Implementierung und Untersuchung paralleler Volumen-Rendering-Techniken für Graphikprozesoren

Tim Werner

Bayreuth Reports on Parallel and Distributed Systems

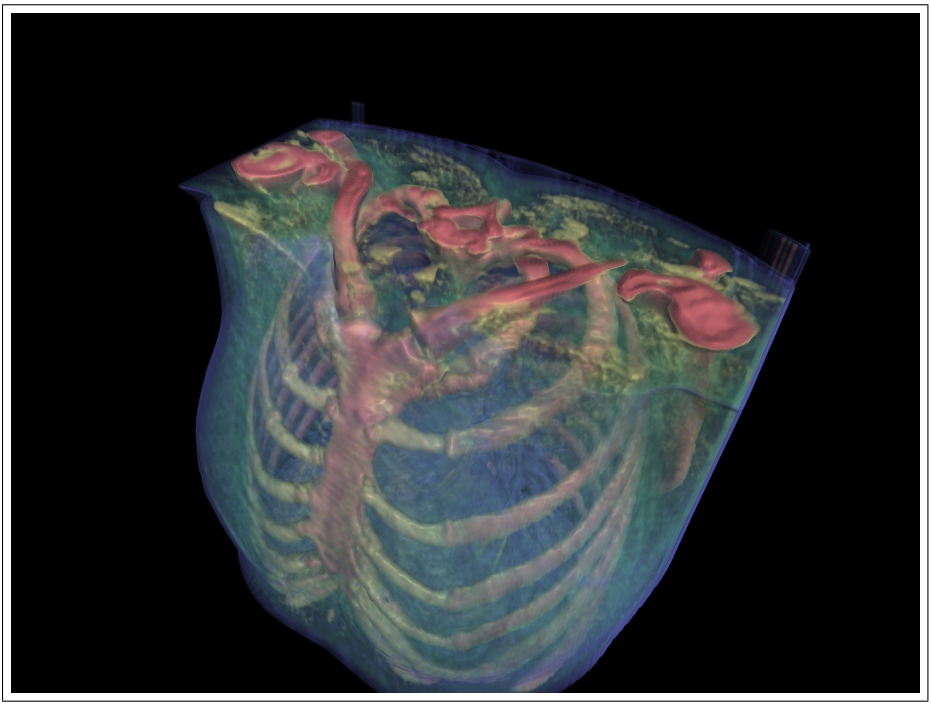
No. 5, January 2013

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Implementierung und Untersuchung paralleler Volumen-Rendering-Techniken für Graphikprozessoren

Bachelorarbeit



Tim Werner

14. Januar 2013

Angewandte Informatik 2 — Parallele und Verteilte Systeme

Universität Bayreuth
95440 Bayreuth

Zusammenfassung

Deutsche Zusammenfassung

Zuerst soll in dieser Arbeit ein Standardverfahren für einen Volumenraycaster auf einer GPU in OpenCL implementiert und erläutert werden. Hierbei werden kurz die benötigten Algorithmen aus der Computergraphik sowie deren mathematischen und physikalischen Herleitungen erklärt. Dabei werden einige einfache Optimierungen vorgestellt, welche spezifisch auf die Hardware der GPU eingehen. Auch werden einfache Verbesserungen erläutert, welche rein optischer Natur sind. Durch deren Implementierung lassen sich die dreidimensionalen Strukturen der Volumendaten besser erkennen. Zusätzlich ist es mit diesen Verbesserungen möglich die interessanten Bereiche der Volumendaten hervorzuheben. Des Weiteren läuft dieser implementierte Standard-Volumenraycaster auf moderner Hardware bereits mit Bildwiederholraten, welche Interaktivität ermöglichen.

Danach soll dieser Standard-Volumenraycaster durch Empty-Space-Skipping, also durch das Überspringen von leeren Bereichen innerhalb der Volumendaten, mit Hilfe eines Octrees beschleunigt werden. Hierbei ist es das Ziel den Raycaster mit dem Octree auf die spezifischen Gegebenheiten der GPU anzupassen. Insbesondere soll versucht werden die SIMD-Effizienz und damit die Performance durch eine geschickte Kombination von der Octreetraversierung und dem Zeichnen des Inhalts eines Octreeknotens zu erhöhen. Durch diese Octreeoptimierung lässt sich die Performance des Raycasters bei Volumendaten mit vielen leeren Bereichen stark erhöhen.

Als Nächstes wird in dieser Arbeit versucht, den Raycaster weiter mit persistenten Threads zu beschleunigen. Dieser Versuch scheitert jedoch. So sind sowohl die Performance als auch die SIMD-Effizienz mit persistenten Threads stets geringer als ohne persistente Threads.

Abschliessend soll die Ausführung des Raycasters mit Octree auf der GPU näher untersucht werden. Zuerst werden hierfür Messungen durch selbst erstellte Benchmarks vorgenommen. Bei der darauffolgenden Diskussion der Messergebnisse zeigt es sich, dass das Verfahren linear mit dem Chiptakt und der Zahl der Multiprozessoren skaliert. Da es nur einen kleinen Bruchteil der maximalen Speicherbandbreite benötigt, skaliert es nicht mit dem Speichertakt. Die Speicherzugriffe dieses Verfahrens sind bereits so lokal, dass die GPU-Caches gut ausgenutzt werden. Deshalb können die Speicherzugriffszeiten so gut verborgen werden, dass sie sich nicht negativ auf die Performance auswirken. Die Rechenleistung der GPU lässt sich durch das Verfahren jedoch nur mittelmässig bis schlecht ausnutzen. Diese niedrige Ausnutzung ist vor allem auf eine zu niedrige Occupancy und auf eine zu ungleichmässige Auslastung der verschiedenen Ressourcen der GPU zurückzuführen. Jedoch ergeben sich aus den Untersuchungen noch einige weitere Optimierungsansätze, um die erzielte Rechenleistung und damit die Performance zu erhöhen.

Obwohl kurz graphische Optimierungen in der Arbeit vorgestellt und implementiert werden, so ist es kein Ziel dieser Arbeit die Bildqualität des Raycasters massiv zu verbessern oder diesen durch verbesserte Näherungslösungen oder Ähnliches zu beschleunigen. Bei vielen dieser Verbesserungen wäre es nur schlecht möglich gewesen spezifisch auf die Eigenschaften der GPU einzugehen. Auch beschäftigt sich diese

Arbeit hauptsächlich mit der Verbesserung der SIMD-Effizienz und im Gegensatz zu vielen anderen GPGPU-Arbeiten nur wenig mit der Optimierung der Speicherzugriffe.

Englische Zusammenfassung

This thesis starts with an explanation and an OpenCL-implementation of a standard volumetric ray caster. Therefore the necessary algorithms from computer graphics and their mathematical and physical derivations are explained. Furthermore some simple optimizations are presented, which particularly make use of the GPU hardware. Also some simple improvements to the image quality are explained and implemented. Those improvements make the three dimensional structures of the volumetric data better visible. Moreover it is possible by those improvements to highlight the areas of interest of the volumetric data. Furthermore this ray caster runs already at high frame rates, which allow interactivity.

Then the performance of this volumetric ray caster shall be improved by empty space skipping. For this empty space skipping an octree is used. Especially it was a goal to improve the performance by adapting the octree ray caster to the GPU hardware. In particular it is tried to increase the SIMD-efficiency and thus the performance by an adept combination of the octree traversal and the rendering of the contents of the octreenodes. By those optimizations the performance of rendering volumetric data with many empty spaces is greatly increased.

Next it will be tried in this thesis to increase the performance of the ray caster even further by implementing persistent threads. However this attempt failed. So the performance and the SIMD-efficiency both are decreased by those persistent threads.

Finally the execution of the improved octree ray caster on the GPU shall be examined. Therefore some measurements by self written benchmarks are taken. By the discussion of the measurement results it turns out, that the octree ray caster scales linearly with the GPU clock and the amount of the GPU's multiprocessors. Since this algorithm just uses a small fraction of the maximum memory bandwidth, it doesn't scale with the memory clock. The memory accesses of this octree ray caster are already very local. Thus the caches of the GPU are used efficiently. That way the memory latencies can be very well hidden, so that they don't decrease the performance. However the octree ray caster just harnesses a small to medium amount of the maximum processing power of the GPU. This is mostly caused by a too low occupancy and an uneven utilization of the different resources of the GPU. However during the discussion some optimization approaches are suggested, which might increase the amount of harnessed processing power.

Although some simple graphical optimizations are introduced and implemented, it wasn't a goal of this thesis to greatly increase image quality of the ray caster or to improve its performance by improved approximation methods. Because with many of those improvements it wouldn't have been possible to adopt those especially to the GPU's architecture. Also this thesis mainly tries to increase the ray caster's SIMD-efficiency. However this thesis doesn't try in contrast to many other GPGPU-works to optimize the memory accesses of the ray caster.

INHALTSVERZEICHNIS

1 Einleitung **5**
1.1 Motivation . . . 5
1.2 Zielstellung und Gliederung . . . 7

2 OpenCL- und GPGPU-Grundlagen **8**
2.1 OpenCL-Grundlagen . . . 8
2.2 OpenCL-Implementierung auf einer GPU und GPGPU-Grundlagen . . 11
2.2.1 Allgemeines . . . 11
2.2.2 Ausführeinheiten einer GPU . . . 12
2.2.3 Ausführungsmodell und Ausführungsverhalten einer GPU . . . 15
2.2.4 Auswirkung von Sprungbefehlen auf die Performance . . . 17
2.2.5 Speichermodell einer GPU . . . 18
2.2.6 Occupancy . . . 19
2.3 Limitierungen von OpenCL bei GPGPU . . . 20

3 Standard-Volumenraycaster **22**
3.1 Einleitende Bemerkungen . . . 22
3.2 Volumendaten . . . 23
3.3 Allgemeines . . . 24
3.4 Anmerkungen . . . 26
3.5 Vorbereitende Schritte . . . 26
3.5.1 Erstellen einer Farbtextur und transformierten Opazitätstextur durch die Transferfunktion . . . 26
3.5.2 Erstellen einer dreidimensionalen Normalentextur aus den Volumendaten . . . 29
3.6 Raycasting . . . 33
3.6.1 Threaderstellung . . . 33
3.6.2 Strahlerstellung . . . 39
3.6.3 Schnittpunktberechnung mit der Boundingbox . . . 41
3.6.4 Transformation des Strahls in Normalisierte Texturkoordinaten 44
3.6.5 Raymarching . . . 46
3.6.5.1 Allgemeines . . . 46
3.6.5.2 Schattierung durch das Phong-Beleuchtungsmodell . . 49
3.6.5.3 A-Compositing . . . 53
3.6.6 Erweiterung des Standardverfahrens um eine Deep-Shadow-Map 56
3.6.7 Modifikation des Standardverfahrens auf eine Post-Classified Transferfunktion . . . 57

4 Empty-Space-Skipping mit Hilfe eines Octrees **59**
4.1 Aufbau und Konstruktion des Octrees . . . 59
4.2 Traversierung . . . 61
4.3 Kombination der Traversierung und des Raymarchings . . . 65
4.3.1 Allgemeines . . . 65
4.3.2 First-Approach-Kombination . . . 68
4.3.3 For-For-Kombination . . . 69

4.3.4 Buffered-For-For-Kombination . . . 75
4.3.5 Buffered-For-For-With-Flags . . . 79

5 Implementierung von persistenten Threads 83

6 Untersuchungen 88
6.1 Allgemeines . . . 88
6.2 Messungen . . . 89
6.2.1 Laufzeitabhängigkeit vom Chiptakt . . . 89
6.2.2 Laufzeitabhängigkeit vom Speichertakt . . . 89
6.2.3 Laufzeitabhängigkeit von der Multiprozessorzahl . . . 90
6.2.4 Laufzeitabhängigkeit von der Workgroupgrösse . . . 91
6.2.5 Laufzeitabhängigkeit von der Occupancy . . . 93
6.2.6 Visual-Profiler-Messungen . . . 93
6.2.7 Messungen der Samplerate . . . 97
6.2.8 Messung der SIMD-Effizienzen . . . 97
6.2.9 Messung der Raymarchingschritte und Octreetraversierungsschritte . . . 98
6.3 Diskussion der Messergebnisse . . . 98
6.3.1 Allgemeines . . . 98
6.3.2 Effizienz der Caches . . . 100
6.3.3 Auslastung und Ausnutzung der Graphikkarte . . . 102
6.3.3.1 Allgemeines . . . 102
6.3.3.2 Auslastung der Speicherbandbreite . . . 103
6.3.3.3 Auslastung der Warp-Scheduler . . . 104
6.3.3.4 Auslastung und Ausnutzung der CUDA-Cores . . . 105
6.3.3.5 Auslastung der SFUs . . . 106
6.3.3.6 Auslastung der LSUs . . . 106
6.3.3.7 Auslastung und Ausnutzung der TMUs . . . 107
6.3.4 Skalierbarkeit . . . 110
6.3.5 Einfluss der Workgroupgrösse auf die Performance . . . 117
6.3.6 Einfluss der Occupancy auf die Performance . . . 117
6.4 Limitierung der Performance . . . 121
6.5 Fazit der Untersuchungen . . . 123

7 Schluss 125

8 Anhang 126
8.1 Probleme mit OpenCL . . . 126
8.2 Praktische Arbeit . . . 127
8.3 Quellenangaben . . . 127
8.4 Erklärung . . . 130

ABBILDUNGSVERZEICHNIS

1.1 Beispiele für das Volumenrendering aus den verschiedenen Anwendungsbereichen . . . 6

2.1 OpenCL-Logo . . . 8
2.2 OpenCL-Host mit mehreren Devices . . . 8
2.3 Globale OpenCL-2D-Range . . . 9
2.4 OpenCL-Vektoraddtionskernel . . . 10
2.5 Ausführung der OpenCL-Vektoraddition . . . 10
2.6 OpenCL-Speichermodell . . . 11
2.7 Blockdiagramm einer Nvidia 580 GTX . . . 13
2.8 Blockdiagramm eines Fermi-Multiprozessors einer Nvidia 580 GTX . . . 14
2.9 Warpaufteilung in einer Workgroup . . . 15

3.1 Boundingbox der Volumendaten . . . 24
3.2 Übersicht der Vorgehensweise beim Standard-Volumenraycaster . . . 25
3.3 Anwendung der Pre-Classified Transferfunktion auf eine CT-Aufnahme eines weiblichen Brustkorbs . . . 27
3.4 Unterschiedliche Vorgehensweisen bei der Pre-Classified und bei der Post-Classified Transferfunktion . . . 28
3.5 Vergleich zwischen der Bildqualität von Post-Classified und Pre-Classified Transferfunktionen . . . 30
3.6 Visualisierung der Gradienten und der daraus resultierenden Normalen eines elliptischen Opazitätsverlaufs . . . 31
3.7 Filtermasken der Gradientenfilter . . . 32
3.8 Visualisierung der Normalentexturen von Central-Difference und Sobel-Filter . . . 33
3.9 Vergleich der Bildqualität zwischen Sobel-Filter und Central-Difference-Filter . . . 34
3.10 Kachelung des zu rendernden Bildes durch Workgroups . . . 34
3.11 Umsortieren der Workitems in der Workgroup . . . 36
3.12 Warpaufteilung innerhalb einer Workgroup vor und nach dem Umsortieren 37
3.13 Benötigte Geometrie für die Strahlerstellung . . . 39
3.14 Schnittpunkte der Strahlengerade mit der Boundingbox . . . 42
3.15 Beispiel für die Schnittpunktberechnung mit der Boundingbox . . . 43
3.16 Illustration des normalisierten Volumenkoordinatensystems und des normalisierten Texturkoordinatensystems . . . 45
3.17 Visualisierung des Raymarchings . . . 47
3.18 Pseudocode des Raymarchings . . . 48
3.19 Unterpunkte beim Raymarching . . . 48
3.20 Vergleich der Bildqualität zwischen Nearest-Neighbor-Interpolation und trilinearer Interpolation . . . 49
3.21 Die verschiedenen Intensitäten des Phong-Beleuchtungsmodells . . . 51
3.22 Für die Phong-Beleuchtungsmodell benötigte Geometrie . . . 52
3.23 A-Compositing mehrerer transparenter Rechtecke . . . 53
3.24 Simulierter Intensitätsverlauf beim Raymarching. . . . 55
3.25 Pseudocode des A-Compositings . . . 56

3.26 Schattenwurf durch eine Deep-Shadow-Map 57
3.27 Pseudocode des Raymarchings für die Post-Classified Transferfunktion . . 58

4.1 Aufbau und Konstruktion des verwendeten Octrees 60
4.2 Pseudocode der Octreetraversierung 62
4.3 Visualisierung des zum Bonsai gehörenden Octrees 66
4.4 Standardansicht des Bonsais 67
4.5 Pseudocode der First-Approach-Kombination 68
4.6 SIMD-Effizienzen der First-Approach-Kombination 68
4.7 Visualisierung der Raymarching SIMD-Effizienz der First-Approach-Kombination durch einen bestimmte Knoten 70
4.8 Visualisierung der SIMD-Effizienzen der First-Approach-Kombination . . 71
4.9 Pseudocode von der If-If-Kombination für einen Raytracer 71
4.10 Pseudocode von der While-While-Kombination für einen Raytracer 72
4.11 Pseudocode von der If-If-Kombination für einen Volumenraycaster 72
4.12 Pseudocode von der While-While-Kombination für einen Volumenraycaster 73
4.13 Pseudocode von der For-For-Kombination 73
4.14 SIMD-Effizienzen der First Approach-, If-If-, While-While- und For-For-Kombinationen 74
4.15 Visualisierung der SIMD-Effizienzen beim Traversieren bei der If-If-, While-While- und For-For-Kombination 74
4.16 Visualisierung der SIMD-Effizienzen beim Raymarching bei der If-If-, While-While- und For-For-Kombination 75
4.17 Pseudocode der Buffered-For-For-Kombination 77
4.18 SIMD-Effizienzen der For-For-Kombinationen und der Buffered-For-For-Kombination 78
4.19 Visualisierung der SIMD-Effizienzen der Buffered-For-For-Kombination . 78
4.20 Pseudocode der Buffered-For-For-With-Flags-Kombination 79
4.21 SIMD-Effizienzen der Buffered-For-For-Kombination und der Buffered-For-For-With-Flags-Kombination 81
4.22 Visualisierung der SIMD-Effizienzen der Buffered-For-For-With-Flags-Kombination 82

5.1 Pseudocode der persistenten Threads-Verfahren 84
5.2 SIMD-Effizienzen und FPS der Buffered-For-For-With-Flags Kombination und der persistenten Threads 86
5.3 Visualisierung der SIMD-Effizienzen beim Raymarching bei den persistenten Threads 86
5.4 Visualisierung der SIMD-Effizienzen bei der Octreetraversierung bei den persistenten Threads 87

6.1 Standardansichten vom Bonsai, vom Sparschwein und vom Teddy 89
6.2 Performancemessung in Abhängigkeit von dem Chiptakt der Graphikkarte 90
6.3 Performancemessung in Abhängigkeit von dem Speichertakt der Graphikkarte 90
6.4 Performancemessung in Abhängigkeit von der Zahl der Multiprozessoren . 92
6.5 Performancemessung in Abhängigkeit der Workgroupgrösse 92
6.6 Performancemessung in Abhängigkeit von der Occupancy 93

6.7 Visual-Profiler-Messungen 95
6.8 Messung der Samplerate beim Raycaster 97
6.9 Messung der Samplerate in einem TMU-limitierten Benchmark 97
6.10 Messung der SIMD-Effizienzen 98
6.11 Messung der Traversierungsschritte und der Raymarchingschritte 98
6.12 Traversierungsschritte und Raymarchingschritte 98
6.13 Verhältnis von komplett durchsichtigen Raymarchingschritten zu allen Raymarchingschritten 100
6.14 Auslastung der Speicherbandbreite 103
6.15 Auslastung der Warpscheduler 104
6.16 Auslastung und Ausnutzung der CUDA Cores 106
6.17 Auslastung der TMUs 107
6.18 Ausnutzung der TMUs 110
6.19 Speedup in Abhängigkeit von der Zahl der Multiprozessoren 111
6.20 Skalierungseffizienz in Abhängigkeit von der Zahl der Multiprozessoren . . 111
6.21 Diagramm des Speedups in Abhängigkeit von der Zahl der Multiprozessoren 112
6.22 Diagramm der Skalierungseffizienz in Abhängigkeit von der Zahl der Multiprozessoren 113
6.23 Speedup in Abhängigkeit vom normierten Chiptakt 114
6.24 Skalierungseffizienz in Abhängigkeit vom Chiptakt 114
6.25 Diagramm des Speedups in Abhängigkeit des Chiptakts 115
6.26 Diagramm der Skalierungseffizienz in Abhängigkeit des Chiptakts 116
6.27 Normierte Laufzeit in Abhängigkeit von der Workgroupgrösse 118
6.28 Speedup in Abhängigkeit von der Occupancy 119
6.29 Diagramm des Speedups in Abhängigkeit von der Occupancy 120
6.30 Raymarchingmessungen durch eine volle und eine leere Boundingbox . . . 122
6.31 Octreetraversierungsmessung 122
6.32 Ausnutzung der Rechenleistung insgesamt und Vergleich mit der maximalen Rechenleistung einer CPU 124

1 Einleitung

1.1 Motivation

Volumenrendering ist ein Oberbegriff für eine Vielzahl von Verfahren, welche einen dreidimensionalen volumetrischen Datensatz oder eine Funktion in ein zweidimensionales Bild transformieren. Viele dieser zu visualisierenden Datensätze stammen aus Scans von realen Objekten durch MRT (Magnetresonanztomographie) oder CT (Computertomographie), meist aus dem Medizin oder dem Ingenieursbereich. Auch können diese Datensätze synthetischen Ursprungs oder das Ergebnis von Computersimulationen sein. Beispielhaft hierfür wären physikalische Simulationen, chemische Simulationen, ingenieurtechnische Simulationen oder Visualisationen oder Wettersimulationen. Die zu rendernden Funktionen können mathematischen Visualisationen beispielsweise von dreidimensionalen Fraktalen dienen. Auch können sie prozeduraler Natur sein, um in Filmen, Bildern oder Spielen diverse visuelle Effekte zu erzeugen. Dementsprechend findet das Volumenrendering in der Medizin, im Ingenieurwesen, im wissenschaftlichen Bereich, im künstlerischen Bereich und im Entertainmentbereich Anwendung.

Die Renderingverfahren selbst teilen sich in indirekte und direkte Volumenrendering-Verfahren auf. Bei indirekten Verfahren wird das Volumenmodell zuerst durch einen Segmentierungsalgorithmus in ein Oberflächenmodell transformiert und danach als solches gezeichnet. Eine einfache Version eines solchen Algorithmus berechnet lediglich eine Isosurface aus dem Volumenmodell. Im Gegensatz dazu zeichnen die direkten Volumenrendering Techniken das Volumen direkt, ohne zuvor eine Oberflächengeometrie zu extrahieren. Dadurch kann man bei den Visualisationen dieser Verfahren meist die inneren Strukturen des Volumendatensatzes erkennen, welche einem bei Oberflächenmodellen verborgen bleiben. Allerdings kosten diese Techniken auch wesentlich mehr Rechenzeit. Mögliche Beispiele für Vertreter der direkten Techniken sind Raytracing, Raycasting, Shear-Warp, Splatting oder Texture-Slicing. Diese Techniken sind in [RTVG06] erklärt. Am häufigsten wird auf GPUs das Raycasting für das Volumenrendering verwendet. Beispielhafte Visualisationen durch dieses Verfahren sind in Abbildung 1.1 dargestellt.

