U. Hoffmann und W. Strasser: *Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering using OpenGL and Extensions*. In: *Proc. of IEEE Visualization*, Seiten 207–214, San Franisco, California, USA, Oktober 1999. IEEE Computer Society Press.

[60] Meissner, M., J. Huang, D. Bartz, K. Mueller und R. Crawfis: *A Practical Evaluation of Four Popular Volume Rendering Algorithms*. In: *Proc. of ACM Symposium on Volume Visualization*, 2000.

[61] Mentor Graphics Corporation: *Precision Synthesis 2004a*. Precision Physical Synthesis Datasheet, erhältlich von http://www.mentor.com, 2004.

[62] Neumann, Ulrich: *Parallel volume rendering algorithm performance on mesh connected multicomputers*. In: *Proceedings on the Parallel Rendering Symposium*, Seiten 97–104, San Jose, California, USA, Oktober 1993.

[63] Nieh, J. und M. Levoy: *Volume Rendering on Scalable Shared-Memory MIMD Architectures*. In: *Proceedings of the 1992 Workshop on Volume Visualization*, Seiten 17–24, Boston, Massachusetts,USA, 1992.

[64] Ning, P. und L. Hesselink: *Vector Quantization for Volume Rendering*. In: *Proc. of IEEE Visualization*, Seiten 69–74, Boston, Massachusetts, USA, Oktober 1992.

[65] Nyquist, H.: *Certain topics in telegraph transmission theory*. In: *Transactions of the AIEE*, Oktober 1928.

[66] Pfister, H., J. Hardenbergh, J. Knittel, H. Lauer und L. Seiler: *The VolumePro Real-Time Ray-Casting System.* In: *Computer Graphics*, Proc. of ACM SIGGRAPH, Seiten 251–260, August 1999.

[67] Pfister, H., A. Kaufman und T. Chiueh: *Cube-3: A Real-Time Architecture for High-Resolution Volume Visualization.* In: *Proc. of Workshop on Volume Visualization*, Seiten 75–83, Washington, D.C., USA, Oktober 1994.

[68] Pfister, H. und A. E. Kaufman: *Cube-4 - A Scalable Architecture for Real-Time Volume Rendering.* In: *Proc. of Symposium on Volume Visualization*, Seiten 47–ff., San Francisco, California, USA, 1996.

[69] Philips Research: *Aneurism.* Erhältlich von http://www.volvis.org, 2004.

[70] Philips Research: *Menschlicher Fuß.* Erhältlich von http://www.volvis.org, 2004.

[71] Philips Research: *Thorax.* 2004.

[72] Phong, B. T.: *Illumination for Computer Generated Picture, Vol. 18, No 6.* In: *Communications of the ACM*, Juni 1975.

[73] RamBus, Inc: *800/1066/1200 MHz RDRAM®, Datasheet.* Erhältlich von http://www.rambus.com, Januar 2005.

[74] Ray, H., H. Pfister, D. Silver und T. Cook: *Ray-Casting Architectures for Volume Visualization.* IEEE Transactions on Visualization and Computer Graphics, 5(3):210–223, Juli 1999.

[75] Ray, H. und D. Silver: *A Memory Efficient Architecture for Real-Time Parallel and Perspective Direct Volume Rendering.* Technischer Bericht CAIP-TR-237, Department of Computer Aids for Industrial Productivity, Rutgers University, 1999.

[76] Rezk-Salama, C., K. Engel, M. Bauer, G. Greiner und T. Ertl: *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Texturing and Multi-Stage Rasterization.* In: *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Seiten 109–118, Interlaken, Switzerland, August 2000.

[77] Rodler, F.: *Wavelet Based 3D Compression for Very Large Volume Data Supporting Fast Random Access.* In: *Pacific Graphics*, Seiten 108–117, oct 1999.

[78] Sayood, K.: *Introduction to Data Compression.* Morgan Kaufmann Publishers, San Francisco, California, USA, 2000.

[79] Sedgewick, R.: *Algorithmen in C++.* Addison-Wesley GmbH, Deutschland, 1994.

[80] SFB 382 der Deutschen Forschungsgemeinschaft: *Dieseleinspritzung*. Erhältlich von http://www.volvis.org, 2004.

[81] SIG, PCI: *PCI Local Bus Specification, Revision 2.2*. 3.5.4.1 Bandwidth and Latency Considerations, Dezember 1998.

[82] SIG, PCI: *PCI-X 1.0b Specification*. Erhältlich von http://www.pcisig.com, Juli 2003.

[83] Stollnitz, E. J., T.D. DeRose und T.D. Salesin: *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann, 1996.

[84] Strengert, Magnus, Marcelo Magallon, Daniel Weiskopf, Stefan Guthe und Thomas Ertl: *Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters*. Parallel Graphics and Visualization, 2004.

[85] SUNY Stony Brook: *Neghip*. Erhältlich von http://www.volvis.org, 2004.

[86] Sweldens, W. und P. Schröder: *Building your own wavelets at home*. In: *Wavelets in Computer Graphics*, SIGGRAPH Course Notes, 1996.

[87] TeraRecon, Inc: *VolumePro® 1000, Real-Time 3D Volume Rendering Single Board Special Purpose Computer*. Datasheet, erhältlich von http://www.terarecon.com, Dezember 2004.

[88] U.S. National Library of Medicine: *The Visible Human Project®*. Erhältlich von http://www.nlm.nih.gov/research/visible/visible_human.html, Dezember 2004.

[89] Vettermann, B. und J. Hesser: *Solving the Hazard Problem for Algorithmically Optimized Real-Time Volume Rendering*. In: *Proc. of 1st Workshop on Volume Graphics*, März 1999.

[90] Westermann, R.: *A Multiresolution Framework for Volume Rendering*. In: *Symposium on Volume Visualization*, Proc. of the 1994 symposium on Volume visualization, Seiten 51–58, Tysons Corner, Virginia, United States, Oktober 1994. ISBN:0-89791-741-3.

[91] Westover, L.: *Footprint Evaluation for Volume Rendering*. In: *Computer Graphics*, Proc. of ACM SIGGRAPH, Seiten 367–376, August 1990.

[92] Xilinx Inc.: *Virtex-II Platform FPGAs: Complete Data Sheet (v3.3)*. Product Specification, erhältlich von http://www.xilinx.com, Juni 2004.

[93] Yeo, B-L. und B. Liu: *Volume Rendering of DCT-Based Compressed 3D Scalar Data*. In: *IEEE Transactions on Visualization and Computer Graphics*, Band 1, Seiten 29–43, 1995.

[94] Ziv, J. und A. Lempel: *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory, 23(3):337–343, 1977.

Lebenslauf

20. März 1974	Geboren in Iserlohn
1980 – 1984	Grundschule Bildungszentrum Sankt Konrad, Ravensburg
1984 – 1993	Katholisches Freies Gymnasium Sankt Konrad, Ravensburg Abschluß: Abitur
1994 – 2000	Studium der Physik mit Nebenfach Informatik an der Eberhard-Karls-Universität Tübingen Abschluß: Diplom Physiker
1998 – 1999	Studienaustausch an der California State University Long Beach, Kalifornien, USA
seit 2000	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Graphisch-Interaktive Systeme am Wilhelm-Schickard-Institut für Informatik der Eberhard-Karls-Universität Tübingen (Prof. Straßer)