Wegen des hohen Aufwands dieser Verfahren war es lange Zeit nur mit der Hilfe von Rechnerclustern oder von Workstations mit spezialisierter Hardware möglich, diese Verfahren in Echtzeit darzustellen und dadurch Interaktivität zu ermöglichen. Dies änderte sich mit dem Aufkommen von programmierbaren und immer leistungsfähigeren GPUs. Da GPUs im Gegensatz zu den CPUs für Datenparallelverarbeitung optimiert und die meisten Volumenrenderverfahren gut datenparallelisierbare Techniken sind, können die GPUs diese Techniken im Vergleich zu CPUs stark beschleunigen. Deshalb übernehmen die GPUs zunehmend die komplette Aufgabe des Volumenrenderings. So ist es mittlerweile dank diesen möglich, Volumenrendering in Echtzeit auf jedem gewöhnlichen PC durchzuführen. Dennoch bringen diese Volumenrendertechniken je nach gewünschter Bildqualität moderne Graphikkarten an ihre Leistungsgrenzen. Deshalb werden Optimierungen nötig, welche speziell auf die Hardwarearchitektur der Graphikkarte zugeschnitten sind.

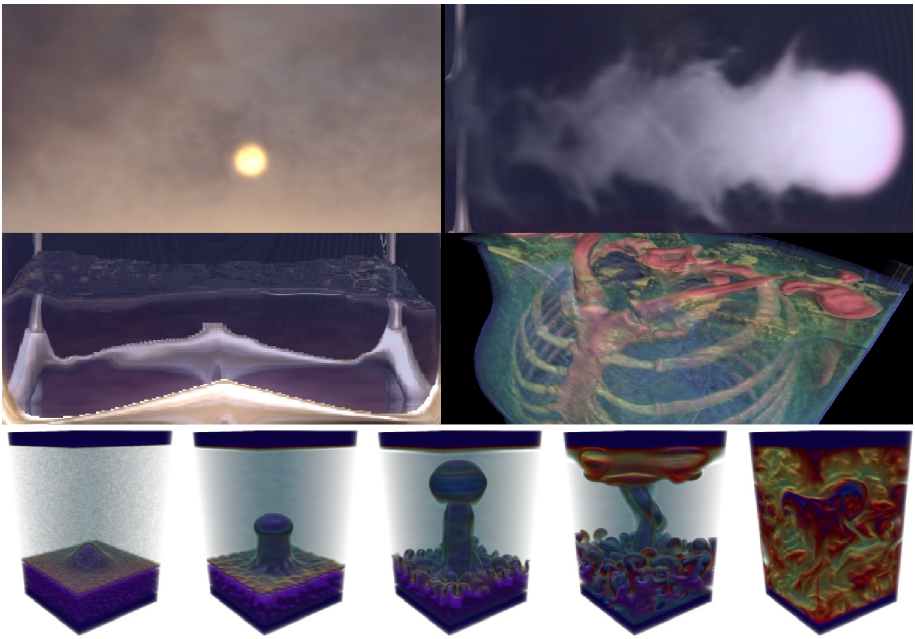


Abbildung 1.1: **Beispiele für das Volumenrendering aus den verschiedenen Anwendungsbereichen.** Oben links: Prozedurale volumetrische Wolken aus der Unigine-Heaven-Demo; oben rechts: Volumetrischer und physikalisch simulierter Rauch in der Nvidia Box-of-Smoke-Demo; Mitte links: Volumetrisches und physikalisch simuliertes Wasser in der Nvidia Box-of-Smoke-Demo, Mitte rechts: Volumetrisches Rendering eines CT-Scans durch den praktischen Teil der Arbeit, unten: Volumetrisches Rendering einer Thermal-Plume aus einer physikalischen Simulation aus [JM10]

1.2 Zielstellung und Gliederung

Das erste Ziel dieser Arbeit ist es ein Standardverfahren für einen Volumenraycaster auf einer GPU in OpenCL zu implementieren und zu erläutern. Des Weiteren soll versucht werden durch Optimierungen die Performance dieses Raycasters zu verbessern. Hierfür soll probiert werden den Raycaster durch Empty-Space-Skipping mit Hilfe eines Octrees zu beschleunigen. Insbesondere ist es das Ziel, die SIMD-Effizienz beim Octree-Raycaster zu erhöhen. Als Nächstes soll versucht werden die Performance weiter mit Hilfe von persistenten Threads zu erhöhen. Abschliessend soll die Ausführung des Volumenraycasters auf der GPU untersucht werden. So ist die Arbeit in folgende Kapitel gegliedert:

- In Kapitel 2 werden zuerst die für die Arbeit benötigten Grundlagen der OpenCL-API erklärt. Danach wird beschrieben, wie die OpenCL-API auf einer Graphikkarte implementiert ist. Anschliessend wird kurz erläutert, wodurch man bei der GPGPU-Programmierung durch die OpenCL-API limitiert wird.
- In Kapitel 3 wird das implementierte Standardverfahren eines Volumenraycasters vorgestellt. Unter anderem werden die benötigten Algorithmen aus der Computergraphik und ihre mathematischen bzw. physikalischen Herleitungen beschrieben. Zusätzlich werden einfache GPU-spezifische Optimierungen vorgestellt.
- In Kapitel 4 wird erläutert wie ein Octree im praktischen Teil der Arbeit beim Volumenraycaster herangezogen wird, um die komplett durchsichtigen Bereiche der Volumendaten zu überspringen, um die Performance zu erhöhen. Als Nächstes wird versucht die SIMD-Effizienz und damit die Performance weiter durch eine geschickte Kombination von Octreetraversierung und dem Zeichnen des Knoteninhalts zu optimieren.
- In Kapitel 5 wird das Verfahren der persistenten Threads vorgestellt. Zusätzlich wird mit diesem Verfahren versucht die Performance des Raycasters weiter zu verbessern.
- In Kapitel 6 wird die Ausführung des Verfahrens auf der GPU näher untersucht. Hierfür werden zuerst Messreihen durch selbst erstellte Benchmarks aufgezeichnet. Anschliessend werden die Messreihen ausgewertet. Bei dieser Auswertung wird insbesonders darauf eingegangen, wie gut die GPU ausgelastet und ausgenutzt wird und wodurch die Performance letztendlich limitiert wird. Auch wird die erzielte Rechenleistung und Skalierbarkeit bestimmt.
- In Kapitel 7 wird kurz der Erfolg der Arbeit bewertet.

2 OpenCL- und GPGPU-Grundlagen

2.1 OpenCL-Grundlagen

Als Erstes werden in diesem Kapitel die für die GPGPU-Programmierung benötigten OpenCL(Open Computing Language)-Grundlagen erklärt.

Abbildung 2.1: OpenCL-Logo

> "It (OpenCL) is more than a language."
> aus The OpenCL Specification [KH11].

Die Beschreibungen hierfür stammen aus [KH11]. Jedoch wird diese Quelle hier stark vereinfacht und zusammengefasst wiedergegeben. So ist OpenCL eine Bibliothek für die Programmierung heterogener Systeme. Diese ermöglicht es einem Programm, das auf einer CPU bzw. einem Host läuft, Unterprogramme, sogenannte Kernels, auf der CPU selbst und auf diversen Coprozessoren, Devices genannt, auszuführen. Ein OpenCL-Host mit mehreren Devices ist in Abbildung 2.2 gezeigt. Da die meisten Coprozessoren Parallelrechner sind, ist auch OpenCL für die Parallelverarbeitung ausgelegt. Viele dieser Devices besitzen ihren eigenen Speicher, weshalb OpenCL zusätzlich Funktionen zur Datenübertragung zu diesen Devices und zur Verwaltung des Device-Speichers zu Verfügung stellt.

OpenCL spezifiziert eine eigene Programmiersprache OpenCL-C und besitzt einen Compiler um diese zur Laufzeit des Host-Programmes für das gewünschtes Device zu einem Kernel zu kompilieren. OpenCL-C basiert auf C99 und definiert zusätzlich eine Vielzahl von Vektordatentypen und Operationen für diese Datentypen. Allerdings besitzt sie diverse zusätzliche Limitierungen, wie keine Rekursion bei Funktionsaufrufen oder keine Unterstützung für Funktionspointer.

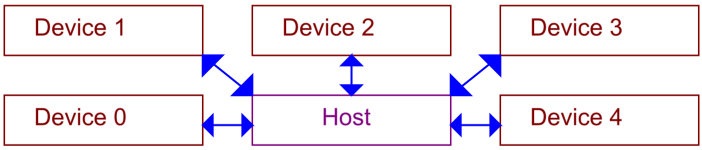


Abbildung 2.2: OpenCL-Host mit mehreren Devices.

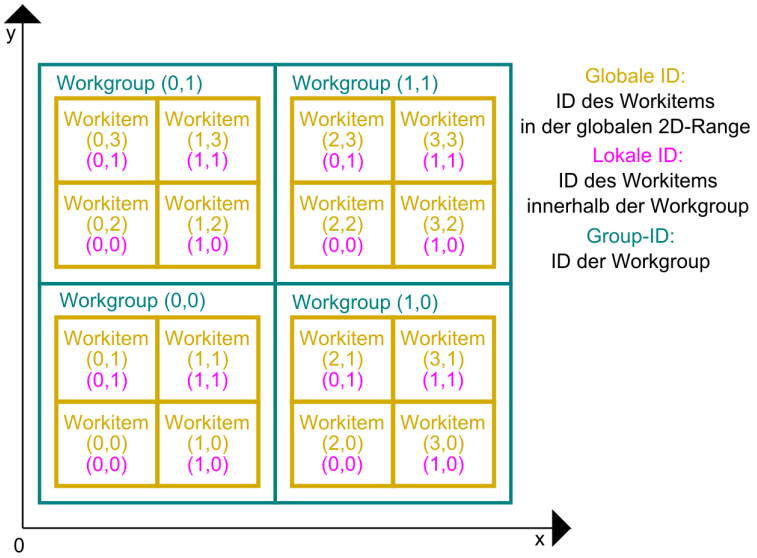


Abbildung 2.3: **Globale OpenCL-2D-Range.**

Beim Ausführen eines Kernels definiert man einen bis zu dreidimensionalen "globalen" Indexraum, ND-Range genannt. Eine solche globale ND-Range ist in Abbildung 2.3 gezeigt. Für jeden Index wird nun eine eigene Instanz des Kernels ausgeführt, welche als Workitem bezeichnet wird. Diese Workitems werden noch einmal zu gleich grossen Gruppen, den so genannten Workgroups zusammengefasst. Workgroups haben die selbe Dimension wie die globale ND-Range. Ein Workitem kann seine Position bzw. ID innerhalb der globalen ND-Range und innerhalb der Workgroup abfragen und abhängig von diesen Positionen seinen eigenen dynamischen Kontrollfluss beschreiten. Es existieren in OpenCL-C nur Barriers für die Workitems innerhalb einer Workgroup, jedoch keine Synchronisationsmöglichkeiten für Workitems aus verschiedenen Workgroups. Ein Quelltext eines OpenCL-Beispielprogramms wird in Abbildung 2.4 und die Ausführung des Beispielprogrammes in Abbildung 2.5 gezeigt.

Als Nächstes sollen kurz das OpenCL-Speichermodell und dessen Objekte erläutert werden. Dieses Speichermodell wird in Abbildung 2.6 dargestellt. OpenCL spezifiziert sogenannte Bufferobjekte um Speicherplatz auf dem Device zu allokieren. Der Host kann durch Buffer Daten mit dem Device austauschen. Ebenfalls können Kernels in diese Buffer hineinschreiben und aus diesen Buffern lesen. Zusätzlich besitzt OpenCL spezielle Image- bzw. Texturobjekte. Diese können mit Nearest-Neighbor-Interpolation oder trilinearer Interpolation und mit normierten oder unnormierten Koordinaten innerhalb eines Kernels gesamplet werden. Auch ist es möglich in diese innerhalb eines Kernels zu schreiben. Gleichzeitiger Lese- und Schreibzugriff auf das

OpenCL-Host-Code in C++ mit dem OpenCL-C++ Wrapper:

```
//Einreihen der Ausführung eines Kernels in eine OpenCL-Warteschlange
clqueue.enqueueNDRangeKernel(
VektorAddition, //Auszuführendes Kernel
cl::NDRange(8),//Globale 1D-Range 8 Workitems werden gestartet
//Dadurch werden die ersten 8 Vektorkomponenten addiert
cl::NDRange(2)) //Workgroupgrösse 2 Workitems
//per Workgroup; daduch werden 4 Workgroups gestartet
clqueue.finish();// Wartet blockierend bis
//die Warteschlange geleert ist
```

OpenCL-C Kernelcode:

```
//Berechnet VektorC = VektorA+VektorB
void kernel VektorAddition(global float* VektorA,
        global float* VektorB,
        global float* VektorC)
{
//Fragt die globale Position des Workitems ab:
int Index=get_global_id(0);
//Berechnet die Summe Vektorkomponenten
//an der zuvor abgefragten Position:
VektorC[Index]=VektorA[Index]+VektorB[Index];
}
```

Abbildung 2.4: **OpenCL-Vektoraddtionskernel**

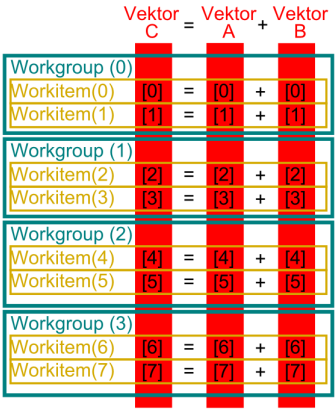


Abbildung 2.5: **Ausführung der OpenCL-Vektoraddition.**

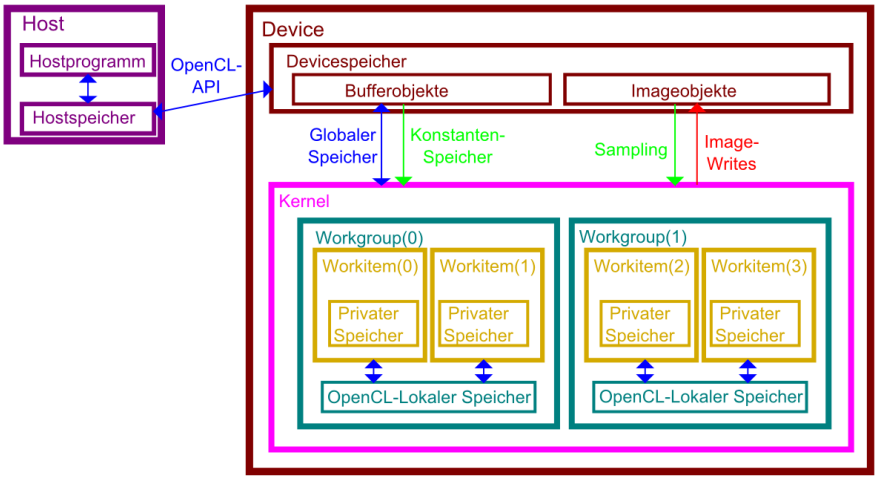


Abbildung 2.6: **OpenCL-Speichermodell.**

selbe Texturobjekt innerhalb eines einzigen Kernels ist nicht möglich. Es gibt diese Texturobjekte in 1D (ab OpenCL 1.2), 2D und 3D.

In einem Kernel existieren folgende Speicherräume:

- **Privater Speicher:** Nur für ein Workitem sichtbar und lebt so lange wie das Workitem.
- **OpenCL-lokaler Speicher:** Nur für eine Workgroup sichtbar und lebt so lange wie die Workgroup.
- **Konstanten-Speicher:** Man kann einem Buffer diesen Speicherraum zuweisen. Dann haben alle Workitems wahlfreien Lesezugriff auf den Speicher.
- **Globaler Speicher:** Ebenfalls kann man einem Buffer diesen Speicherraum zuweisen. Dann haben alle Workitems wahlfreien Lese- und Schreibzugriff auf diesen Buffer.

2.2 OpenCL-Implementierung auf einer GPU und GPGPU-Grundlagen

2.2.1 Allgemeines

Als Nächstes soll erläutert werden, wie die OpenCL-Spezifikationen auf einer Graphikkarte implementiert werden. Da jeder GPU-Entwickler seine eigene Nomenklatur

eingeführt hat und das OpenCL-Modell zu abstrakt für genauere Erklärungen der GPU-Architektur ist, werden im folgenden die Nvidia-Begriffe des CUDA-Modells verwendet. CUDA (Compute Unified Device Architecture) selbst ist eine alternative API für die GPGPU-Programmierung, welche allerdings im Gegensatz zu OpenCL speziell für Graphikkarten entwickelt worden ist. Aus diesem Grund wird durch CUDA auch die Hardware der Graphikkarten genau spezifiziert. Nvidia bezeichnet die Architektur seiner Graphikkarten deshalb als CUDA-Architektur. Um zu erklären wie OpenCL auf einer GPU implementiert ist, muss deshalb die Funktionsweise der CUDA-Architektur erläutert werden. Auch gilt es zu erklären, wie OpenCL durch die CUDA-Architektur umgesetzt wird.

So werden [NVa], [NVc], [NVb] und [NVd] als Quellen für die folgenden Erläuterungen verwendet. Diese Quellen werden hier wieder stark vereinfacht und zusammengefasst wiedergegeben. Abgesehen von der unterschiedlichen Nomenklatur sind die Architekturen der verschiedenen Graphikkartenhersteller, wie Intel oder AMD, sehr ähnlich. Auch gibt es kleinere Abweichungen zwischen den verschiedenen Graphikkartengenerationen der Hersteller. Deshalb beziehen sich die folgenden Erklärungen auf eine moderne Nvidia Geforce 580 GTX.

2.2.2 Ausführeinheiten einer GPU

So besteht die Geforce 580 GTX-GPU, wie sie beispielhaft in Abbildung 2.7 gezeigt wird, aus mehreren Multiprozessoren. Die Architektur eines solchen Multiprozessors wird von Nvidia als Fermi bezeichnet. Jeder dieser Fermi-Multiprozessoren, abgebildet in 2.8, besitzt verschiedene Ausführeinheiten für die unterschiedlichen Befehle, welche innerhalb eines OpenCL Kernels auftreten können:

- **CUDA-Cores:** Jeder CUDA-Core besteht aus einer ALU und einer FPU für SP und DP. Die CUDA-Cores sind noch einmal in Gruppen zu Lanes zusammengefasst. So besitzt ein Fermi-Multiprozessor jeweils zwei Lanes CUDA-Cores. Dieses spezielle Architekturmerkmal hat allerdings für die restliche Arbeit nur geringe Auswirkungen, weshalb auf es nicht weiter eingegangen wird.
- **Special-Function-Units (SFUs):** Diese dienen für das Berechnen transzendenter Funktionen (wie zB. Sinus und Cosinus), der Wurzelfunktion, der Potenzfunktion oder der Kehrwertfunktion.
- **Texture-Mapping-Units (TMUs):** Spezielle Texturhardware, welche zum Samplen von Texturen bzw. Images dient. Die TMUs übernehmen sowohl die komplette Adressberechnung der Texel bzw. Voxel im RAM der Graphikkarte als auch die Interpolation. Zusätzlich besitzen die TMUs eines Multiprozessors jeweils einen separaten Texturcache, dessen Cachingverhalten auf lokale Zugriffe innerhalb der Textur hin optimiert ist. Jedoch wird dieser Cache nicht nach Schreibvorgängen aktualisiert, weshalb man in einem Kernel nicht gleichzeitig aus einer Textur lesen und in die selbe Textur schreiben kann. Zusätzlich werden die Texturen durch einen einzigen L2-Cache auf der GPU zwischengespeichert. Durch diese separate Hardwareunterstützung wird deshalb beim Sampling einer Textur durch die TMUs ein großer Speedup gegenüber einer Softwareimplementierung erreicht. Auch werden die CUDA-Cores dadurch entlastet und können

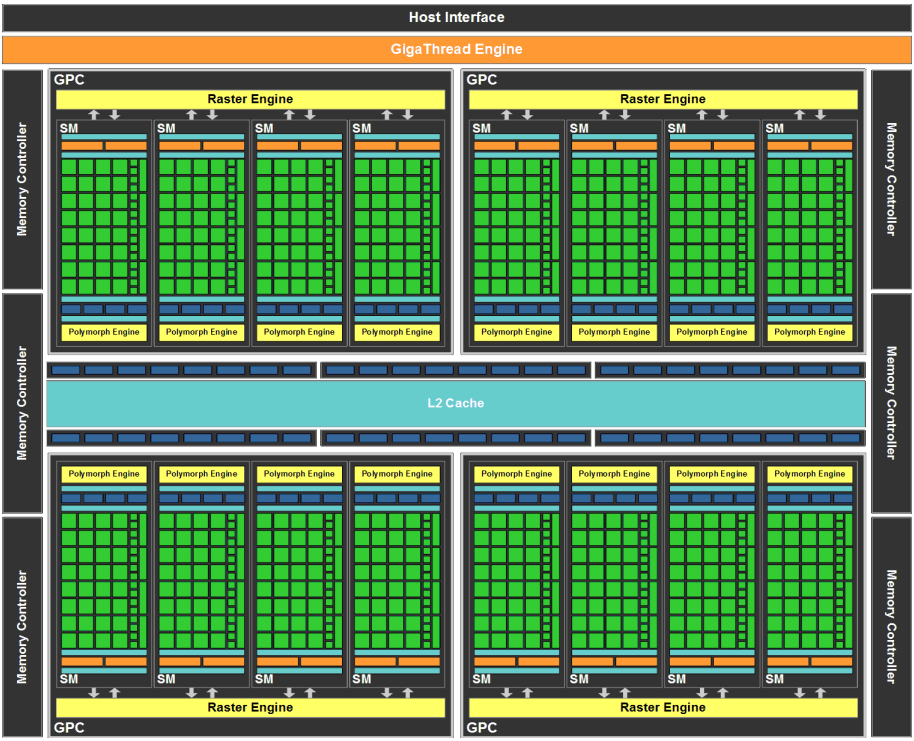


Abbildung 2.7: **Blockdiagramm einer Nvidia 580 GTX.** Quelle für dieses Bild ist [NVc]. SM steht hierbei für Streaming-Multiprocessor, welche in dieser Arbeit einfach Multiprozessor genannt werden. So hat die GPU insgesamt 16 Multiprozessoren. Deutlich zu erkennen ist ebenfalls der L2-Cache in der Mitte und die sechs Memory-Controller für den RAM der Graphikkarte am Rand des Diagramms. Die Gigathread-Engine beinhaltet unter anderem dem Thread-Block-Scheduler. Nicht mit GPGPU-APIs sondern nur mit Rasterisierungs-APIs nutzbar sind die Raster-Engines und die Polymorph-Engines.

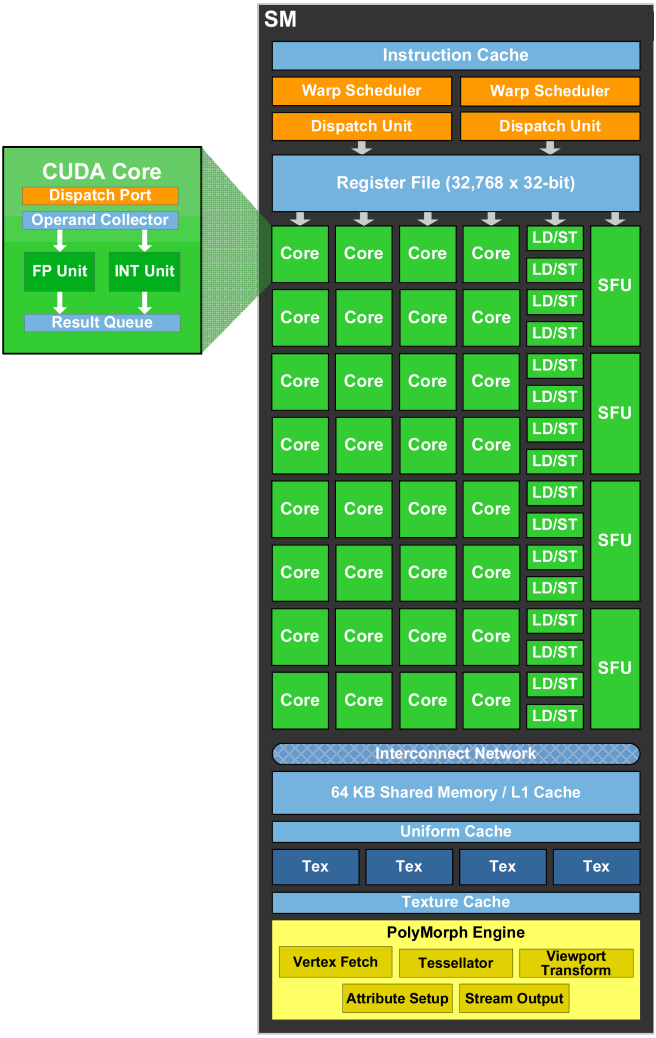


Abbildung 2.8: **Blockdiagramm eines Fermi-Multiprozessors einer Nvidia 580 GTX.** Quelle für dieses Bild ist [NVc] Die Polymorph-Engine ist nur bei Rasterisierungs-APIs und nicht bei GPGPU-APIs verwendbar. Mit LD/ST sind die LSUs gemeint, mit Tex die TMUs und mit Uniform-Cache der Konstanten-Cache. So enthält jeder Multiprozessor 32 CUDA-Cores, 16 LSUs, 4 SFUs, 2 Warpscheduler und 4 TMUs.

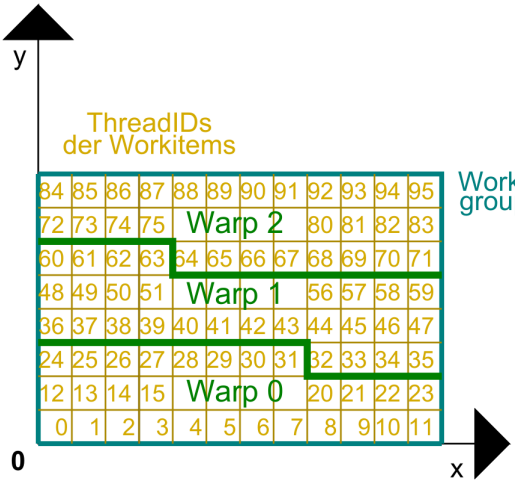


Abbildung 2.9: **Warpaufteilung in einer Workgroup.**

sich in der Zwischenzeit anderen Berechnungen widmen. Dadurch eignen sich diese 3D-Texturen gut, um die verschiedenen Volumendaten darinnen abzuspeichern und anschliessend diese 3D-Texturen mit den TMUs zu samplen.

- **Load-Store-Units (LSUs):** Diese führen Shared-Memory Befehle, CUDA-lokale Speicherbefehle und globale Speicherbefehle aus. Die so eben genannten Speicherbefehle sind auf das CUDA-Speichermodell bezogen, welches in Punkt 2.2.5 erläutert wird.

2.2.3 Ausführungsmodell und Ausführungsverhalten einer GPU

Als Nächstes soll beschrieben werden, wie die GPU eine globale ND-Range ausführt. Wird ein OpenCL-Kernel gestartet, so werden jedem der Multiprozessoren durch einen Thread-Block-Scheduler eine oder mehrere Workgroups, welche in CUDA Thread-Blocks genannt werden, zugewiesen. Preemption bei einer Workgroup ist nicht möglich. So kann dem Multiprozessor erst wieder eine neue Workgroup zugewiesen werden, wenn er mit der Berechnung einer Workgroup abgeschlossen hat.

Ein Workitem selbst entspricht einen Thread. Dieser Thread hat einen eigenen Befehlszähler. Jedem Thread wird eine ThreadID zugeordnet. Jeweils 32 aufeinanderfolgende ThreadIDs in einer Workgroup werden, wie in Abbildung 2.9 gezeigt wird, zu einem Warp zusammengefasst. Ein Multiprozessor besitzt einen oder mehrere Warpscheduler. Ein Warpscheduler selektiert nun einen bereiten Warp nach diversen Schedulingkriterien und gibt dessen nächsten Befehl in Auftrag. Da sämtliche Threadinformationen in den Registern liegen, ist der Kontextwechsel zwischen den Threads bzw.

Warps kostenlos und die Auswahl kann sehr schnell erfolgen. Ein Warp gilt als bereit, wenn alle Warpthreads bereit sind den nächsten Befehl des Warps auszuführen. Dies ist meistens nicht der Fall, wenn die Threads noch wegen Registerabhängigkeiten auf einen Operanden für den nächsten Befehl warten. Mögliche Ursachen hierfür sind Latenzen bei Speicherzugriffen, Cachezugriffen, oder Latenzen, welche durch die Pipelines der Recheneinheiten, wie zB. durch die Pipelines der CUDA-Cores, entstehen. Diese Latenzen werden dadurch überbrückt, dass in einem Multiprozessor mehrere Warps gleichzeitig hausen, so dass im Optimalfall immer ein Warp bereit ist. Somit ist die Vorgehensweise der GPU ein SMT-Verfahren, ähnlich wie es auf vielen modernen CPUs implementiert ist.

Spaltet sich der Kontrollfluss der Threads des Warps auf, so dass die Threads des Warps unterschiedliche Pfade durch das Programm nehmen, so gilt der im Programm als erstes auftretende Befehl als nächster auszuführender Warpbefehl. Threads deren nächster Befehl ein anderer ist, nehmen nicht an diesem Befehl teil. Dieses Phänomen wird als Warpdivergenz oder als Branching innerhalb eines Warps oder allgemein nur als Branching bezeichnet. Um Sprungbefehle zu vermeiden, besitzt die GPU die Möglichkeit kleinere If- oder If-Else-Blöcke mit Hilfe von Predication abzuarbeiten. Threads, deren Predicationregister nicht gesetzt ist, nehmen ebenfalls an einem solchen Befehl nicht teil.

Diese Rechnerarchitektur eines Multiprozessor bzw. dessen Ausführungsverhalten wird von Nvidia Single-Instruction-Multiple-Threads, abgekürzt als SIMT, genannt. SIMT ist eine Obermenge von SIMD. Die SIMD-Effizienz wird bei SIMT definiert als:

$$\text{SIMD-Effizienz} = \frac{\text{An einer Instruktion teilnehmende Threads}}{\text{Warpgrösse}}$$

Eine niedrige SIMD-Effizienz kann sich auf vielen Wegen negativ auf die Performance auswirken:

- **CUDA-Cores:** Bei den CUDA-Cores ist die Zuordnung zwischen den Warpthreads und den Rechenkernen statisch. Dadurch ist die Zeit, die der Warp zum Ausführen eines Befehls benötigt, unabhängig davon, wie viele Threads des Warps am Ausführen dieses Befehls teilnehmen. Diejenigen CUDA-Cores, deren Threads nicht an dieser Instruktion teilnehmen, bleiben unbenutzt. Dadurch ist die erzielte Rechenleistung bei einem CUDA-Core-Befehl direkt proportional zur SIMD-Effizienz.
- **SFUs:** Die selbe statische Zuordnung erfolgt bei den SFUs mit den selben Auswirkungen.
- **LSUs:** Das genaue Verhalten von LSUs in Bezug auf die SIMD-Effizienz ist leider von Nvidia nicht genau dokumentiert. Jedoch kann zumindest die Dauer von der Zahl der teilnehmenden Threads abhängig sein, so dass eine Instruktion für den kompletten Warp bei hoher SIMD-Effizienz länger dauern kann. Denn je mehr Threads teilnehmen, desto mehr Cache- oder Speicherzugriffe sind für die Abarbeitung des Befehls nötig.

- **TMUs:** Auch das Verhalten der TMUs ist bezüglich der SIMD-Effizienz nicht dokumentiert. Selbst ausgeführte Benchmarks haben ergeben, dass sich die TMUs dynamisch auf die teilnehmenden Threads aufteilen. Im Falle, dass keine Cache-Misses auftreten, ist die Dauer in etwa direkt proportional zu den teilnehmenden Threads und die SIMD-Effizienz spielt keine grosse Rolle. Bei Cache-Misses erlaubt eine hohe SIMD-Effizienz es den TMUs intern die auftretenden Latenzen zu verbergen, so dass ihre Samplingperformance nicht unter den Latenzen leidet.

Allerdings ist selbst bei den TMUs und den LSUs eine hohe SIMD-Effizienz wünschenswert, damit die Warpscheduler möglichst wenige Befehle für Speicherinstruktionen oder Textursampling-Instruktionen herausgeben müssen. Somit ist es eine der obersten Prioritäten bei der GPU-Programmierung die SIMD-Effizienz zu optimieren.

Generell kann man die Ausführeinheiten auch nach der Exklusivität ihrer Belegtheit unterscheiden:

- Die CUDA-Cores und die SFUs sind exklusiv von einem Warp belegt.
- Die LSUs und die TMUs können von mehreren Warps gleichzeitig belegt werden, um dadurch Latenzen, welche durch Cache- oder Speicherzugriffe entstehen, zu verbergen, wodurch die Performance der Ausführeinheiten erhöht wird.

2.2.4 Auswirkung von Sprungbefehlen auf die Performance

Als Nächstes sollen kurz die Auswirkungen von Sprungbefehlen bezüglich der SIMD-Effizienz auf die Performance erläutert werden. Dabei wird zusätzlich angenommen, dass sich innerhalb der Sprungbefehle keine weiteren Sprungbefehle befinden, und diejenige Zeit, welche ein Warp benötigt um einen Befehl auszuführen, unabhängig von der Zahl der teilnehmenden Threads sei.

- **If-Block ohne Predication:** Wenn auch nur ein Warpthread in das If springt, so wird der komplette If-Block ausgeführt. Die für das Ausführen benötigte Zeit des If-Blocks ist in diesem Fall unabhängig davon, wie viele Threads insgesamt in den If-Block springen. Dementsprechend ist in diesem Fall die erzielte Rechenleistung direkt proportional zu dem Anteil der Warpthreads, welche in den IF-Block springen.
- **If-Else-Block ohne Predication**: Wie beim If-Block, nur dass wenn auch nur ein Thread in den Else-Block springt, innerhalb des Else-Blocks die Rechenleistung ebenfalls direkt proportional zu dem Anteil der Warpthreads ist, welche in den Else-Block springen.
- **Schleifen:** Eine Schleife wird immer so oft durchlaufen, bis der am längsten iterierende Warpthread sie verlassen hat. Dementsprechend ist die erzielte Rechenleistung direkt proportional zu der über alle Schleifendurchläufe gemittelten SIMD-Effizienz. So ist die erzielte Rechenleistung besonders niedrig, wenn sich die Zahl der Schleifendurchläufe der Warpthreads stark voneinander unterscheidet. Dies ist insbesondere der Fall, wenn viele Warpthreads keine oder

nur sehr wenige Schleifendurchläufe ausführen, während wenige Warpthreads die Schleife sehr oft durchlaufen.

Bei Predication ergibt sich folgendes:

- **If-Block mit Predication:** Die Ausführung dieses Blocks dauert immer gleich lang, unabhängig davon ob überhaupt ein Thread in den If-Block springt. So ist hier die erzielte Rechenleistung immer direkt proportional zur SIMD-Effizienz.
- **If-Else-Block mit Predication:** Die Ausführung des If-Blocks hier dauert ebenfalls immer gleich lang, unabhängig davon ob überhaupt ein Thread in den If-Block springt. Ähnlich verhält es sich mit dem Else-Block. So dauert die Ausführungszeit des Else-Blocks ebenfalls immer gleich lange, unabhängig davon ob ein Thread nicht in den If-Block sondern in den Else-Block springt. So ist sowohl die Rechenleistung des Else-Blocks als auch des If-Blocks direkt proportional zur SIMD-Effizienz innerhalb des entsprechenden Blocks.

2.2.5 Speichermodell einer GPU

Das OpenCL-Speichermodell bildet sich wie folgt auf eine Graphikkarte ab:

- **Privater Speicher**: Der private Speicher besteht in CUDA aus zwei Speicherbereichen:
  - **Registerspeicher:** Zuerst wird für den privaten Speicher der Registersatz des Multiprozessors verwendet. Selbst wenn ein Thread nicht auf die Register eines anderen Threads zugreifen kann, so teilen sich alle Threads der Workgroups, welche ein Multiprozessor gerade bearbeitet, den selben Registerspeicherplatz. Allerdings ist der Registersatz eines Multiprozessors im Vergleich zu CPUs sehr gross, so dass ein Multiprozessor viele Threads gleichzeitig beherbergen kann. So besitzt ein Fermi-Multiprozessor 32768 32-Bit Register. Zusätzlich gibt es eine Hardwarelimitierung, wie viele Register ein Thread maximal belegen darf. Bei einem Fermi-Multiprozessor sind dies 64 Register. Auch besteht eine Möglichkeit, beim Kompilieren des Kernels die Zahl der Register durch einen Compilerbefehl auf einen bestimmten Wert zu begrenzen. Dadurch kann ein Multiprozessor mehr Threads gleichzeitig beherbergen.
  - **CUDA-lokaler Speicher:** Dieser Speicherraum ist nicht zu verwechseln mit dem OpenCL-lokalen Speicher. Der CUDA-lokale Speicher selbst liegt im RAM der Graphikkarte, wird allerdings durch den L1-Cache und einen einzigen L2-Cache zwischengespeichert. Reicht die entweder durch Hardwarelimitierung oder durch den Compilerbefehl gegebene maximale Zahl an Registern nicht für den privaten Speicherverbrauch des Kernels aus, so wird der Registerinhalt in den CUDA-lokalen Speicher ausgelagert. Dieser Vorgang wird als Registerspilling bezeichnet. Auch werden Arrays, auf welche indexiert zugegriffen wird, in diesem CUDA-lokalen Speicher abgespeichert, da auf die Register nicht indexiert zugegriffen werden kann.

- **Der OpenCL-lokale Speicher wird in CUDA als Shared-Memory bezeichnet.** Dieser ist auf dem GPU-Chip und besteht aus mehreren Bänken, wobei die Speicheradressen interleaved auf die Bänke abgebildet werden. Jede dieser Bänke kann maximal eine Anfrage gleichzeitig bearbeiten. Treffen bei einem Speicherzugriff durch einen Warp bei einer Bank mehrere Anfragen gleichzeitig ein, so entsteht ein Bank-Conflict und diese Anfragen werden sequentiell abgearbeitet. Fragen mehrere Threads eines Warps gleichzeitig den Inhalt der selben Shared-Memory Speicheradresse an, so wird dieser durch die Bank an all diese Threads durch eine einzige Operation gebroadcastet. Auch kann man auf einem Fermi-Multiprozessor einen Teil des L1-Caches zu Shared-Memory umwandeln.
- **Konstanten-Speicher:** Dieser liegt ebenfalls im RAM der Graphikkarte. Jeder Multiprozessor besitzt jedoch einen kleinen und performanten Konstanten-Cache für diesen Konstanten-Speicher. Dieser Konstanten-Cache sequentialisiert Zugriffe, wenn nicht alle Threads eines Warps auf die selbe Adresse zugreifen.
- **Globaler Speicher:** Der globale Speicher liegt im RAM der Graphikkarte. Dieser wird jedoch von dem L1-Cache jedes Multiprozessors und zusätzlich von einem einzigen L2-Cache zwischengespeichert.

2.2.6 Occupancy

Die Zahl der Workgroups, welche ein Multiprozessor gleichzeitig bearbeiten kann, wird durch die Ressourcen des Multiprozessors, den Registerverbrauch eines Threads und dem Shared-Memory-Verbrauch einer Workgroup begrenzt. Auch besteht eine Hardwarelimitierung, wie viele Warps und Workgroups ein Multiprozessor gleichzeitig verwalten kann. So kann zB. ein Fermi-Multiprozessor einer Geforce 580 GTX maximal bis zu acht Workgroups und 48 Warps verwalten. Somit muss eine Workgroup mindestens sechs Warps beinhalten, damit der Multiprozessor die maximal verwaltbare Warpzahl beherbergen kann. In diesem Fall darf ein Thread höchstens 21 Register benötigen. Benötigt er mehr, so reduziert sich die Zahl an Warps, welche ein Multiprozessor beherbergen kann. Ähnliches gilt für dem Shared-Memory-Verbrauch einer Workgroup. Diejenige maximale Warpzahl, welche bei einem bestimmten Kernel und einer bestimmten Workgroupgrösse auf einem Multiprozessor maximal aktiv sein können, wird als Occupancy bezeichnet.

Eine hohe Occupancy wirkt sich aus den selben Gründen, wie viele gleichzeitig aktive Threads auf einem SMT-Prozessor, vorteilhaft auf die Performance aus:

- Die Latenzen bei Registerabhängigkeiten können besser überbrückt werden.
- Es existiert ein grösserer Pool an Befehlen, welche die Warpscheduler als Nächstes in Auftrag geben können. Dadurch erhöht sich die Wahrscheinlichkeit, dass für jede der verschiedenen Ausführungseinheiten ein Befehl vorhanden ist.

Durch beides wird im Endeffekt die Auslastung der GPU durch das Programm verbessert. Wie hoch die Occupancy sein muss, damit die GPU gut ausgelastet wird, so

dass eine höhere Occupancy keinen Performancegewinn mehr bringt, ist von Fall zu Fall verschieden. Ist das Kernel sehr inhomogen, so dass verschiedene Ausführeinheiten jeweils über einen kurzen Zeitraum sehr stark ausgelastet werden, so wird man viele Warps und damit eine hohe Occupancy benötigen, um die Ausführeinheiten immer gleichmässig auszulasten. Wenn ein Kernel viele Registerabhängigkeiten mit grossen Latenzen zB. durch RAM-Zugriffe besitzt, werden ebenfalls viele Warps und damit eine hohe Occupancy benötigt. Es gibt die Möglichkeit die Occupancy durch die Begrenzung der Register mit Hilfe eines Compilerbefehls zu erhöhen. Dadurch kann man zwar die Latenzen besser überbrücken und damit auch potentiell die Performance verbessern, jedoch entsteht dabei allerdings zusätzliches Registerspilling. Dies verursacht wieder zusätzliche Latenzen und verbraucht mehr Cache- und Speicherbandbreite, wodurch die Performance potentiell reduziert werden kann. Deshalb ist es stark fallabhängig, ob eine Begrenzung der Register die Performance verbessert.

Des Weiteren können sich die Warps je nach Kernel in ihrer Laufzeit stark voneinander unterscheiden. Ein Multiprozessor kann jedoch erst mit einer neuen Workgroup beginnen, wenn alle Warps einer Workgroup terminiert haben. Auch kann es sein, dass am Ende der Ausführung einer globalen ND-Range keine neuen Workgroups mehr zur Verfügung stehen, weshalb dem Multiprozessor keine neue Workgroup mehr zur Bearbeitung zugeteilt werden kann. Dadurch kann es zu einem bestimmten Zeitpunkt vorkommen, dass auf einem Multiprozessor weniger Warps aktiv sind, als es die Occupancy erwarten lässt. Deshalb definiert man die Achieved-Occupancy, als diejenigen Warpzahl, welche zeitlich und über alle Multiprozessoren gemittelt während der Ausführung einer globalen ND-Range aktiv sind.

Sowohl die Occupancy als auch die Achieved-Occupancy werden von Nvidia oft normiert auf diejenige Warpzahl angegeben, welche ein Multiprozessor maximal verwalten kann.

2.3 Limitierungen von OpenCL bei GPGPU

OpenCL soll auf den verschiedenen potentiellen Devices wie GPUs, SPEs der CELL-Prozessoren oder sogar auf CPUs möglichst performant funktionieren. Dadurch beinhaltet es alle spezifischen harten Limitierungen der verschiedenen Devices. Die unterstützten Features stellen ebenfalls einen Kompromiss dar, wie viel Sinn sie auf den unterschiedlichen Devices ergeben und ob sie sich auf allen Devices umsetzen lassen. Das Ausführungsmodell wird auch nicht genau sondern nur abstrakt als SIMD oder SPMD spezifiziert, damit jedes Device sein spezifisches Ausführungsmodell verwenden kann. Jedoch ist es beim SIMT-Modell wichtig dass man auf es explizit eingeht, da es zu einem der grössten Performance Probleme werden kann. Dementsprechend kennt OpenCL keinen Warpbegriff. Allerdings ist es immer sinnvoll das Kernel so zu entwerfen, dass die Warpthreads einen möglichst ähnlichen Kontrollfluss und synchronisierte Speicherzugriffe haben. Da in OpenCL deshalb auch nicht spezifiziert ist, wie sich die Warps innerhalb einer Workgroup aufteilen, muss man hierfür Annahmen treffen. Diese Aufteilung kann allerdings prinzipiell von der Graphikkarte abhängig sein, wodurch diese Annahmen riskant sind. Auch benötigt man oft Kommunikation innerhalb des Warps. Die Kommunikation lässt sich nur wenig performant über den Shared-Memory implementieren. Dies ist sofern ärgerlich, da moderne Nvidiakarten

prinzipiell für solche Warpoperationen in CUDA spezielle Funktionen (sogenannte Warp Vote-Functions) mit Hardwareunterstützung hätten. Zudem verwendet man in kritischen Codeabschnitten aus Performancegründen oft die Annahme, dass die Befehle der Threads eines Warps synchron abgearbeitet werden. So wird in dieser Arbeit aus Performancegründen sehr oft Code benötigt, welcher sich nur durch obige Annahmen verwirklichen lässt. Beispielhaft sei folgender Code genannt:

Unoptimierter Code:

```
if(BufferDesThreadsIstLeer())
  DannSollDerThreadSeinenBufferFüllen();
```

Auf SIMT-Modell optimierter Code:

```
if(BufferInMindestensEinemThreadDesWarpsLeer())
  DannSollenAlleWarpthreadsIhrenBufferFüllen();
```

Beim unoptimierten Code würde nun diejenigen Warpthreads ihren Buffer füllen, bei denen der Buffer leer ist. Dies würde beim Füllen eine niedrige SIMD-Effizienz bedeuten und Rechenzeit würde verloren gehen. Bei der Optimierung wird dies dadurch umgangen, dass die Entscheidung des Füllens nicht auf Thread-Basis sondern auf Warp-Basis ausgeführt wird. Dadurch nehmen alle Threads des Warps am Füllen des Buffers teil, wenn auch nur der Buffer eines Threads leer ist. So wird die SIMD-Effizienz erhöht. Dies führt nun insgesamt dazu, dass aus Performancegründen gezwungenermassen für Graphikkarten optimierter Code in vielen Fällen auf anderen Devices langsam läuft oder Fehler erzeugt. Auf diese Weise wird der Gedanke hinter OpenCL, eine offene und soweit abstrakte Programmierplattform zu sein, so dass mit dieser Plattform geschriebene Programme auf allen unterstützen Devices funktionieren, ad absurdum geführt.

3 Standard-Volumenraycaster

3.1 Einleitende Bemerkungen

Vektoren werden in dieser Arbeit mit $\overrightarrow{Pfeil}$ dargestellt und Punkte werden **Fett** geschrieben. In der folgenden Ausarbeitung wird die OpenCL-Syntax für Vektorrechnungen verwendet. Um die Komponenten eines Vektors zu bezeichnen wird folgende Notation verwendet:

$$\overrightarrow{a} = \begin{pmatrix} \overrightarrow{a}.x \\ \overrightarrow{a}.y \\ \overrightarrow{a}.z \end{pmatrix}$$

Die Syntax definiert die Verknüpfung zweier Vektoren $\overrightarrow{a}$ und $\overrightarrow{b}$ mit dem Multiplikationsoperator und Divisionsoperator als:

$$\overrightarrow{a} * \overrightarrow{b} = \begin{pmatrix} \overrightarrow{a}.x * \overrightarrow{b}.x \\ \overrightarrow{a}.y * \overrightarrow{b}.y \\ \overrightarrow{a}.z * \overrightarrow{b}.z \end{pmatrix} \qquad \frac{\overrightarrow{a}}{\overrightarrow{b}} = \begin{pmatrix} \frac{\overrightarrow{a}.x}{\overrightarrow{b}.x} \\ \frac{\overrightarrow{a}.y}{\overrightarrow{b}.y} \\ \frac{\overrightarrow{a}.z}{\overrightarrow{b}.z} \end{pmatrix}$$

Multiplikationen und Divisionen zwischen zwei Punkten werden analog definiert. Zusätzlich definiert man analog zu den eben genannten Regeln die folgenden Operationen zwischen Punkten und Vektoren:

$$\mathbf{Punkt}_c = \overrightarrow{Vektor}_a * \mathbf{Punkt}_b$$

$$\mathbf{Punkt}_c = \frac{\mathbf{Punkt}_a}{\overrightarrow{Vektor}_b}$$

$$\overrightarrow{Vektor}_c = \frac{\overrightarrow{Vektor}_a}{\mathbf{Punkt}_b}$$

Auch werden Addition und Subtraktion zwischen einem Vektor $\overrightarrow{a}$ und einem Skalar s definiert als:

$$\overrightarrow{a} + s = \begin{pmatrix} \overrightarrow{a}.x + s \\ \overrightarrow{a}.y + s \\ \overrightarrow{a}.z + s \end{pmatrix} \qquad \overrightarrow{a} - s = \begin{pmatrix} \overrightarrow{a}.x - s \\ \overrightarrow{a}.y - s \\ \overrightarrow{a}.z - s \end{pmatrix}$$

Diese beiden Operationen werden für Punkt und Skalar ebenfalls analog definiert. Diese eigenartig anmutenden Definitionen für Punkt- und Vektorrechnung werden sich in der Arbeit als vorteilhaft erweisen, da man sonst bei vielen Gleichungen auf die umständlichere (homogene) Matrizenrechnung zurückgreifen müsste.

An vielen Stellen der Literatur wird der Begriff Alpha bzw. Alphakanal und Opazität gleichrangig verwendet. Da der Begriff der Opazität eindeutiger ist, wird im Folgenden dieser verwendet. Diverse Fachbegriffe, die sich von diesem Alpha (zB. Alpha-Compositing) ableiten, werden so übernommen.

OpenCL und Cuda haben bedauerlicherweise, obwohl sie oft ein und das selbe meinen, eine andere Nomenklatur. Da die Arbeit einerseits im OpenCL geschrieben worden ist, andererseits das Cuda-Modell für genauere Erklärungen benötigt stellt die Nomenklatur ein Problem dar. Dieses Problem wurde wie folgt gelöst:

- Es werden die Begriffe des Cuda-Speichermodells verwendet.
- Es wird der Threadbegriff verwendet wenn er sich auf das Cuda-Ausführungsmodell bezieht. Ein Thread wird als Workitem bezeichnet, wenn ein Bezug zum OpenCL Ausführungsmodell besteht.
- Der Begriff Workgroup wird dem Threadblockbegriff vorgezogen.
- Statt dem Image-Begriff aus OpenCL wird in dieser Arbeit der Texturbegriff verwendet, da der Texturbegriff aus dem Graphikkartenjargon entstammt.

3.2 Volumendaten

Die für dieses Verfahren verwendeten Volumendaten stammen von [VOL]. Sie sind entweder synthetisch erzeugt worden oder stammen aus MRT oder CT-Aufnahmen und bestehen aus einem regelmässigen und quaderförmigen Voxelgitter. Entlang der Kanten besitzt das Gitter $\overrightarrow{V}$ Voxel. Bei den Volumendaten besitzt jeder Voxel einen Opazitätswert und auf Grund des MRTs bzw. CTs eine unterschiedliche Kantenlänge $\overrightarrow{v}$ entlang jeder Achse. Die Volumendaten werden dabei so interpretiert, dass der Mittelpunkt des ersten Voxels des Gitters im Nullpunkt des Weltkoordinatensystems liegt und die Achsen des Gitters den Achsen des Weltkoordinatensystems entsprechen. Dadurch lassen sich die Volumendaten leicht durch eine Axis-Aligned-Boundingbox begrenzen. Zusätzlich wird festgelegt, dass die Boundingbox im ersten Quadranten des Weltkoordinatensystems liegt. Dies wird in Abbildung 3.1 gezeigt. So ist **P** die dem Ursprung gegenüberliegende Boundingboxecke und das Zentrum des letzten Voxels und **N** ist die im Ursprung liegende Boundingboxecke und Zentrum des ersten Voxels.

In der Arbeit werden normalisierte Volumenkoordinaten benötigt. Diese seien wie folgt definiert:

- Die Achsen der normalisierten Volumenkoordinaten seien parallel zu denen der Weltkoordinaten
- **P** liege im normalisierten Volumenkoordinaten bei (1,1,1)
- **N** liege in beiden Koordinatensystemen im Nullpunkt

So gilt für eine Transformation für einen beliebigen Punkt **K** von Weltkoordinaten in normalisierte Volumenkoordinaten:

$$\mathbf{K}_{NormVolumen} = \frac{\mathbf{K}_{Welt}}{\mathbf{P}_{Welt}}$$

Eine Transformation für einen Vektor erfolgt analog.

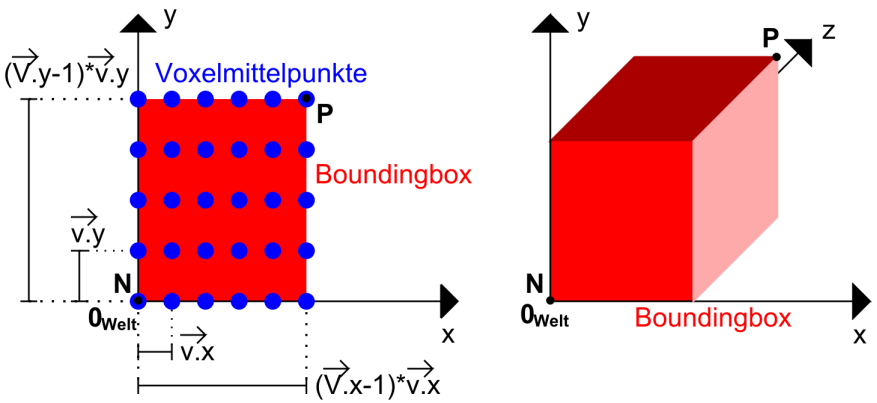


Abbildung 3.1: **Boundingbox der Volumendaten.**

3.3 Allgemeines

Ein Volumenraycaster sendet für jeden Bildpunkt einen Strahl aus und berechnet anhand der Volumendaten entlang des Verlaufs des Strahls die Farbe des Bildpunkts.

Zuerst werden bei diesem Verfahren die Opazitätswerte der Volumendaten in einer Opazitätstextur auf der Graphikkarte abgespeichert. Danach wird in einem vorbereitenden Schritt aus der Opazitätstextur mit Hilfe einer Pre-Classified Transferfunktion eine Farbtextur und eine transformierte Opazitätstextur für das spätere Raymarching erzeugt. Diese Transferfunktion ermöglicht es dem Benutzer die Volumendaten einzufärben und bestimmte Details hervorzuheben. Auch wird eine dreidimensionale Normalentextur aus dem Gradienten der transformierten Opazitätstextur vorbereitet. Diese wird später für die Lichtberechnung benötigt.

Nun beginnt das eigentliche Raycasting-Verfahren mit der Ausführung des Raycasting-Kernels. Eine Übersicht der Schritte dieses Verfahrens wird in Abbildung 3.2 gezeigt. Beim Starten ordnet dieses Kernel jeweils einem Pixel des Ausgabebildes einen entsprechenden Thread bzw. Workitem zu.

Jeder Thread arbeitet anschliessend die folgenden Schritte ab:

- **Strahlerstellung:** Hier wird für den zum entsprechenden Thread gehörenden Pixel die Strahlengerade berechnet.
- **Schnittpunkteberechnung:** Bei diesem Schritt wird die Strahlengerade mit der Boundingbox der Volumendaten geschnitten, um den Startpunkt und den Endpunkt für das Raymarching zu erhalten.

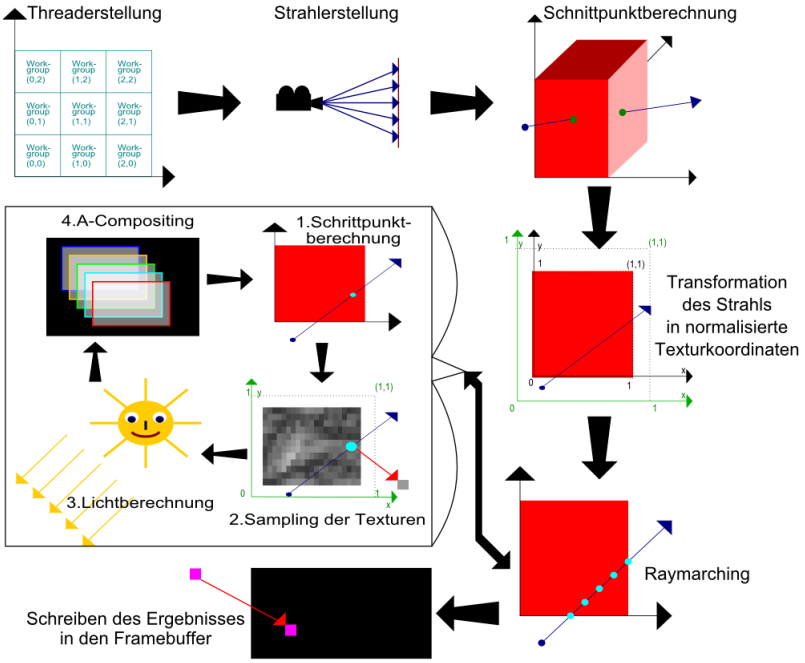


Abbildung 3.2: **Übersicht der Vorgehensweise beim Standard-Volumenraycaster.**

- **Transformation in normalisierte Texturkoordinaten:** In einem weiteren Schritt wird der Strahl zum einfacheren Samplen der Texturen in normalisierte Texturkoordinaten transformiert.
- **Raymarching:** Nun findet das Raymarching statt. Hier marschiert man in regelmässigen Schritten entlang des Strahles durch das Volumen. Jeder Raymarchingschritt lässt sich noch einmal in folgende Unterpunkte unterteilen:
  - Zuerst wird die Schrittposition im Volumen berechnet.
  - Als Nächstes werden die verschiedenen Texturen an der Schrittposition gesamplet
  - Mit Hilfe dieser Samples wird eine Lichtberechnung durch das Phong-Beleuchtungsmodell ausgeführt
  - Mit den Samples und dem Ergebnis der Lichtberechnung wird das A-Compositing ausgeführt. Das A-Compositing ist eine Transparenzberechnung und bestimmt, wie viel der jeweilige Schritt zur Gesamtfarbe des Pixels beiträgt.

- **Schreiben in den Framebuffer:** Im letzten Schritt wird die berechnete Farbe an der Stelle des Pixels in den Framebuffer geschrieben.

Nachdem alle Threads fertig sind, wird das so eben gerenderte Bild im OpenCL-Framebuffer mit Hilfe der OpenCL-OpenGL-Interoperabilität in den OpenGL-Framebuffer eines Fensters kopiert und danach angezeigt. Dieser Vorgang findet ebenfalls komplett auf der Graphikkarte statt.

3.4 Anmerkungen

Selbst bei diesem grundlegenden Verfahren gibt es viele kleinere und grössere Abweichungen in der Vorgehensweise. Gerade bei der Komposition der einzelnen Samples des Raymarchings, dem Einfärben und der Lichtberechnung gibt es sehr viele Ansätze. Der zu wählende Ansatz hierbei ist davon abhängig, was visualisiert werden soll und wofür diese Visualisierung dienen soll. So wird man für wissenschaftliche oder medizinische Visualisierungen eher abstrakte Modelle, auf welchen man die dreidimensionale Struktur des Volumens und die interessanten Merkmale möglichst leicht erkennen kann, wählen. Jedoch wird man sich in einem Computerspiel für möglichst photorealistische und schnell zu berechnende Modelle entscheiden. Für Filme können die Berechnungen nahezu beliebig lange dauern, weshalb man viel aufwändigere Modelle wählen kann.

Wegen der vielen verschiedenen Möglichkeiten wird in diesem Kapitel nur dasjenige grundlegende Verfahren erläutert, welches in dem praktischen Teil der Arbeit implementiert worden ist. Selbst wenn dieses Verfahren in seiner speziellen Implementierung einzigartig ist, so sind viele der einzelnen Bestandteile als Standard oft in entsprechender Fachliteratur wie zB. in [RTVG06] oder [AK05] erläutert. Des Weiteren sind alle Schritte nach dem Starten des Kernels peinlich parallel auf Pixelbasis. Dennoch beinhalten diese in den folgenden Kapiteln vorgestellten Berechnungen Schritte, welche für alle Pixel eines Bildes identisch sind. Diese können auf der CPU vorberechnet und deren Ergebnisse mit Hilfe eines Konstantenbuffers innerhalb des Kernels ausgelesen werden.

3.5 Vorbereitende Schritte

3.5.1 Erstellen einer Farbtextur und transformierten Opazitätstextur durch die Transferfunktion

In diesem Punkt wird die Pre-Classified Transferfunktion auf die aus den Volumendaten erstellte Opazitätstextur angewendet, um eine Farbtextur und eine transformierte Opazitätstextur zu erstellen. Die Anwendung der Transferfunktion auf die Volumendaten stellt eine einfache Möglichkeit dar, durch Einfärben oder Verändern des Opazitätswerts verschiedene Bereiche des Volumens hervorzuheben. Dieser Vorgang wird als Klassifikation bezeichnet. Ein optischer Eindruck der Anwendung der Transferfunktion auf ein Volumenmodell wird in Abbildung 3.3 gezeigt.

Die Transferfunktion ist definiert als:

$$f : Opazität \rightarrow (Opazität', Rot, Grün, Blau)$$

Mit Transferfunktion: Ohne Transferfunktion:

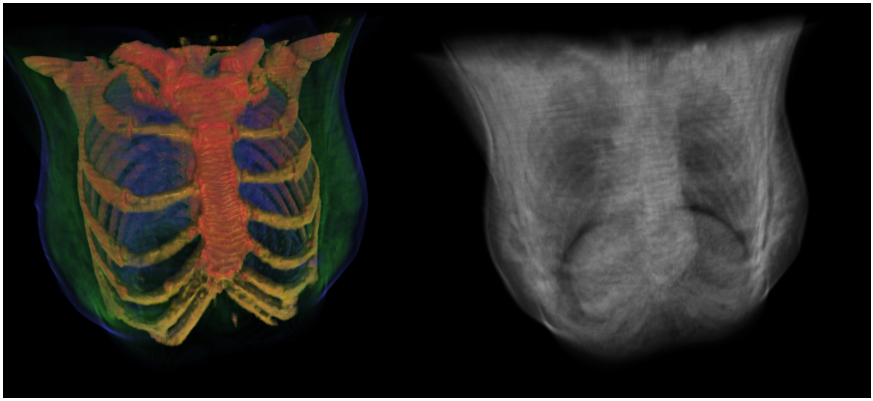


Abbildung 3.3: **Anwendung der Pre-Classified Transferfunktion auf eine CT-Aufnahme eines weiblichen Brustkorbs.** Auf dem linken Bild wurde die Transferfunktion so konfiguriert, dass die Knochen rot und die Knorpel gelb eingefärbt und beide gut sichtbar sind. Dem Muskelgewebe wurde eine grüne und den äusseren Hautschichten und den Lungenbläschen eine blaue Farbe zugewiesen. Trotz Pre-Classified Transferfunktion sind auf dem Brustbein dünne gelbe Linien sichtbar. Hier besitzen also Teile des Brustkorbs, welche in Wirklichkeit Knochen oder Muskelgewebe sein sollten, in der Aufnahme die selbe Opazität wie der Knorpel. Dies ist auf die Verschwommenheit der CT-Aufnahme zurückzuführen. Rechts ist die selbe Aufnahme ohne Transferfunktion zu sehen. Die verschiedenen Gewebesorten sind hier kaum von einander zu unterscheiden.

Die Transferfunktion ist in dem praktischen Teil als Lookuptable bzw. Lookuptexture implementiert, welche der Benutzer des Programmes interaktiv editieren kann.

Generell gibt es beim späteren Raymarching durch die Volumendaten zwei Möglichkeiten die Transferfunktion anzuwenden:

- **Pre-Interpolated (auch Pre-Classified genannt):** Hier wird zuerst die Transferfunktion auf die Opazitätswerte der Volumendaten angewendet und danach das Ergebnis der Transferfunktion, also die Farb- und Opazitätswerte, an der Samplestelle interpoliert.
- **Post-Interpolated (auch Post-Classified genannt):** Bei diesem Modell werden zuerst die Opazitätswerte der Volumendaten an der Samplestelle interpoliert. Danach dient dieser interpolierte Wert als Eingabewert für die Transferfunktion um an der Samplestelle einen Farb- und Opazitätswert zu erhalten.

Beide Möglichkeiten sind in Abbildung 3.4 gezeigt.

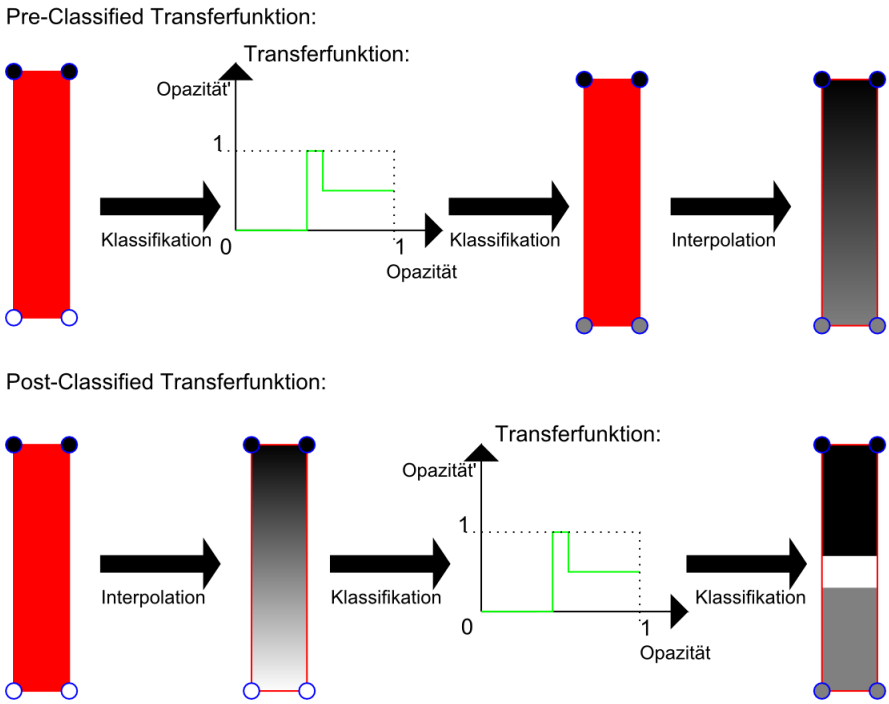


Abbildung 3.4: **Unterschiedliche Vorgehensweisen bei der Pre-Classified und bei der Post-Classified Transferfunktion.**

Welcher dieser beiden Anwendungen eine bessere Bildqualität erzeugt hängt davon ab, ob zwischen den acht benachbarten Voxeln bei der Interpolationsstelle ein stetiger oder ein unstetiger Opazitätsverlauf vorherrscht:

- Als Erstes soll der Fall des unstetigen Opazitätsverlaufs, wie er zB. an Objektkanten und in bereits segmentierten Volumendaten auftreten sollte, betrachtet werden. Hier werden beim Sampling der Opazität durch die Interpolation sämtliche Opazitätswerte zwischen der grössten und der kleinsten Opazität der acht benachbarten Voxel angenommen. Wendet man nun auf die Samples die Post-Classified Transferfunktion an so kommen auch alle Ergebnisse bzw. Klassen der Transferfunktion für dieses Eingabewerteintervall vor. Dadurch können die einzelnen Samples zwischen diesen Voxeln einer bestimmten Klasse zugeordnet werden, obwohl keiner dieser Voxel und wegen des unstetigen Opazitätsverlaufs nichts vom Volumen zu dieser Klasse gehören würde. Diese “falsche“ Klassifika-

tion äussert sich im gerenderten Bild durch Artefakte an den Objektgrenzen, wie ungewollte Isosurfaces oder Farbsäume. Hierdurch wird die Bildqualität reduziert. Der Nachteil wird bei dem Pre-Classified Modell dadurch, dass man die Transferfunktion vor dem Interpolieren anwendet, grösstenteils umgangen. Denn hier treten nur die Klassen der acht benachbarten Voxel auf, zwischen welchen dann interpoliert wird. Zum Teil bleibt dieses Problem jedoch bestehen, da sämtliche MRT und CT-Bilder wegen der beschränkten MRT und CT-Auflösung leicht verschwommen sind. So entstehen an in der Wirklichkeit harten Objektkanten in den Aufnahmen Opazitätsverläufe, welche die Transferfunktion danach ebenfalls "falsch" klassifiziert.

- Als Nächstes folgt der Fall des stetigen Opazitätsverlaufs zwischen den Voxeln. In diesem Fall würden alle Klassen zwischen derjenigen Klasse der grössten und derjenigen Klasse der kleinsten Opazität der acht benachbarten Voxel auftreten. Diese Klassen werden alle nur erkannt, wenn man die Transferfunktion nach der Interpolation, also Post-Classified, anwendet. Wendet man allerdings die Transferfunktion vor Interpolation der Opazität an so erhält man nur diejenigen Klassen der acht benachbarten Voxel, und alle anderen Klassen im Volumen zwischen den Voxeln werden verpasst. Dieses Verpassen tritt vor allem bei hochfrequenten Transferfunktionen und Volumendaten auf. Dadurch entstehen im komplett gerenderten Bild blockige Artefakte oder Unschärfe.

Einen optischer Vergleich der Bildqualitäten der beiden verschiedenen Transferfunktionen ist in Abbildung 3.5 gezeigt.

Während sich die Post-Classified Version "on the fly" beim späteren Raymarching auf der GPU performant implementieren lässt, kann man bei der Pre-Classified Version, wenn man sie ebenfalls "on the fly" anwenden will, nicht mehr auf die hardwaregestützte trilineare Interpolation der TMUs zurückgreifen. Deshalb muss man in diesen Fall die Interpolation in Software durchführen, wodurch die Performance stark reduziert wird. Die im praktischen Teil verwendete Lösung um dennoch die Pre-Classified Version mit der Hardware-Interpolation der GPU benutzen zu können besteht darin, die Transferfunktion in einem vorbereitenden Schritt zuerst auf alle Voxel der Volumendaten anzuwenden. Dadurch erhält man für jeden Voxel einen transformierten Opazitätswert und einen Farbwert, welchen man dann in einer Farbtextur und einer transformierten Opazitätstextur abspeichert. Werden diese beiden Texturen nun beim späteren Raymarching mit trilinearer Interpolation von den TMUs gesamplet, so entspricht das Ergebnis dem der Pre-Classified Transferfunktion. Diese beiden Texturen müssen jedoch bei jeder Änderung der Transferfunktion aktualisiert werden. Das Erstellen dieser beiden Texturen wurde im praktischen Teil auf der GPU implementiert.

3.5.2 Erstellen einer dreidimensionalen Normalentextur aus den Volumendaten

Um das Volumenmodell zu schattieren soll später das Phong-Beleuchtungsmodell verwendet werden. Für dieses Beleuchtungsmodell benötigt man allerdings eine Oberflächennormale in Weltkoordinaten, welche durch den negativen normalisierten Gradienten der transformierten Opazität approximiert wird. Eine einfache Visualisierung, wie

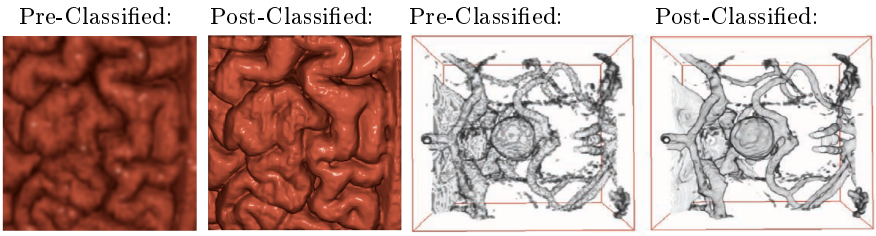


Abbildung 3.5: **Vergleich zwischen der Bildqualität von Post-Classified und Pre-Classified Transferfunktionen.** In den linken beiden Bildern, welche aus [AK05] stammen, konfigurierte man die Transferfunktion so, dass sämtliches Volumen, das über einem bestimmten Opazitätsschwellwert liegt, als komplett undurchsichtig dargestellt wird. Die Pre-Classified Transferfunktion klassifiziert nun im ersten Bild an der Objektkante die Voxel als komplett undurchsichtig oder komplett durchsichtig, worauf die Interpolation erfolgt. Dadurch erhält man einen Opazitätsverlauf von komplett undurchsichtig nach komplett durchsichtig an der Objektkante, wodurch Unschärfe entsteht. Im zweiten Bild interpoliert man jedoch zuerst, wodurch zwischen den Voxeln an der Objektkante eine Oberfläche, innerhalb welcher die Opazität grösser als die Schwellwertkonstante ist, entsteht. Wendet man nun die Post-Classified Transferfunktion an, so wird eben diese Oberfläche visualisiert. Auf diese Art wird diese Unschärfe vermieden. Ähnlich verhält es sich im zweiten Beispiel beim dritten und vierten Bild, welche beide aus [RTVG06] stammen. Hier versucht man durch eine Transferfunktion mit einer hohen Frequenz eine Oberfläche hervorzuheben. Der Verlauf dieser Oberfläche insbesonders zwischen den Voxeln lässt sich, wie auf dem vierten Bild zu sehen, wieder durch die Post-Classified Transferfunktion genauer darstellen. Auf dem dritten Bild erfolgte zuerst durch die Pre-Classified Transferfunktion die Klassifikation, ob ein Voxel zur Oberfläche gehört oder nicht, und danach die Interpolation. Dies führt zu den im dritten Bild sichtbaren blockigen Artefakten.

sich der Gradient und die Normale aus der Opazität ergeben, wird in Abbildung 3.6 gezeigt. Für die Gradienten bzw. Normalenberechnung wurde ein Central-Difference-Filter und ein Sobel-Filter implementiert. Die Faltungs- bzw. Filtermasken sind in Abbildung 3.7 gezeigt.

Da die Berechnung des Gradienten aufwändig ist, wird in diesem vorbereitenden Schritt für jeden Voxel der transformierten Opazitätstextur die Normale berechnet und in einer weiteren 3D-Textur abgespeichert. Die beiden vorgestellten Filter werden, damit die Normale in Welt- und nicht in normalisierten Volumenkoordinaten vorliegt, skaliert, um die ungleichmässige Voxelkantenlänge zu berücksichtigen. Die Skalierung erfolgt so, dass der grösste Abstand zwischen zwei benachbarten Filtersamplepunkten in normalisierten Volumenkoordinaten entlang einer Achse einen Voxel beträgt. Dies entspricht der minimalen Voxelkantenlänge in Weltkoordinaten. Ist die Voxelkantenlänge in Weltkoordinaten entlang einer Achse grösser, so muss der Sampleabstand in normalisierten Volumenkoordinaten entlang dieser Achse verkleinert werden. Dadurch besitzen die Sampleabstände in Weltkoordinaten entlang jeder

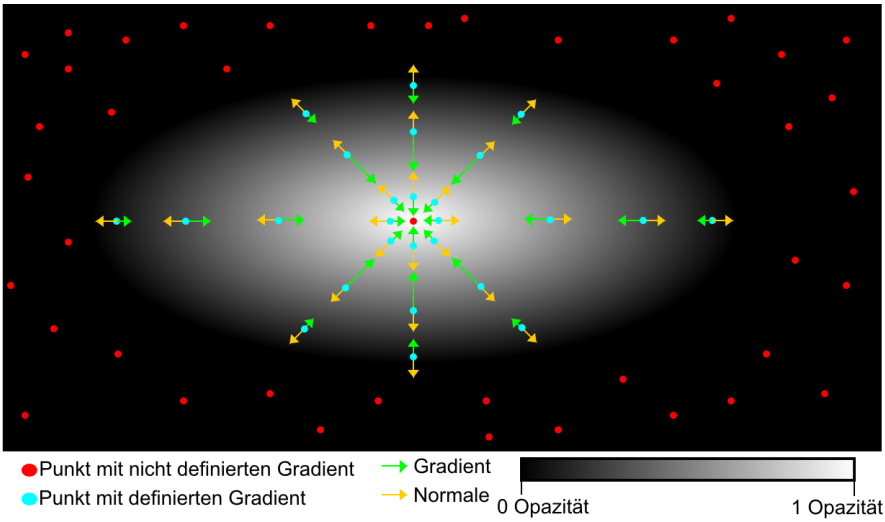


Abbildung 3.6: **Visualisierung der Gradienten und der daraus resultierenden Normalen eines elliptischen Opazitätsverlaufs.** Obwohl der Opazitätsverlauf linear ist, sind die Gradienten nahe am Rand der Ellipse oder nahe in der Mitte kleiner. Dies ist darauf zurückzuführen, dass die Gradientenfilter für die Berechnung immer eine Umgebung um den Punkt herum betrachten. An den roten Punkten ist der Gradient und damit ebenfalls die Normale nicht definiert. Dies ist sowohl am Rand des Bildes der Fall, da hier alle Werte in der Umgebung des Punktes die selbe Opazität besitzen, als auch im Ellipsenmittelpunkt, da hier der Gradient eine Polstelle besitzt.

Achse gesehen die selbe Länge. Für das Sampeln der zwischen zwei Voxeln liegenden Werte der transformierten Opazitätstextur wird dann die trilineare Interpolation der TMUs verwendet.

Man verwendet bei der Normalentextur ein vorzeichenloses Byte pro Kanal, wodurch man die Normale vor dem Abspeichern in der Textur zusätzlich noch von dem Wertebereich von minus eins bis eins in den Wertebereich von null bis 255 umrechnen muss. Auf diese Normalentextur wird dann beim Raymarching zugegriffen, um eine interpolierte Normale an der jeweiligen Strahlenposition für die Lichtberechnung zu erhalten.

Verändert sich die Transferfunktion so verändert sich der Gradient ebenfalls, wodurch die Normalentextur neu berechnet werden muss. Um dennoch eine Interaktivität beim Ändern der Transferfunktion zu gewährleisten, wurde die Berechnung der Normalentextur im praktischen Teil auf der GPU implementiert.

Generell ist die Lokalität der Normale für das verwendete Phong-Beleuchtungsmodell beim Volumenrendering ein Kompromiss. Denn ist die Normale zu lokal, so entstehen

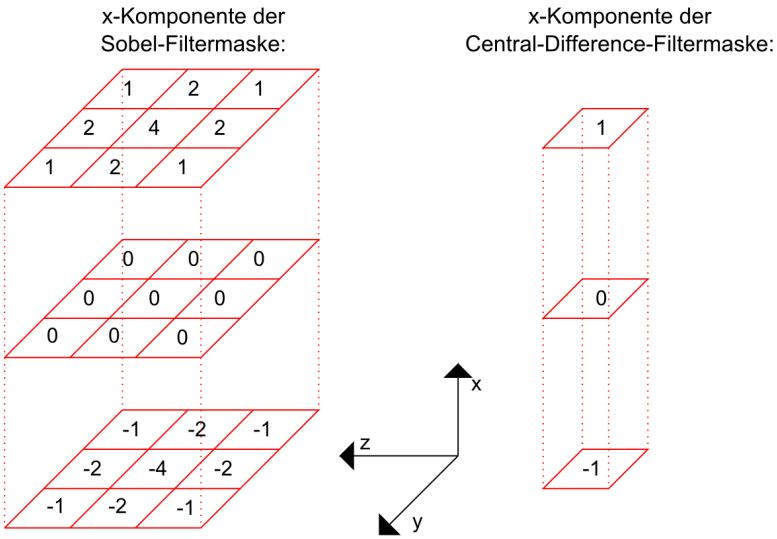


Abbildung 3.7: **Filtermasken der Gradientenfilter.** Beide sind für die x-Komponente des Gradienten. Die Werte der Sobel-Filtermaske stammen hierbei aus [RTVG06]. Die Filtermasken für die y- und z-Komponente ergeben sich durch vertauschen der Koordinatenachsen. Beide Masken sind unnormiert, da dies für die Normale wegen des Normalisierens des Gradienten keine Rolle spielt. Beim Sobel-Filter fliessen also 18 Werte der Volumendaten für jede Komponente in die Normalenberechnung mit ein. Beim Central-Difference-Filter sind dies nur zwei Werte. Dadurch besitzt der Sobel-Filter eine geringere Lokalität, ist allerdings wesentlich rechenaufwändiger.

störende harte Kanten bzw. harte Treppeneffekte entlang der Voxelebenen bei der Beleuchtungsberechnung. Ausserdem besteht durch zu hohe Lokalität ein höheres Risiko, dass der Gradient und damit die Normale an einer Stelle nicht definiert sind. Auf diese Art entstehen bei der späteren Visualisierung störende dunkle Punkte. Ist sie nicht lokal genug so liefert das Phong-Beleuchtungsmodell merklich falsche Ergebnisse. Zusätzlich wächst mit abnehmender Lokalität die Rechenzeit, da immer mehr Voxel zur Normale eines bestimmten Voxels beitragen.

Allgemein lassen sich mit dem Sobel-Filter, wegen dessen geringerer aber dennoch ausreichender Lokalität im Vergleich zum Central-Difference-Filter, bessere Ergebnisse erzielen, wie man beispielhaft in Abbildung 3.8 und Abbildung 3.9 erkennen kann. Der Sobel-Filter kostet zwar eine höhere Rechenzeit, welche aber wegen der Vorberechnung der Normale auf der GPU nicht weiter ins Gewicht fällt.

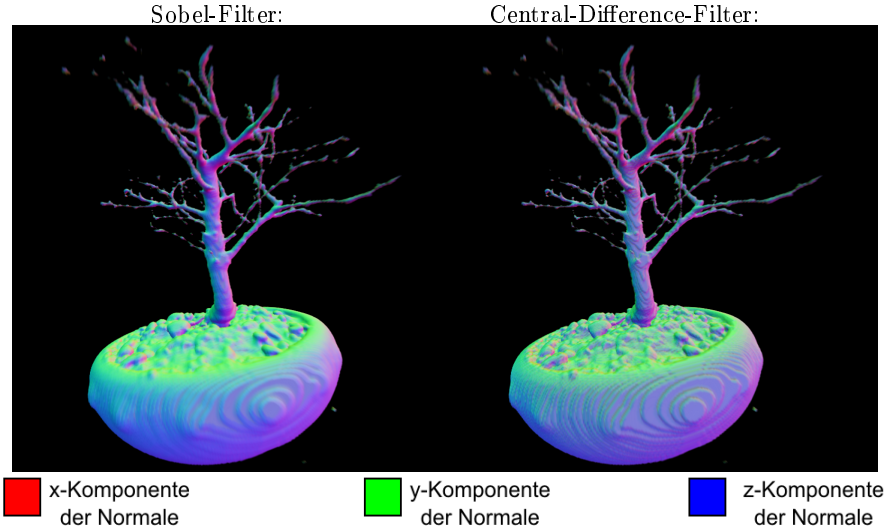


Abbildung 3.8: **Visualisierung der Normalentexturen von Central-Difference und Sobel-Filter.** Jeweils eine Normalenkomponente wurde in dieser Abbildung einem Farbkanal zugeordnet. Die Helligkeitsverläufe und Farbverläufe entstanden nur durch diese Farbzuordnung. Es fand keine Schattierung bzw. Lichtberechnung statt. Beim Central-Difference-Filter sind entlang der Voxelebenen deutliche Treppeneffekte zu erkennen. Diese Treppeneffekte verlaufen beim Sobel-Filter deutlich sanfter.

3.6 Raycasting

3.6.1 Threaderstellung

Nachdem alle für das Raycasting benötigten Daten vorberechnet worden sind, kann mit dem Raycasting selbst begonnen werden. Im ersten Schritt dieses Verfahrens wird ein Kernel mit einer globalen 2D-Range gestartet, so dass mindestens für jeden Pixel des zu rendernden Bildes ein Thread erstellt wird. Im Folgenden geht es darum die Zahl der zu startenden Workgroups bzw. die zu startende globale 2D-Range zu bestimmen. Auch gilt es hierbei eine Tessellationsregel zu definieren, wie sich die Workitems bzw. Threads und die Workgroups des gestarteten Kernels auf die Pixel des zu rendernden Bildes aufteilen. Zudem soll kurz diskutiert werden, weshalb man nicht ohne Benchmarks eine optimale Workgroupgrösse angeben kann.

Hierfür wird ein Pixelkoordinatensystem benötigt. Dies wird so gewählt, dass der Ursprung der Pixelkoordinaten in der unteren linken Bildecke liegt. Die rechte obere Bildecke habe in den Pixelkoordinaten den Wert der Auflösung $\overrightarrow{R}$ des zu rendernden Bildes und liegt damit bei $(\overrightarrow{R}.x, \overrightarrow{R}.y)$. Dies ist in Abbildung 3.10 gezeigt.

Die Tessellationsregel sollte eine einfache Kachelung von Workgroups und Warps er-

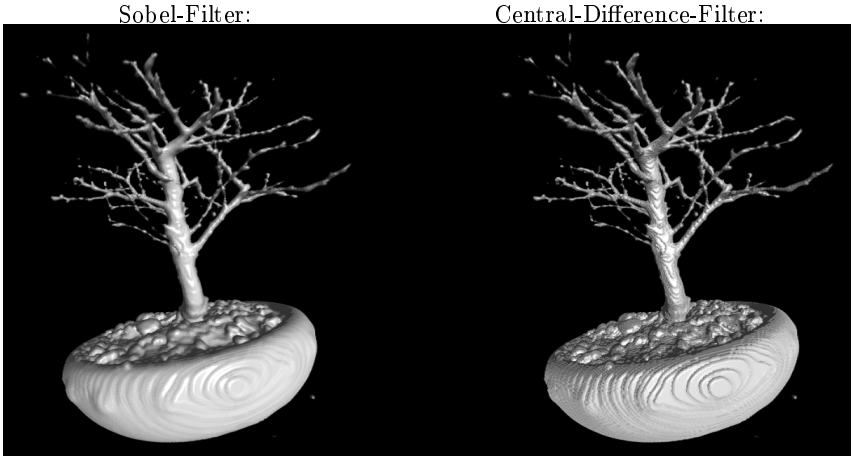


Abbildung 3.9: **Vergleich der Bildqualität zwischen Sobel-Filter und Central-Difference-Filter.** Durch das Phong-Beleuchtungsmodell schattiertes Rendering des selben Bonsais für einen weiteren Vergleich der Qualität der Normalen der beiden verschiedenen Filter. Auch nach der Beleuchtungsberechnung sind die Treppeneffekte beim Central-Difference-Filter deutlich zu erkennen. Das Bild des Sobel-Filters zeigt wieder viel sanftere Helligkeitsverläufe.

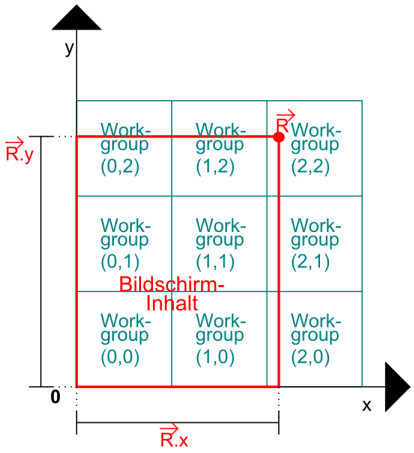


Abbildung 3.10: **Kachelung des zu rendernden Bildes durch Workgroups.** Durch die im Optimalfall quadratische Form wird die Cache-Effizienz erhöht.

lauben und gleichzeitig dafür sorgen, dass die einzelnen Strahlen der Pixel eines Warps und Strahlen der Pixel einer Workgroup möglichst lokal durch die Volumendaten verlaufen werden. Denn diese Lokalität bringt folgende Vorteile mit sich:

- Jeder Multiprozessor besitzt einen eigenen Texturcache für seine TMUs. Verlaufen nun die Strahlen einer Workgroup möglichst lokal durch die Volumendaten, so wird die Cache-Effizienz erhöht. Das selbige gilt für den L1-Cache bei der späteren Octreetraversierung.
- Durch die Lokalität eines Warps verlaufen die Threads des Warps selbst ähnlich durch die Volumendaten. Dadurch haben sie später beim Raymarching eine ähnliche Schrittzahl durch die Boundingbox der Volumendaten. Ausserdem ist die Wahrscheinlichkeit hoch, dass alle Threads bei den gleichen Raymarching Schritten nicht durch komplett transparenten Raum marschieren und deshalb alle Threads die Licht- und die Transparenzberechnung durchführen werden. Auch werden sie wahrscheinlich wegen der Early-Ray-Termination (erklärt in Punkt 3.6.5.3) bei einer ähnlichen Raymarchingschrittzahl mit dem Raymarching abbrechen. Des Weiteren werden die Threads bei der späteren Octreeoptimierung den Octree ähnlich traversieren. Dies wirkt sich alles positiv auf die SIMD-Effizienz aus.
- Durch die Lokalität der Warps innerhalb einer Workgroup besitzen diese eine ähnliche Laufzeit. So terminieren all diese Warps zu ähnlichen Zeitpunkten. Dadurch wird die Achieved-Occupancy erhöht.

Um die 32 Threads umfassenden Warps einerseits möglichst lokal und andererseits möglichst leicht kachelbar zu machen, wird eine Warpkachelgrösse $\overrightarrow{WaK}$ von (8,4) gewählt. Dadurch bedecken die Threads eines Warps stets ein Rechteck mit der Grösse von 8 auf 4 Pixeln auf dem Bildschirm. Dies wird auf dem linken Bild in Abbildung 3.11 gezeigt. Die Threads einer Workgroup sollen später ebenfalls, wie in Abbildung 3.10 gezeigt, ein Rechteck mit der Thread- bzw. Pixelkantenlänge $\overrightarrow{WGK}$ auf dem Bildschirm bedecken. Damit man die Workgroups mit diesen Warpkacheln füllen kann muss ihre Threadkantenlänge $\overrightarrow{WGK}$ entlang jeder Achse ein Vielfaches der Warpkachelgrösse $\overrightarrow{WaK}$ sein. $\overrightarrow{WGK}$ sollte hierbei zusätzlich so gewählt werden, dass das Rechteck der Workgroup idealerweise nahe an einem Quadrat ist.

Nun gilt es kurz zu begründen, weshalb man nicht ohne Benchmarks eine optimale Workgroupgrösse angeben kann. Damit die Achieved-Occupancy möglichst hoch wird, will man die Workgroups möglichst klein wählen. Denn in diesem Fall ist die Arbeit am feinsten granuliert und lässt sich am besten auf die Multiprozessoren aufteilen. Auch unterscheiden sich die Laufzeiten der Warps beim Raycaster stark. Dadurch können wenige lang laufende Warps eine Workgroup lange am Leben erhalten, wodurch die Achieved-Occupancy reduziert wird. Da diese Wahl sehr von der Graphikkarte abhängig ist, wird hier aus Gründen der Einfachheit angenommen, dass es sich um eine Geforce 580 GTX handelt, wie sie bereits in dem Abschnitt 2.2 erläutert und abgebildet worden ist. So verbraucht, sofern man die Zahl der Register nicht begrenzt, ein Thread des in diesem Kapitel erläuterten Volumenraycasters 45 Register. Diese Zahl

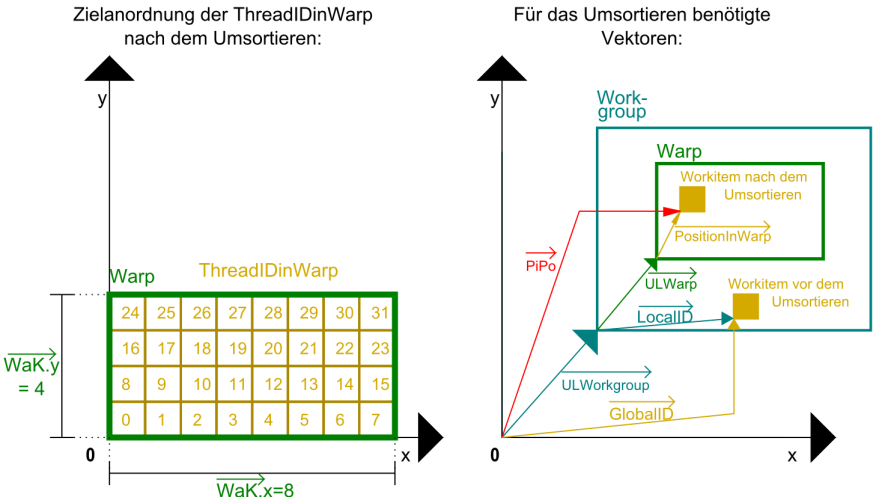


Abbildung 3.11: **Umsortieren der Workitems in der Workgroup**

ist jedoch ebenfalls vom Compiler und damit prinzipiell von der Treiberversion abhängig. Da ein Warp auf der Geforce 580 GTX immer ein Vielfaches von 64 Registern benötigt, benötigt ein Warp dieses Kernels insgesamt $\lceil 45{*}32 \rceil \, auf \, 64 = 1472$ Register. Ein Multiprozessor besitzt 32768 Register. Dadurch kann ein Multiprozessor maximal 22 Warps bei der Ausführung dieses Kernels beherbergen. Die Primfaktorzerlegung von 22 ist eins, zwei und elf. Bei einer Workgroupgrösse von einem oder zwei Warps würde man jedoch an die Hardwarelimitierung der maximal acht verwaltbaren Workgroups pro Multiprozessor stossen, und ein Multiprozessor könnte maximal 8 bzw. 16 Warps beherbergen. Deshalb kann ein Multiprozessor erst bei einer Workgroupgrösse von elf Warps die Occupancy von 22 Warps erreichen.

Hierbei steht man jedoch vor dem Problem, dass diese Achieved-Occupancy nicht nur von der Occupancy selbst, sondern auch von der Workgroupgrösse abhängig ist. Bei dieser grossen Workgroupgrösse gibt es jedoch 22 Warps, wodurch es wahrscheinlicher als bei einer kleinen Workgroupgrösse ist, dass ein sehr lang laufender Warp die Workgroup lange am Leben erhält, während alle anderen Warps bereits terminiert haben. Dies wird dadurch verstärkt, dass 11 eine Primzahl ist und deshalb eine Workgroup einem Rechteck von 1 auf 11 Warpkacheln entsprechen muss. Aus diesem Grund ist es mit einem Seitenverhältnis von 1 zu 5.5 weit von einem Quadrat entfernt. So ist es wahrscheinlicher, dass sich die Laufzeiten der Warps stark unterscheiden. Würde man jedoch auf die maximale Occupancy verzichten und eine kleine Workgroupgrösse von drei Warps wählen, so hätte man insgesamt sieben kleine Workgroups und eine Occupancy von 21 Warps. Diese kleinen Workgroups könnten dadurch, dass die Arbeit feiner granuliert ist, trotz niedrigerer Occupancy die Achieved-Occupancy und damit die Performance erhöhen.

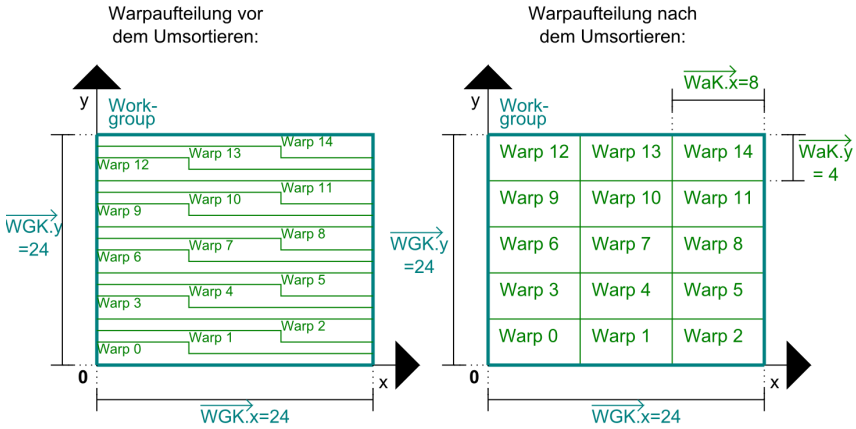


Abbildung 3.12: **Warpaufteilung innerhalb einer Workgroup vor und nach dem Umsortieren.** Durch das Umsortieren ist die Fläche, welche ein Warp auf dem Bildschirm bedeckt, wesentlich näher an einem Quadrat, wodurch die SIMD-Effizienz erhöht wird.

Deshalb lässt es sich nicht ohne Benchmark sagen, welche Workgroupgrösse bei einem gegebenen Registerverbrauch optimal ist. Zusätzlich könnte man ebenfalls versuchen, den Registerverbrauch mit Hilfe des Compilerbefehls zu begrenzen, um die Performance zu erhöhen. Ausführliche Benchmarks würden jedoch hier zu weit führen. So wird die Auswirkung der Workgroupgrösse auf die Performance erst beim Volumenraycaster mit Octree in Kapitel 6 untersucht.

Um eine Formel für die Kachelung des Bildes durch Workgroupkacheln und Warpkacheln herzuleiten, wird zuerst der Fall betrachtet, dass die Zuweisung der Threads zu der Position des zu zeichnenden Pixels $\overrightarrow{PiPo}$ über ihre ID innerhalb der globalen 2D-Range $\overrightarrow{GlobalID}$ erfolgt:

$$\overrightarrow{PiPo} = \overrightarrow{GlobalID}$$

Hier würden die Threads einer Workgroup, wie in Abbildung 3.10 gezeigt, bereits auf dem Bildschirm ein Rechteck mit der Kantenlänge $\overrightarrow{WGK}$ auffüllen. Allerdings würden die Warps, wie auf dem linken Bild der Abbildung 3.12 oder in Abbildung 2.9 gezeigt, innerhalb der Workgroup keinem 8 auf 4 Pixel grossem Rechteck auf dem Bildschirm entsprechen. Deshalb gilt es die Threads innerhalb der Workgroup umzusortieren. Die für das nun folgende Umsortieren benötigten Vektoren sind alle in Abbildung 3.11 eingezeichnet. Sei $\overrightarrow{LocalID}$ die abfragbare lokale ID des Threads innerhalb der Workgroup. So gilt für die $ThreadID$:

$$ThreadID = \overrightarrow{LocalID}.y * \overrightarrow{WGK}.x + \overrightarrow{LocalID}.x$$

Sei $WarpID$ die ID des zum Thread gehörenden Warps. So gilt für diese:

$$WarpID = \lfloor \frac{ThreadID}{32} \rfloor$$

Und für die ID des Threads innerhalb des Warps, $ThreadIDinWarp$:

$$ThreadIDinWarp = ThreadID \mod 32$$

Für die Position der linken unteren Ecke eines Warps $\overrightarrow{ULWarp}$ gilt innerhalb einer Workgroup im Falle der Kachelung:

$$\overrightarrow{ULWarp} = \begin{pmatrix} (\overrightarrow{WaK}.x * WarpID) \mod \overrightarrow{WGK}.x \\ \overrightarrow{WaK}.y * \lfloor \frac{\overrightarrow{WaK}.x * WarpID}{\overrightarrow{WGK}.x} \rfloor \end{pmatrix}$$

Für die Position $\overrightarrow{PositionInWarp}$ des Threads innerhalb des Warps muss nach dem Umsortieren in Warpkoordinaten gelten:

$$\overrightarrow{PositionInWarp} = \begin{pmatrix} ThreadIDinWarp \mod \overrightarrow{WaK}.x \\ \lfloor \frac{ThreadIDinWarp}{\overrightarrow{WaK}.x} \rfloor \end{pmatrix}$$

Sei $\overrightarrow{GroupID}$ die im Kernel abfragbare ID der zum Thread gehörigen Workgroup. Für die Position des linken unteren Pixel $\overrightarrow{ULWorkgroup}$ gilt:

$$\overrightarrow{ULWorkgroup} = \overrightarrow{WGK} * \overrightarrow{GroupID}$$

Für die Position des Pixels $\overrightarrow{PiPo}$ des Threads gilt dann:

$$\overrightarrow{PiPo} = \overrightarrow{ULWorkgroup} + \overrightarrow{ULWarp} + \overrightarrow{PositionInWarp}$$

Da in OpenCL die globale ND-Range immer einem Vielfachen der Workgroupgrösse entsprechen muss, gilt für die Zahl der zu startenden Workgroups $\overrightarrow{WStart}$:

$$\overrightarrow{Wstart} = \lceil \frac{\overrightarrow{R}}{\overrightarrow{WGK}} \rceil$$

Für die beim Kernel-Start angegebene globale 2D-Range $\overrightarrow{Global2DRange}$ gilt:

$$\overrightarrow{Global2DRange} = \overrightarrow{Wstart} * \overrightarrow{WGK}$$

Diejenigen Threads, deren Pixel ausserhalb der Auflösung $\overrightarrow{R}$ liegen, werden, nachdem sie dies abgefragt haben, sofort terminieren.

Beträgt die Kantenlänge der Workgroupgrösse entlang der x-Achse jedoch nur eine Warpkachel, so entspricht die Anordnung der Warps vor dem Umsortieren bereits der Anordnung der Warps nach dem Umsortieren. In diesem Fall ist das Umsortieren überflüssig.

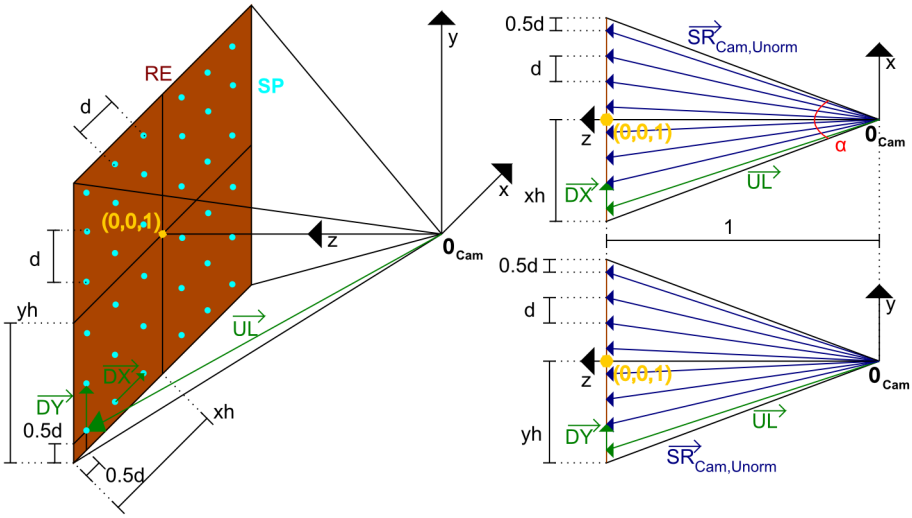


Abbildung 3.13: **Benötigte Geometrie für die Strahlerstellung.**

3.6.2 Strahlerstellung

Das Ziel dieses Punktes ist es ausgehend von einer perspektivischen Kamera für jede Pixelkoordinate eine Strahlengerade in Weltkoordinaten zu erstellen. Der erstellte Strahl $\mathbf{S}(\lambda, \overrightarrow{PiPo})$ besitzt einen Ursprung **SU**, eine Richtung $\overrightarrow{SR}$ und ein Skalar λ:

$$\mathbf{S}(\lambda, \overrightarrow{PiPo}) = \overrightarrow{SR}(\overrightarrow{PiPo}) * \lambda + \mathbf{SU}$$

Die Strahlengerade selbst beschreibt als solche eine Punktmenge. Die für die folgende Herleitung der Strahlengerade benötigte Geometrie ist in Abbildung 3.13 eingezeichnet. Die Kamera sei hierbei definiert durch ihren horizontalen Öffnungswinkel α, ihre Auflösung $\overrightarrow{R}$, und eine homogene Koordinate, welche ihre Position in Weltkoordinaten angibt. Diese homogene Koordinate setze sich zusammen aus der Drehmatrix $DrMat$ und dem Punkt **CamP**. Das Sichtvolumen einer perspektivischen Kamera ist eine Pyramide mit unendlicher Höhe. In den Kamerakoordinaten schaue die Kamera in die +z-Achse, während die +y-Achse Richtung Oberkante des Bildes zeige. Eine Eigenschaft einer perspektivischen Kamera ist, dass sich alle Strahlen der Bildpunkte der Kamera in dem Nullpunkt 0_{Cam} des Kamerakoordinatensystems schneiden. Dies entspricht dem Punkt **CamP** im Weltkoordinatensystem. Deshalb gilt für alle Strahlen:

$$\mathbf{SU} = \mathbf{CamP}$$

Des Weiteren schneiden die Strahlen jede beliebige Ebene, auf welcher die Blickrichtung, also bei der hier verwendeten Kameradefinition die z-Achse, senkrecht steht, im regelmässigen Abstand d. Diese Schnittpunkte lassen sich nun in Abhängigkeit der Koordinaten des Bildpunktes parametrisieren. Dies kann man ausnutzen um für jeden Bildpunkt die Strahlenrichtung $\overrightarrow{SR}$ zu berechnen. Der Abstand dieser Ebene zum Nullpunkt des Kamerakoordinatensystems 0_{Cam} kann hierbei willkürlich gewählt werden, ist aber in dieser Herleitung auf 1 festgelegt. Zusätzlich gilt für die Ebenengleichung dieser Ebene, dass der Aufhängepunkt $\overrightarrow{UL}$ der Schnittpunkt des Strahls der Pixelkoordinaten (0,0) mit dieser Ebene sei. $\overrightarrow{DX}$ sei der Vektor zwischen zwei horizontal benachbarten Schnittpunkten im Kamerakoordinatensystem und $\overrightarrow{DY}$ der Vektor zwischen zwei vertikal benachbarten Schnittpunkten. Somit gilt für den Schnittpunkt **SP** zwischen Ebene und dem Strahl an einer bestimmten Pixelposition $\overrightarrow{PiPo}$:

$$\mathbf{SP}_{Cam}(\overrightarrow{PiPo}) = \overrightarrow{UL}_{Cam} + \overrightarrow{DX}_{Cam} * \overrightarrow{PiPo}.x + \overrightarrow{DY}_{Cam} * \overrightarrow{PiPo}.y + \mathbf{0}_{Cam}$$

Da der Null-Punkt $\mathbf{0}_{Cam}$ ebenfalls auf dem Strahl liegt, gilt für die Richtung der Strahlengerade im Kamerakoordinatensystem:

$$\begin{aligned}\overrightarrow{SR}_{Cam,Unorm}(\overrightarrow{PiPo}) =& \mathbf{SP}(\overrightarrow{PiPo})_{Cam} - \mathbf{0}_{Cam} \\ =& \overrightarrow{UL}_{Cam} + \overrightarrow{DX}_{Cam} * \overrightarrow{PiPo}.x + \overrightarrow{DY}_{Cam} * \overrightarrow{PiPo}.y\end{aligned}$$

Deshalb gilt es die Parameter $\overrightarrow{UL}$, $\overrightarrow{DX}$ und $\overrightarrow{DY}$ ausgehend von den Konstanten der Kamera zu bestimmen. Sei RE das Rechteck, das durch den Schnitt zwischen Sichtvolumen und Ebene entsteht. So gilt für die halben Seitenlängen von RE:

$$\begin{aligned}xh =& tan\left(\frac{\alpha}{2}\right) \\ yh =& xh * \frac{\overrightarrow{R}.y}{\overrightarrow{R}.x}\end{aligned}$$

Als Nächstes soll der Abstand zwischen zwei Schnittpunkten d berechnet werden. So besitzt das Rechteck RE entlang der x-Achse $\overrightarrow{R}.x - 1$ Schnittpunkte mit den Strahlen. Zusätzlich sind der erste und der letzte Schnittpunkt entlang der x-Achse jeweils $0.5 * d$ von dem Rand des Rechtecks entfernt. Die Ursache hierfür ist, dass die Strahlen durch die Pixelmittelpunkte und nicht durch die linken untere Pixelecken verlaufen sollen. Für d gilt somit:

$$d = 2 * \frac{xh}{\overrightarrow{R}.x}$$

Dadurch kann man die Vektoren $\overrightarrow{DX}$, $\overrightarrow{DY}$ und $\overrightarrow{UL}$ im Kamerakoordinatensystem berechnen:

$$\overrightarrow{UL}_{Cam} = \begin{pmatrix} -xh + 0.5 * d \\ -yh + 0.5 * d \\ 1 \end{pmatrix} \qquad \overrightarrow{DX}_{Cam} = \begin{pmatrix} d \\ 0 \\ 0 \end{pmatrix} \qquad \overrightarrow{DY}_{Cam} = \begin{pmatrix} 0 \\ d \\ 0 \end{pmatrix}$$

Nun muss $\overrightarrow{SR}_{Cam,Unorm}$ ins Weltkoordinatensystem zu $\overrightarrow{SR}_{Unorm}$ transformiert werden:

$$\begin{aligned}\overrightarrow{SR}_{Unorm}(\overrightarrow{PiPo}) &= DrMat * \overrightarrow{SR}_{Cam,Unorm}(\overrightarrow{PiPo}) \\ &= DrMat * (\overrightarrow{UL}_{Cam} + \overrightarrow{DX}_{Cam} * \overrightarrow{PiPo}.x + \overrightarrow{DY}_{Cam} * \overrightarrow{PiPo}.y) \\ &= \overrightarrow{UL}_{Welt} + \overrightarrow{DX}_{Welt} * \overrightarrow{PiPo}.x + \overrightarrow{DY}_{Welt} * \overrightarrow{PiPo}.y\end{aligned}$$

Später wird für die Lichtberechnung der normalisierte Vektor vom Betrachter zu einem Raymarchingschrittpunkt in Weltkoordinaten benötigt. Dies entspricht für alle Punkte entlang eines Strahls der normalisierten Strahlenrichtung $\overrightarrow{SR}_{Norm}$:

$$\overrightarrow{SR}_{Norm}(\overrightarrow{PiPo}) = normalize(\overrightarrow{SR}_{Unorm}(\overrightarrow{PiPo}))$$

Zuletzt muss der Vektor $\overrightarrow{SR}_{Norm}$ für das spätere Raymarching noch auf den Abstand zwischen zwei Raymarchingschritten dz normiert werden, wodurch sich die gesuchte Strahlenrichtung $\overrightarrow{SR}$ ergibt:

$$\overrightarrow{SR}(\overrightarrow{PiPo}) = dz * \overrightarrow{SR}_{Norm}$$

3.6.3 Schnittpunktberechnung mit der Boundingbox

In diesem Kapitel werden die Schnittpunktskalare des Strahls mit der Boundingbox berechnet, um den Anfangspunkt und den Endpunkt für das spätere Raymarching zu bestimmen. Der hierfür beschriebene Algorithmus stammt aus [SO98]. Zuerst wird der Fall betrachtet, dass die Kamera ausserhalb der Boundingbox ist. Hier muss man später im Raymarching beginnend von dem Eintrittspunkt des Strahls in die Boundingbox zum Austrittspunkt marschieren. Dieser Algorithmus benötigt zuerst die Position zweier gegenüberliegender Ecken. Dafür werden die im Koordinatenursprung liegende Ecke **N** und deren gegenüberliegende Ecke **P** genommen. Diese beide Ecken sind in Abbildung 3.14 oder Abbildung 3.1 eingezeichnet. Für die beiden Ecken gilt:

$$\begin{aligned}\mathbf{P} &= (\overrightarrow{V} - 1) * \overrightarrow{v} + \mathbf{0} \\ \mathbf{N} &= \mathbf{0}\end{aligned}$$

Als Nächstes berechnet der Algorithmus die Skalare der Schnittpunkte des Strahls mit den Seitenebenen der Boundingbox. $\overrightarrow{\lambda}_1$ sei hierbei die Skalare der Schnittpunkte mit den Ebenen, die durch **N** verlaufen. $\overrightarrow{\lambda}_2$ sei die Skalare der Schnittpunkte mit den Ebenen, die durch **P** verlaufen. Dann gilt:

$$\begin{aligned}\overrightarrow{\lambda}_1 &= \frac{\mathbf{N} - \mathbf{SU}}{\overrightarrow{SR}} \\ \overrightarrow{\lambda}_2 &= \frac{\mathbf{P} - \mathbf{SU}}{\overrightarrow{SR}}\end{aligned}$$

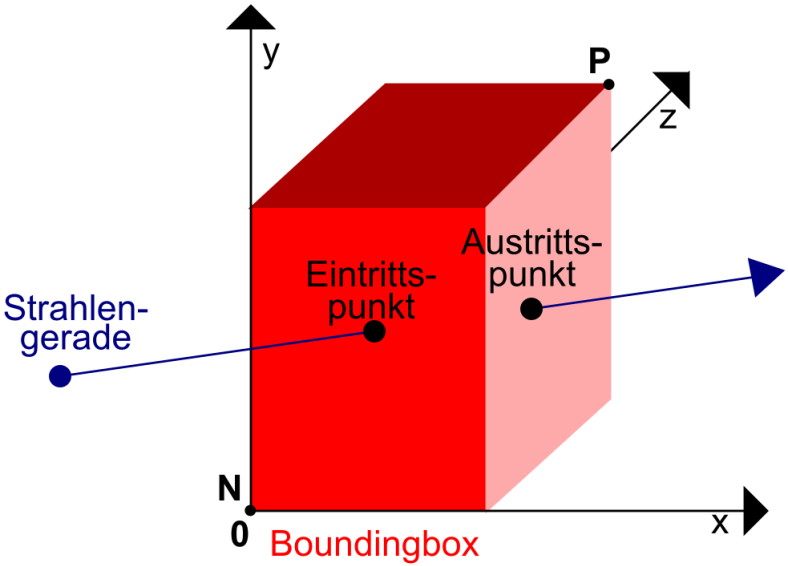


Abbildung 3.14: **Schnittpunkte der Strahlengerade mit der Boundingbox.**

Jeweils zwischen zwei gegenüberliegenden und damit parallelen Ebenen, welche durch die Seitenflächen der Boundingbox verlaufen, existiert ein unendlich grosses Volumen. So existieren bei einer Boundingbox insgesamt drei solcher Volumen. Die Schnittmenge all dieser Volumen ist die Boundingbox. Nun werden die Skalare der Eintrittspunkte und Austrittspunkte der Strahlengerade in bzw. aus diesen drei Volumen berechnet:

$$\overrightarrow{Eintrittsskalare} = min(\overrightarrow{\lambda}_1, \overrightarrow{\lambda}_2)$$
$$\overrightarrow{Austrittsskalare} = max(\overrightarrow{\lambda}_1, \overrightarrow{\lambda}_2)$$

Der Eintrittspunkt in die Boundingbox muss nun im Falle eines Schnitts bei dem grössten aller Eintrittsskalare sein. Denn ab hier befindet sich der Strahl innerhalb jeder dieser drei Volumen und hat damit die Boundingbox betreten. Der Austrittspunkt muss bei dem kleinsten aller Austrittsskalare, liegen, da hier der Strahl das erste dieser Volumen verlässt. Dies bedeutet gleichzeitig, dass er dort ebenfalls die Boundingbox verlässt:

$$Eintrittsskalar = max(\overrightarrow{Eintrittsskalare})$$
$$Austrittsskalar = min(\overrightarrow{Austrittsskalare})$$

Ist nun das berechnete Eintrittsskalar grösser als das berechnete Austrittsskalar, so verlässt der Strahl das erste dieser Volumen bereits, bevor er das letzte Volumen

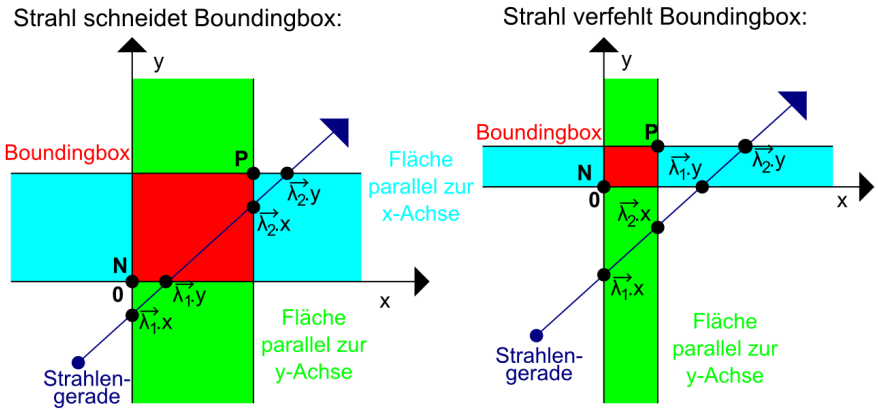


Abbildung 3.15: **Beispiel für die Schnittpunktberechnung mit der Boundingbox.**

betreten hat. Hierdurch verfehlt der Strahl die Boundingbox. Hier gilt also:

$$Eintrittsskalar > Austrittsskalar$$

Schneidet er jedoch die Boundingbox so hat der Strahl alle dieser Volumen betreten, bevor er eines der Volumen wieder verlassen hat. Deshalb gilt in diesem Fall:

$$Eintrittsskalar <= Austrittsskalar$$

In dem Fall, dass der Strahl die Boundingbox verfehlt, kann der Thread hier sofort die Farbe des Hintergrunds an den entsprechenden Pixel zeichnen und terminieren.

Als Nächstes soll eine beispielhafte Erläuterung des verwendeten Algorithmus für den zweidimensionalen Fall vorgenommen werden. Dieser zweidimensionale Fall verläuft, abgesehen davon, dass die Volumen zu Flächen und die Ebenen zu Geraden werden, analog zum dreidimensionalen Fall. So ist die Boundingbox nun ein Rechteck, welches vier Seiten besitzt. Durch jede der Seiten verläuft eine Gerade. Jeweils zwei dieser Geraden verlaufen parallel, wodurch zwischen ihnen eine unendlich grosse Fläche liegt. Die Schnittmenge all dieser Flächen entspricht wieder der Boundingbox. Bei dem nun erläuterten Beispiel soll die Strahlengerade so verlaufen wie es in Abbildung 3.15 gezeigt ist. Auf dem linken Bild in Abbildung 3.15 gilt:

$$\overrightarrow{\lambda}_1.x < \overrightarrow{\lambda}_1.y < \overrightarrow{\lambda}_2.x < \overrightarrow{\lambda}_2.y$$

Daraus ergibt sich Folgendes:

- Die Eintrittsskalare sind die kleinsten der beiden x-Komponenten und die kleinsten der beiden y-Komponenten von $\overrightarrow{\lambda}_1$ und $\overrightarrow{\lambda}_2$ also $\overrightarrow{\lambda}_1.x$ und $\overrightarrow{\lambda}_1.y$.

- Die Austrittsskalare sind die grössten der beiden x-Komponenten und die grössten der beiden y-Komponenten von $\overrightarrow{\lambda}_1$ und $\overrightarrow{\lambda}_2$ also $\overrightarrow{\lambda}_2.x$ und $\overrightarrow{\lambda}_2.y$.
- Das Eintrittsskalar in die Boundingbox ist das grösste aller Eintrittsskalare also $\overrightarrow{\lambda}_1.y$.
- Das Austrittsskalar aus der Boundingbox ist das kleinste aller Austrittsskalare also $\overrightarrow{\lambda}_2.x$.
- Da gilt $\overrightarrow{\lambda}_1.y < \overrightarrow{\lambda}_2.x$ schneidet der Strahl die Boundingbox.

Auf dem rechten Bild in Abbildung 3.15 gilt:
$\overrightarrow{\lambda}_1.x < \overrightarrow{\lambda}_2.x < \overrightarrow{\lambda}_1.y < \overrightarrow{\lambda}_2.y$
Daraus ergibt sich Folgendes:

- Die Eintrittsskalare sind die kleinsten der beiden x-Komponenten und die kleinsten der beiden y-Komponenten von $\overrightarrow{\lambda}_1$ und $\overrightarrow{\lambda}_2$ also $\overrightarrow{\lambda}_1.x$ und $\overrightarrow{\lambda}_1.y$.
- Die Austrittsskalare sind die grössten der beiden x-Komponenten und die grössten der beiden y-Komponenten von $\overrightarrow{\lambda}_1$ und $\overrightarrow{\lambda}_2$ also $\overrightarrow{\lambda}_2.x$ und $\overrightarrow{\lambda}_2.y$.
- Das Eintrittsskalar in die Boundingbox wäre das grösste aller Eintrittsskalare also $\overrightarrow{\lambda}_1.y$.
- Das Austrittsskalar aus der Boundingbox wäre das kleinste aller Austrittsskalare also $\overrightarrow{\lambda}_2.x$.
- Da gilt $\overrightarrow{\lambda}_1.y > \overrightarrow{\lambda}_2.x$ verfehlt der Strahl die Boundingbox.

Nun wird der Fall betrachtet, dass die Kamera innerhalb der Boundingbox ist. Dadurch kann der Strahl die Boundingbox nicht verfehlen. Da man hier ab dem Strahlenursprung beim Raymarching durch das Volumen marschiert, muss nur noch das Austrittsskalar berechnet werden. Dieses kann analog mit dem soeben erläuterten Algorithmus durchgeführt werden. Die Teile des Verfahrens, welche das Eintrittsskalar berechnen oder berechnen ob der Strahl die Boundingbox überhaupt schneidet, können dementsprechend weggelassen werden.

3.6.4 Transformation des Strahls in Normalisierte Texturkoordinaten

Um die TMUs anzusteuern wird der Strahl in normalisierte Texturkoordinaten, welche in 3.16 gezeigt sind, umgewandelt. Zur einfacheren Herleitung wird der Strahl zuerst in normalisierte Volumenkoordinaten (siehe Anmerkung 3.1) transformiert:

$$\mathbf{S}_{NormVolumen} = \frac{\mathbf{S}}{\mathbf{P}_{Welt}}$$

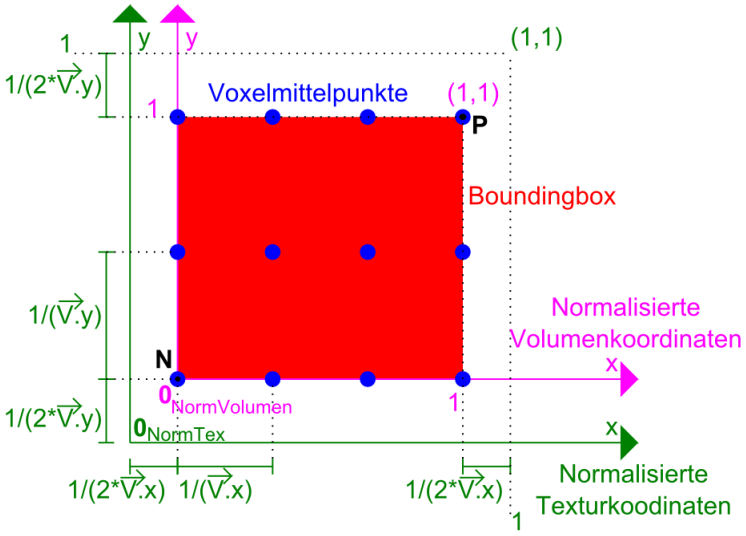


Abbildung 3.16: **Illustration des normalisierten Volumenkoordinatensystems und des normalisierten Texturkoordinatensystems.**

In diesen normalisierten Volumenkoordinaten liegt das Zentrum des ersten Voxels $\mathbf{N}_{NormVolumen}$ immer noch bei (0,0,0) und das Zentrum des äussersten Voxels $\mathbf{P}_{NormVolumen}$ bei (1,1,1). Bei den normalisierten Texturkoordinaten befindet sich allerdings das Zentrum des ersten Texels bzw. Voxels bei $\frac{1}{2*\vec{V}}$ und das Zentrum des äussersten Texels bzw. Voxels befindet sich ebenfalls nicht bei (1,1,1) sondern bei $1 - \frac{1}{2*\vec{V}}$.

Da die Achsen des normalisierten Textur- und des normalisierten Volumenkoordinatensystems parallel sind, muss man lediglich den ins normalisierte Texturkoordinatensystem zu transformierenden Punkt zuerst mit einem Vektor $\vec{s}$ skalieren und danach mit einem Translationsvektor $\vec{d}$ verschieben. Diese beiden Transformationsparameter gilt es nun zu bestimmen. Für die Zentren $\mathbf{N}$ des ersten und $\mathbf{P}$ des letzten Voxel gelten:

$$\vec{s} * \mathbf{N}_{NormVolumen} + \vec{d} = \mathbf{N}_{NormTex}$$
$$\vec{s} * \mathbf{P}_{NormVolumen} + \vec{d} = \mathbf{P}_{NormTex}$$

Die beiden Werte für $N_{NormVolumen}$ und $P_{NormVolumen}$ eingesetzt ergibt:

$$\begin{aligned}\overrightarrow{s} * 0 + \overrightarrow{d} &= \frac{1}{2 * \overrightarrow{V}} \\ \overrightarrow{s} * 1 + \overrightarrow{d} &= 1 - \frac{1}{2 * \overrightarrow{V}}\end{aligned}$$

Daraus folgt für $\overrightarrow{d}$:

$$\overrightarrow{d} = \frac{1}{2 * \overrightarrow{V}}$$

Nun kann man $\overrightarrow{d}$ einsetzen und $\overrightarrow{s}$ berechnen:

$$\begin{aligned}\overrightarrow{s} * 1 + \frac{1}{2 * \overrightarrow{V}} &= 1 - \frac{1}{2 * \overrightarrow{V}} \\ \overrightarrow{s} &= 1 - \frac{1}{\overrightarrow{V}}\end{aligned}$$

Dementsprechend gilt für den Ursprung des Strahls **SU** im normalisierten Texturkoordinatensystem:

$$\mathbf{SU}_{NormTex} = \overrightarrow{s} * \frac{\mathbf{SU}}{\mathbf{P}} + \overrightarrow{d}$$

Da die Strahlenrichtung kein Punkt sondern ein Vektor ist, darf man diese nicht durch $\overrightarrow{d}$ verschieben sondern nur mit $\overrightarrow{s}$ strecken:

$$\overrightarrow{SR}_{NormTex} = \overrightarrow{s} * \frac{\overrightarrow{SR}}{\mathbf{P}}$$

Für den Strahl **S** gilt somit in den normalisierten Texturkoordinaten:

$$\mathbf{S}_{normTex}(\lambda) = \mathbf{SU}_{NormTex} + \lambda * \overrightarrow{SR}_{NormTex}$$

Anmerkung 3.1: Für dieses grundlegende Verfahren ist das Alignment der Voxelmittelpunkte in der Boundingbox nicht sonderlich wichtig. So kann man die normalisierten Texturkoordinaten mit den normalisierten Volumenkoordinaten approximieren, ohne beim Rendern merklich falsche Ergebnisse zu erhalten. Dies ändert sich allerdings sobald man den Octree zur Optimierung heranzieht. Denn bei diesem müssen die Octreeknoten exakt an den Interpolationsgrenzen ausgerichtet werden um optimale Performance zu erzielen. Verwendet man dort eine Approximation für die Texturkoordinaten, entstehen störende Bildfehler an den Knotengrenzen. Da man bei der Transferfunktion eine Lookuptexture verwendet, müsste man eigentlich beim Nachschlagen der Opazitätswerte in dieser LUT immer die gleiche in diesem Punkt beschriebene Transformation durchführen. Mit der Näherung lassen sich jedoch bei der Transferfunktion bereits befriedigende Ergebnisse erzielen. Hier ist es einmal wieder besonders frustrierend, dass Cuda eigentlich "Lookuptable"-Koordinaten bei der TMU-Ansteuerung unterstützt. Diese entsprechen den normalisierten Volumenkoordinaten, wodurch diese Transformationen entfallen würden.

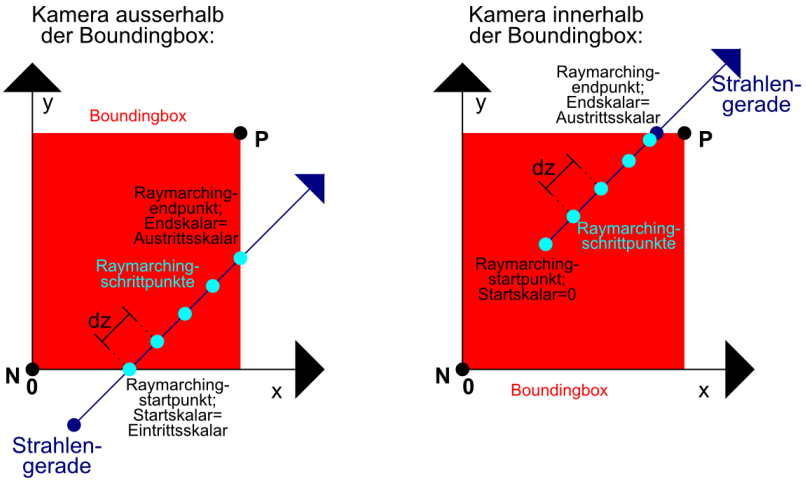


Abbildung 3.17: **Visualisierung des Raymarchings.**

3.6.5 Raymarching

3.6.5.1 Allgemeines

Beim Raymarching wird nun entlang der Strahlengerade in regelmässigen Schritten durch die Volumendaten, wie in Abbildung 3.17 gezeigt wird, marschiert. Das iterative Raymarchingverfahren, welches in diesem Punkt erläutert wird, ist in Abbildung 3.18 in Pseudocode dargestellt. Die einzelnen Raymarchingschrittpunkte werden dabei jeweils durch ein Raymarchingskalar und die Strahlengerade beschrieben. Bei jeder Raymarchingiteration wird dieses Raymarchingskalar um eins erhöht. Dadurch entspricht der Abstand zwischen zwei Raymarchingschritten durch die zuvor erfolgte Normierung des Strahles auf dz einer bestimmten frei wählbaren Streckenlänge dz in Weltkoordinaten. Da man nur diejenigen Volumendaten zeichnen darf, welche im Sichtvolumen der Kamera liegen, muss man in dem Fall, dass die Kamera sich in der Boundingbox befindet, ab dem Strahlenursprung losmarschieren. Der Startwert für das Raymarchingskalar ist somit null. Befindet sich die Kamera ausserhalb der Boundingbox, so liegen die gesamten Volumendaten entlang des Strahls im Sichtvolumen. Der Startwert für das Raymarchingskalar ist deshalb das Eintrittsskalar des Strahls in die Boundingbox.

Bei jedem Raymarchingschritt finden folgende Unterpunkte, welche ebenfalls in Abbildung 3.19 gezeigt werden, statt:

- **Schrittpunktberechnung:** Zuerst berechnet man bei jedem Schritt die jeweilige Schrittposition entlang des Strahls in normalisierten Texturkoordinaten.
- **Sampling der Texturen:** Danach samplet man an der Schrittposition die

```
for(Raymarchingskalar = Startskalar; Raymarchingskalar <
    Austrittsskalar; Raymarchingskalar++)
{
  SchrittTex = StrahlTex(Raymarchingskalar);
  Opazität = NehmeSample(TransformierteOpazitätstextur, SchrittTex);
  if(Opazität > 0)
  {
    Normalentextursample = NehmeSample(NormalenTextur, SchrittTex);
    Farbe = NehmeSample(Farbtextur, SchrittTex);
    PhongIntensität = PhongLichtberechnung(Normalentextursample, Farbe);
    A-Compositing(Opazität, PhongIntensität);
  }
}
```

Abbildung 3.18: **Pseudocode des Raymarchings.**

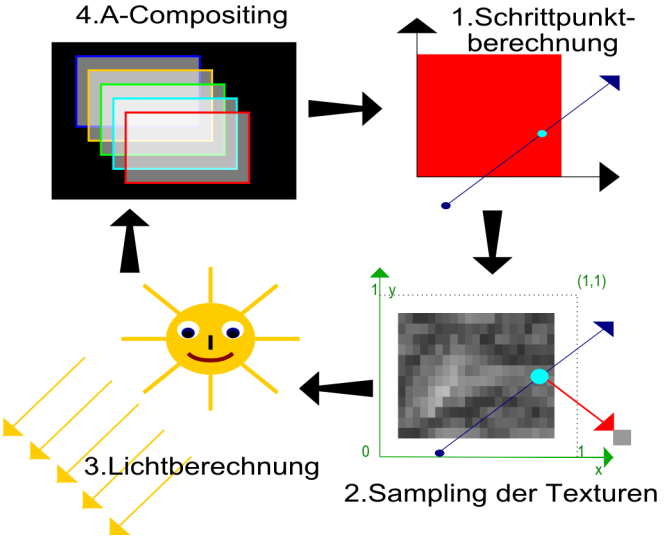


Abbildung 3.19: **Unterpunkte beim Raymarching.**

durch die Pre-Classified Transferfunktion transformierte Opazitätstextur. Ist das Sample null, so ist der Punkt komplett transparent, wodurch der Punkt nichts zur Farbe des Pixels beitragen kann. Deshalb kann in diesem Fall die Lichtberechnung und das A-Compositing übersprungen werden. Dadurch wird es wesentlich billiger durch den leeren Raum innerhalb der Volumendaten zu marschieren, als durch den vollen Raum. Als Nächstes samplet man die Farbtextur und die Normalentextur an der selben Stelle. Das Sampling der Texturen erfolgt beim Raymarching mit Hilfe der TMUs. Hierfür wird die trilineare Interpolation verwendet, wodurch die Bildqualität im Vergleich zur Nearest-Neighbor-Interpolation, wie in Abbildung 3.20 gezeigt wird, erhöht wird. Auch kann man dadurch, dass man ebenfalls die TMUs für Berechnungen heranzieht, die Performance des Raycasters stark erhöhen.

- **Phong-Lichtberechnung:** Danach findet die Lichtberechnung mit Hilfe des Phong-Beleuchtungsmodells, dem Normalentextursample und dem Farbtextursample statt.
- **A-Compositing:** Zuletzt wird bei jedem Raymarchingschritt eine Transparenzberechnung mit Hilfe des A-Compositings ausgeführt.

Ist das Traversierungsskalar grösser als das Austrittsskalar aus der Boundingbox, so hat man die Volumendaten verlassen und kann das Raymarching beenden. Abschliessend wird noch einmal das A-Compositing ausgeführt um die beim Raymarching berechnete Farbe mit der Hintergrundfarbe zu vermischen. Danach wird das Ergebnis in den Framebuffer geschrieben.

3.6.5.2 Schattierung durch das Phong-Beleuchtungsmodell

Zum Schattieren der Volumendaten soll hier das Phong-Beleuchtungsmodell mit einer parallelen Lichtquelle verwendet werden. Das Phong-Beleuchtungsmodell ist eigentlich ein Modell für diskrete Oberflächen, wird allerdings oft zum Volumenrendering verwendet. Teile der folgenden Erklärungen sind an [WPb] angelehnt. Dieses Phong-Beleuchtungsmodell gibt an, welche Intensität $\overrightarrow{I}_{Phong}$ ein gegebener Punkt auf einer Objektoberfläche mit einem bestimmten Material auf Grund einer Lichtquelle in Richtung Kamera reflektiert. Für das Modell benötigt man zuerst eine Normale $\overrightarrow{No}$. Für diese wird beim Raymarching die vorberechnete Normalentextur an der entsprechenden Schrittposition in normalisierten Texturkoordinaten $\mathbf{Schritt}_{Tex}$ gesamplet:

$$\overrightarrow{No}_{Sample} = NehmeSample(NormalenTextur, \mathbf{Schritt}_{Tex})$$

Die TMUs liefern beim Samplen einer Byte-Textur einen interpolierten Wert im Wertebereich von null bis eins zurück. Bei dem Interpolieren wird somit der Byte-Wertebereich von null bis 255 als Wertebereich von null bis eins interpretiert. Deshalb muss man die Normale auf ihren eigentlichen Wertebereich wie folgt umrechnen:

$$\overrightarrow{No}_{Unorm} = (2 * \overrightarrow{No}_{Sample}) - 1$$

Nearest-Neighbor-Interpolation: Trilineare Interpolation:

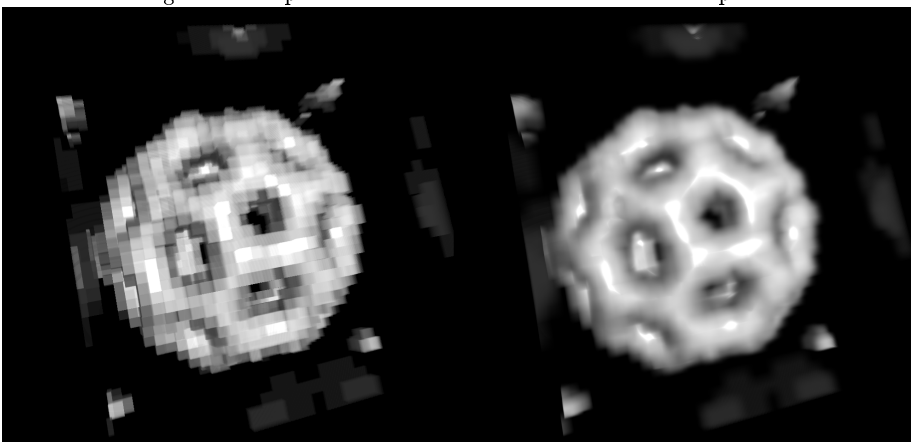


Abbildung 3.20: **Vergleich der Bildqualität zwischen Nearest-Neighbor-Interpolation und trilinearer Interpolation.**

Da der Betrag dieser gesampleten Normale durch die trilineare Interpolation deutlich von eins abweichen kann, muss die Normale für die Lichtberechnung zuerst normalisiert werden:

$$\overrightarrow{No} = normalize(\overrightarrow{No}_{Unorm})$$

Auch benötigt man um das Modell einzufärben eine Farbe $\overrightarrow{Fa}$. Diese ist ein Vektor im RBG-Raum und entsteht beim Raymarching durch das Samplen der vorberechneten Farbtextur an der selben Stelle:

$$\overrightarrow{Fa} = NehmeSample(Farbtextur, \mathbf{Schritt}_{Tex})$$

Ebenfalls wird der normalisierte Vektor vom Betrachter zum Raymarchingschrittpunkt in Weltkoordinaten benötigt. Dies entspricht dem bereits bei der Strahlenerstellung berechneten Vektor $\overrightarrow{SR}_{Norm}$. Die nun folgende Lichtberechnung findet in Weltkoordinaten statt. Die Normale $\overrightarrow{No}$, die Lichtrichtung $\overrightarrow{LR}$ und der Vektor $\overrightarrow{SR}_{Norm}$ liegen dementsprechend alle in Weltkoordinaten vor. Die Formel für die gemäss des Phong-Beleuchtungsmodells von einem Punkt reflektierte Intensität $\overrightarrow{I}_{Phong}$ in Richtung der Kamera lautet:

$$\overrightarrow{I}_{Phong} = \overrightarrow{Fa} * (\overrightarrow{I}_{Ambient} + \overrightarrow{I}_{Diffus} + \overrightarrow{I}_{Specular})$$

Der Farbfaktor $\overrightarrow{Fa}$ ist hierbei kein eigentlicher Bestandteil des Beleuchtungsmodells. Jedoch wird er wie hier oft verwendet um einen weiteren Freiheitsgrad zum Einfärben

$\overrightarrow{I}_{Ambient}$: $\overrightarrow{I}_{Diffus}$: $\overrightarrow{I}_{Specular}$: $\overrightarrow{I}_{Phong}$:

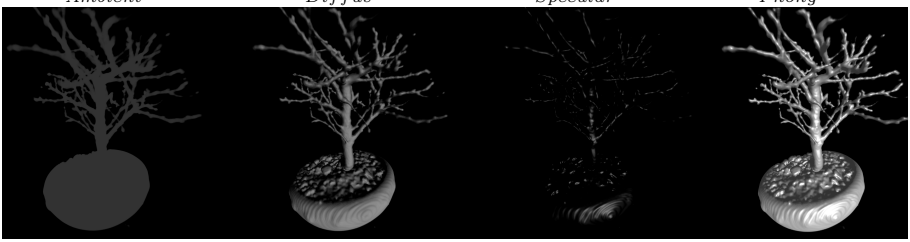


Abbildung 3.21: **Die verschiedenen Intensitäten des Phong-Beleuchtungsmodells.**

eines Objekts zu erzeugen. Die verschiedenen Intensitäten des Beleuchtungsmodells werden in Abbildung 3.21 gezeigt.

Für jede der verschiedenen Intensitäten besitzt das Licht eine eigene Intensitätskonstante und das Material eine eigene Materialkonstante. Diese beschreiben, wie gut das Material die entsprechende Lichtintensität reflektiert. Da man bei diesem Volumenraycaster nur ein Licht und ein Material verwendet, kann man das Produkt von Lichtintensitätskonstante und Materialkonstante bei der Berechnung überall durch eine einzige Lichtmaterialkonstante $\overrightarrow{LMat}$ ersetzen. Theoretisch wäre es jedoch möglich die Transferfunktion so zu erweitern, dass sie ebenfalls Materialien erzeugen würde. Auch wäre eine aufwändigere Beleuchtung mit mehreren Lichtquellen denkbar. In diesen Fällen könnte man dann diese Vereinfachung nicht mehr durchführen. Die verschiedenen Intensitäten $\overrightarrow{I}$ und die Lichtmaterialkonstanten $\overrightarrow{LMat}$ sind hier ebenfalls Vektoren im RGB-Raum.

Die Ambientintensität $\overrightarrow{I}_{Ambient}$ simuliert beim Phong-Beleuchtungsmodell den Anteil des Umgebungslichts, welches isotrop aus allen Richtungen auf das Objekt eintrifft und ebenfalls isotrop in alle Richtungen reflektiert wird. Dadurch ist sie von der Betrachterposition und der Position der Lichtquelle unabhängig. Sie berechnet sich deshalb aus:

$$\overrightarrow{I}_{Ambient} = \overrightarrow{LMat}_{Ambient}$$

Die Diffusintensität $\overrightarrow{I}_{Diffus}$ simuliert die ungerichtete Reflexion des direkt von der Lichtquelle eintreffenden Lichtes gemäss des Lambertschen Gesetzes, wodurch die vom Betrachter wahrgenommene Intensität dieses Anteils von der Betrachterposition unabhängig ist. Sie ist jedoch linear von der Beleuchtungsstärke, also derjenigen Intensität, welche pro Flächeneinheit auf der Oberfläche eintrifft, abhängig. Die für die folgende Herleitung der Diffusintensität benötigte Geometrie ist in Abbildung 3.22 dargestellt. So ergibt sich die Diffusintensität aus dem Winkel γ zwischen Normale $\overrightarrow{No}$ und Lichtrichtung $\overrightarrow{LR}$:

$$\overrightarrow{I}_{Diffus} \sim cos(\gamma)$$

Da $\overrightarrow{LR}$ und die $\overrightarrow{No}$ normalisiert sind gilt:

$$\overrightarrow{I}_{Diffus} \sim dot(\overrightarrow{LR}, \overrightarrow{No})$$

Die Formel behandelt allerdings nicht den Fall, dass das Licht von der der Normalen gegenüberliegenden Seite auf die Oberfläche eintrifft. Dieser Fall entspricht allerdings der Transmission, weshalb sich dies nicht mit Hilfe der Reflexion beschreiben lässt und die Diffusintensität null ist. Somit gilt für die gesamte Diffusintensität:

$$\overrightarrow{I}_{Diffus} = \overrightarrow{LMat}_{Diffus} * max(0, dot(\overrightarrow{LR}, \overrightarrow{No}))$$

Die Specularintensität $\overrightarrow{I}_{Specular}$ ist rein empirischer Natur. Diese tritt ebenfalls nur auf, wenn das Licht auf die Vorderseite der Oberfläche eintrifft. Die für die Herleitung der Specularintensität benötigte Geometrie ist wieder in Abbildung 3.22 dargestellt. Die Specularintensität beschreibt die empirische Beobachtung, dass Materialien das eintreffende Licht bevorzugt um einen gewissen Winkel β um die ideale Reflexionsrichtung $\overrightarrow{Ref}$ reflektieren:

$$\begin{aligned}\overrightarrow{Ref} =& Reflect(\overrightarrow{No}, \overrightarrow{LR}) \\ =& \overrightarrow{LR} - 2 * dot(\overrightarrow{No}, \overrightarrow{LR}) * \overrightarrow{No}\end{aligned}$$

Je näher der Betrachter sich an dieser idealen Reflexionsrichtung $\overrightarrow{Ref}$ befindet bzw. je kleiner β ist, desto höher wird die wahrgenommene Specularintensität. Bei einem Winkel von 90 Grad oder höher zwischen Blickrichtung und Reflexionsrichtung sei die Intensität null, während sie bei null Grad ihr Maximum erreicht. Dies wird über den Cosinus des Winkels β simuliert. Dieser lässt sich wieder über das Skalarprodukt zwischen $\overrightarrow{Ref}$ und $\overrightarrow{SR}_{Norm}$ ausdrücken:

$$\begin{aligned}Basis =& max(0, cos(\beta)) \\ =& max(0, dot(\overrightarrow{Ref}, \overrightarrow{SR}_{Norm}))\end{aligned}$$

Zu guter Letzt gilt es noch einen stetiges Schrumpfen der $Basis$ innerhalb des Wertebereichs von null bis eins zu modellieren. Dies wird über die Potenzfunktion erzielt:

$$Specularfaktor = Basis^{Shininess}$$

Der Exponent Shininess beschreibt hierbei, wie gut ein Material die gerichtete Reflexion ausführt. Ist er gross, so ist das Material ein guter gerichteter Reflektor und reflektiert das Licht nur um einen geringen Winkel um die ideale Reflexionsrichtung. Ist er klein, so ist es ein schlechter gerichteter Reflektor und reflektiert das Licht um einen großen Winkel um die ideale Reflexionsrichtung. Für die gesamte Specularintensität gilt:

$$\overrightarrow{I}_{Specular} = \overrightarrow{LMat}_{Specular} * Specularfaktor$$

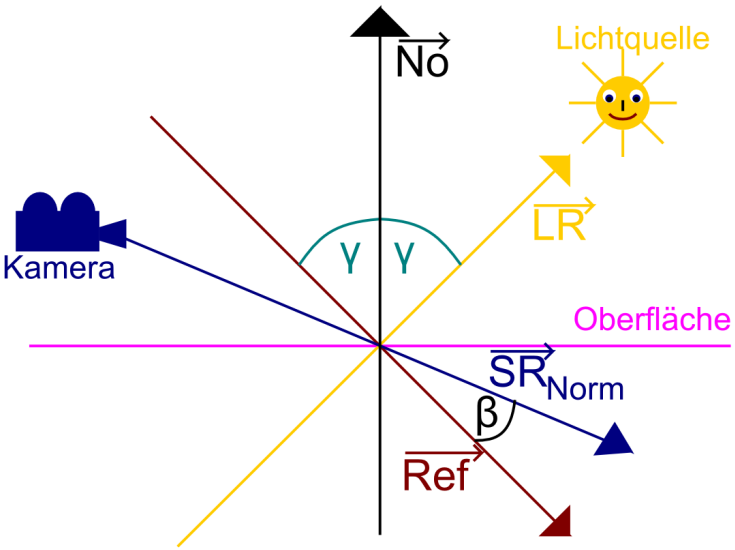


Abbildung 3.22: **Für die Phong-Beleuchtungsmodell benötigte Geometrie.**

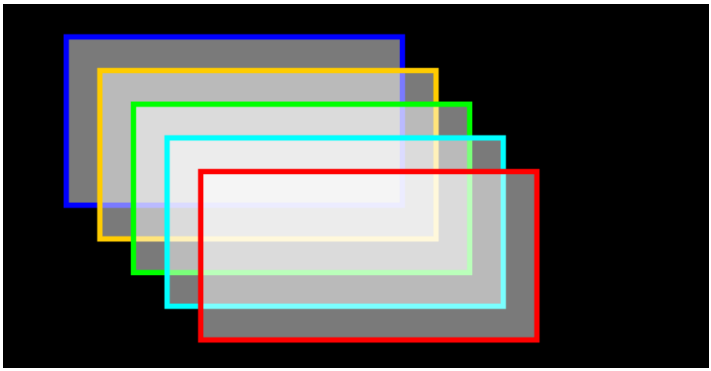


Abbildung 3.23: **A-Compositing mehrerer transparenter Rechtecke.** Deutlich zu erkennen ist, dass je mehr andere Rechtecke vor einem bestimmten Rechteck liegen, umso weniger trägt dieses Rechteck zur Farbe eines Pixels bei.

3.6.5.3 A-Compositing

Das Compositing beschreibt beim Raymarching, wie sich die Farbe des zum Strahl gehörenden Pixels durch die verschiedenen gestreuten Intensitäten und Opazitätswerte, welche an den jeweiligen Schrittpunkten vorherrschen, berechnet. Das A-Compositing ist ein photorealistisches Compositing-Modell, welches auf den physikalischen Gesetzen der Absorbanz bzw. Absorption beruht (siehe Anmerkung 3.2). Dieses Modell ist ebenfalls das Standardmodell um in der Computergraphik Transparenz zu erzeugen. Eine einfache Visualisierung des A-Compositings wird in Abbildung 3.23 gezeigt. Eine einfache mögliche Herleitung für das A-Compositing basiert darauf, dass das Volumenbild durch die Punkte der Raymarchingschritte entlang des Strahls approximiert wird. Zusätzlich wird angenommen, dass jeder dieser Punkte die Intensität des eintreffenden Lichtes $\overrightarrow{I}_{eintreffend}$ durch seine Opazität Op beim Passieren reduziert. Dieser Vorgang wird Absorption genannt. Für die absorbierte Intensität $\overrightarrow{I}_{Absorption}$ gelte:

$$\overrightarrow{I}_{Absorption} = \overrightarrow{I}_{eintreffend} * Op$$

Von dem absorbierten Licht wird wiederum ein Teil gemäss eines Streuungsfaktors $\overrightarrow{StF}$ in Richtung der Kamera gestreut. Für diese zur Kamera gestreute $\overrightarrow{I}_{Streuung}$ gelte:

$$\overrightarrow{I}_{Streuung} = \overrightarrow{StF} * \overrightarrow{I}_{eintreffend} * Op$$

Ausserdem wird das Gegenteil zur Absorption als die Transmission definiert. Für die transmittierte Intensität $\overrightarrow{I}_{Transmission}$ gelte:

$$\overrightarrow{I}_{Transmission} = \overrightarrow{I}_{eintreffend} * (1 - Op)$$

Als Vereinfachung soll sich die Absorption eines Punktes nur auf das bereits gestreute Licht auswirken. Dadurch verliert das von der Lichtquelle kommende Licht, trotz auftretender Streuung, beim Passieren der Volumendaten keinerlei Intensität. Ebenso soll aus Gründen der Einfachheit das Licht der Lichtquelle nur einmal gemäss des Phong-Beleuchtungsmodells reflektiert bzw. gestreut werden. Mehrfachstreuung wird ausgeschlossen. Deshalb setzt man für die gestreute Intensität an:

$$\overrightarrow{I}_{Streuung} = \overrightarrow{I}_{Phong} * Op$$

Das reflektierte Licht eines Punktes wird demnach auf dem Weg zur Kamera von den anderen Punkten, welche sich entlang des Strahls vor diesen Punkt befinden, zum Teil absorbiert. Diese Gegebenheiten und Vereinfachungen werden in Abbildung 3.24 dargestellt. Sei n der Punkt beim $n$ten Raymarchingschritt. So liegen die Punkte der Schritte 0 bis $n-1$ entlang des Strahls vor diesem Punkt. Der Transmissionsgrad Tr_n beim Punkt n verursacht durch die Punkte 0 bis $n-1$ berechnet sich aus:

$$\begin{aligned} Tr_n =& (1 - Op_0) * (1 - Op_1) * (1 - Op_2) * \cdots * (1 - Op_{n-1}) \\ =& \prod_{i=0}^{n-1} (1 - Op_i) \end{aligned}$$

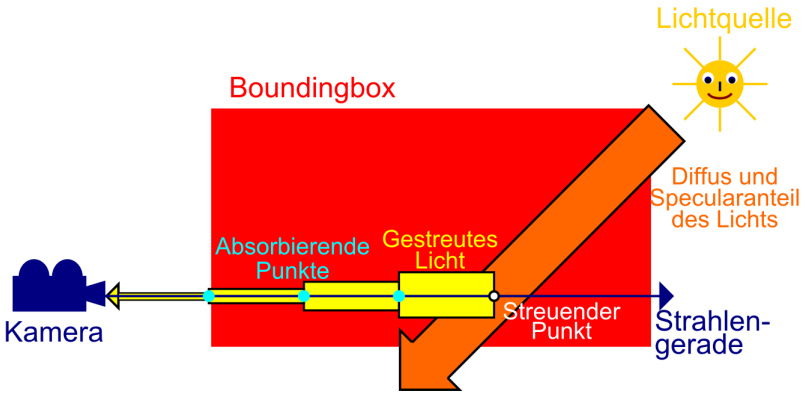


Abbildung 3.24: **Simulierter Intensitätsverlauf beim Raymarching.** Die Dicke der Pfeile gibt die Intensität des Lichts an. Der von der Lichtquelle ausgehende diffuse und specular Lichtstrahl wird zwar beim Passieren des Volumens gestreut, verliert beim Passieren jedoch keine Intensität durch Absorption. Die isotrope Ambient-Strahlung wurde hier in diesem Bild nicht eingezeichnet, jedoch verhält sie sich analog. Erst nach der Streuung im weissen Punkt wird das Licht auf dem Weg zur Kamera durch die hellblauen Raymarchingschrittpunkte zum Teil absorbiert, so dass nur noch ein Bruchteil des gestreuten Lichts bei der Kamera ankommt.

So lässt sich der Transmissionsgrad iterativ berechnen:

$$Tr_0 = 1$$
$$Tr_n = Tr_{n-1} * (1 - Op_{n-1}) \forall n > 0$$

Dadurch gilt für die eintreffende Intensität bei der Kamera für diesen Punkt n:

$$\overrightarrow{I}_{PunktZuKamera,n} = Tr_n * Op_n * \overrightarrow{I}_{PunktZuKamera,n}$$

Die bei der Kamera eintreffende Gesamtintensität aller Punkte entlang des Strahls entspricht nun der Summe der einzelnen Intensitäten der Punkte:

$$\overrightarrow{I}_{Gesamt} = \sum_{n=0} \overrightarrow{I}_{PunktZuKamera,n}$$

Dies kann man in einen iterativen Algorithmus für das Raymarching zusammenfassen. Dieser ist in Abbildung 3.25 gezeigt.

Das vorzeitige Beenden der Schleife, sobald ein Transmissionsgradwert unterschritten wird, stellt hierbei eine einfache Performanceoptimierung dar, welche als Early-Ray-Termination bezeichnet wird. Denn je mehr der Transmissionsgrad abnimmt umso kleiner ist der Anteil, den die verbleibenden Raymarchingschritte zur Gesamtintensität des Strahls beitragen können. Wird der Transmissionsgrad sehr klein, kann man

```
Transmissionsgrad = 1;
AngehäufteIntensität = 0;
for(Raymarchingskalar = Startskalar; Raymarchingskalar <
    Austrittsskalar; Raymarchingskalar++)
{
.....
  Sampling, Phong
.....
  AngehäufteIntensität += Transmissionsgrad * OpazitätDesPunktes *
                          PhongIntensitätDesPunktes;
  Transmissionsgrad *= (1 - OpazitätDesPunktes);
  if(Transmissionsgrad < Schwellwert)
    break;
....
}
```

Abbildung 3.25: **Pseudocode des A-Compositings.**

deshalb das Raymarching abbrechen, ohne dass sich dies störend auf die Bildqualität auswirkt.

Zu guter Letzt verwendet man dieses A-Compositing um aus der Gesamtintensität des Raymarchings $\overrightarrow{I}_{Gesamt}$, dem nach dem Raymarching verbleibenden Transmissionsgrad $Tr_{verbleibend}$ und der Hintergrundfarbe $\overrightarrow{HiFa}$ die Farbe des Pixels $\overrightarrow{PiFa}$ des Strahls zu berechnen:

$$\overrightarrow{PiFa} = \overrightarrow{HiFa} * Tr_{verbleibend} + \overrightarrow{I}_{Gesamt}$$

Diese Pixelfarbe $\overrightarrow{PiFa}$ wird anschliessend an die Stelle des entsprechenden Pixels in den Framebuffer geschrieben.

Anmerkung 3.2: Hier wurde die in der Literatur häufig vorkommende Definition verwendet, dass Absorption das Gegenteil von Transmission ist. Laut physikalischer Definition beschreibt Absorption nur denjenigen Anteil der Intensität der beim Passieren des Lichtes durch Materie in eine andere Energieform, wie zB. Wärmeenergie, umgewandelt wird. Der Begriff der Absorbanz wäre somit in diesem Fall korrekter.

3.6.6 Erweiterung des Standardverfahrens um eine Deep-Shadow-Map

Das Ziel dieses Punktes ist, das Standardverfahren um eine Deep-Shadow-Map zu erweitern und dadurch Schattenwurf zu ermöglichen. Denn eine im letzten Punkt getroffene Vereinfachung ist, dass das Licht der Lichtquelle beim Passieren des Volumens vor der Streuung nicht absorbiert wird und dadurch überall innerhalb der Volumendaten die selbe Intensität besitzt. Um dies zu vermeiden müsste man berechnen, welcher Anteil der Lichtintensität bei einem bestimmten Raymarching-Punkt noch ankommt. Dafür müsste man für die Diffus- und Specularintensität (siehe Anmerkung 3.3) einen

Ohne Deep-Shadow-Map: Mit Deep-Shadow-Map:

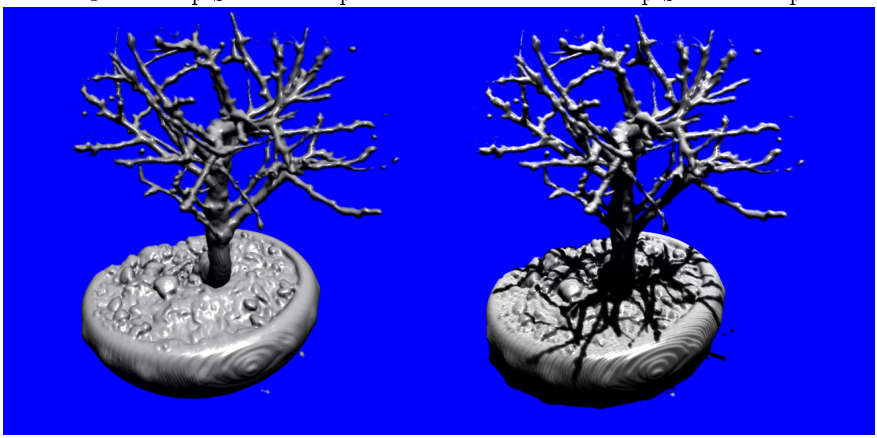


Abbildung 3.26: **Schattenwurf durch eine Deep-Shadow-Map.**

Strahl in Richtung der Lichtquelle aussenden. Entlang dieses Strahls wird dann der Prozentsatz der beim Punkt noch eintreffenden Lichtintensität per Raymarching bestimmt. Dieses Raytracing wäre allerdings wegen des Rechenaufwands für diese zusätzlichen Strahlen nicht mehr echtzeitfähig. Das Problem umgeht man indem man die Transmissionsgrade an den Stellen der Voxelmittelpunkte durch das Aussenden von Strahlen in Richtung des Lichts in einem zusätzlichen vorbereitenden Schritt berechnet. Die Transmissionsgrade werden dann in einer 3D-Textur abspeichert. Die Textur wird anschliessend ebenfalls beim Raymarching an den entsprechenden Schrittpositionen gesamplet. Dieser gesamplte Transmissionsgrad $Tr_{PunktZuLichtquelle}$ fliesst dann in die Phong-Lichtberechnung wie folgt ein:

$$\overrightarrow{I}_{Phong} = \overrightarrow{Fa} * \left(\overrightarrow{I}_{Ambient} + Tr_{PunktZuLichtquelle} * (\overrightarrow{I}_{Diffus} + \overrightarrow{I}_{Specular}) \right)$$

Der Resultat der Deep-Shadow-Map wird in Abbildung 3.26 gezeigt.

Anmerkung 3.3: Dieser Punkt behandelt nur die Absorption der Diffus- und Specularintensität der Lichtquelle und damit den Schattenwurf. Allerdings lässt sich das Verfahren auch auf die Ambientintensität erweitern. Die Absorption des Ambientanteils wird als Ambient-Occlusion bezeichnet. Das Erstellen einer Ambient-Occlusion-Map verläuft im Allgemeinen nahezu analog zur Deep-Shadow-Map. Der Hauptunterschied ist, dass man viele Strahlen in alle Richtungen aussenden muss, um die isotrope Natur des Ambientanteils zu berücksichtigen.

```
for(Raymarchingskalar = Startskalar; Raymarchingskalar <
    Austrittsskalar; Raymarchingskalar++)
{
  SchrittTex = StrahlTex(Raymarchingskalar);
  Opazität = NehmeSample(Opazitätstextur, SchrittTex);
  TransformierteOpazität = NehmeSample(TransferfunktionOpazitätsLUT,
                                       Opazität);
  if(TransformierteOpazität > 0)
    {
    Farbe = NehmeSample(TransferfunktionFarbLUT, Opazität);
    Normalentextursample = NehmeSample(NormalenTextur, SchrittTex);
    PhongIntensität = PhongLichtberechnung(Normalentextursample, Farbe);
    A-Compositing(TransformierteOpazität, PhongIntensität);
    }
}
```

Abbildung 3.27: **Pseudocode des Raymarchings für die Post-Classified Transferfunktion**

3.6.7 Modifikation des Standardverfahrens auf eine Post-Classified Transferfunktion

In diesem Punkt soll das in diesem Kapitel vorgestellte Standardverfahren so umgeändert werden, so dass es statt einer Pre-Classified Transferfunktion eine Post-Classified Transferfunktion verwendet. Dieses abgewandelte Verfahren verläuft weitestgehend analog zum Standardverfahren und unterscheidet sich nur in wenigen Punkten:

- Das Erstellen einer transformierten Opazitätstextur und einer Farbtextur entfällt.
- Man hat die Wahl den Gradienten für die Normalentextur analog zum Pre-Classified Modell oder ohne die Transferfunktion anzuwenden direkt aus der Opazitätstextur zu erstellen. Bei beiden Möglichkeiten kann es von Fall zu Fall zu Einbussen in der Bildqualität kommen. Generell ist die vorberechnete Normale bei der Post-Classified Transferfunktion ungenauer als bei der Pre-Classified Transferfunktion. Deshalb setzen viele Anwendungen darauf, die Normale bei der Post-Classified Transferfunktion ebenfalls "on the fly" beim Raymarching durch mehrfaches Sampling der Opazitätstextur zu berechnen. Dies wurde hier allerdings nicht durchgeführt.
- Das Raymarching muss auf die Post-Classified Transferfunktion angepasst werden. Dessen Pseudocode ist in Abbildung 3.27 gezeigt.

4 Empty-Space-Skipping mit Hilfe eines Octrees

4.1 Aufbau und Konstruktion des Octrees

Viele der Optimierungen für Volumenraycaster beschäftigen sich damit, wie man die komplett transparenten Voxel innerhalb der Volumendaten beim Raymarching überspringen kann. Diese Techniken werden als Empty-Space-Skipping bezeichnet. In diesem Kapitel soll die Verwendung eines Octrees für dieses Empty-Space-Skipping erläutert werden.

Der in der Arbeit verwendete Octree ist ein nahezu vollständiger Octree. Nur diejenigen Knoten, welche komplett ausserhalb der Volumendaten liegen würden, sind nicht im Octree enthalten. Dieser Octree wird Ebene für Ebene in einem einzigen Array bzw. OpenCL-Buffer gespeichert. Dadurch kann man für einen gegebenen Punkt innerhalb der Volumendaten und eine gegebenen Ebene des Octrees die Speicheradresse des Knotens, welcher den Punkt beinhaltet, berechnen. Deshalb benötigen die Octreeknoten keine Zeiger auf ihre Kinder. Für jeden Knoten wird nur die prozentuale Knotenbelegtheit abgespeichert:

- **Tiefste Ebene:** Auf der tiefsten Ebene bzw. Blattebene entspricht ein Blattknoten einem Würfel mit der Kantenlänge von n Voxel. Die Blattknoten werden hier so ausgerichtet, dass ihre Seitenflächen durch die Interpolationsgrenzen verlaufen. Dadurch fallen die Eckpunkte eines Knotens stets auf Voxelmittelpunkte. Die Belegtheit dieser Blattknoten wird abhängig davon ob man eine Pre-Classified oder eine Post-Classified Transferfunktion verwendet anders ermittelt.
  - **Bei der Pre-Classified Transferfunktion gilt:** Sind einer oder mehrere der Voxel der transformierten Opazitätstextur innerhalb des Knotens nicht komplett transparent so ist hier die Belegtheit eins, sonst null.
  - **Bei der Post-Classified Transferfunktion gilt:** Zuerst wird in der Transferfunktion eine unterste Opazitätsgrenze gesucht, unter welcher sämtliche Opazitäten als komplett transparent klassifiziert werden. Eine obere Grenze wird analog gesucht. Sind die Opazitäten aller Voxel des Knotens kleiner als die unterste Grenze so ist die Belegtheit null. Sind alle Voxelopazitäten grösser als die obersten Grenze so ist die Belegtheit ebenfalls null. Ist dies beides nicht der Fall so wird dem Knoten eine Belegtheit von eins zugewiesen.

  In beiden Fällen kann es vorkommen, dass ein Teil der Voxel, welche ein Blattknoten umfasst, ausserhalb der Volumendaten liegen. Diese Voxel werden bei der Berechnung der Belegtheit nicht berücksichtigt. Der Sinn dahinter, auf der tiefsten Ebene mehrere Voxel zu einem Knoten zusammenzufassen, besteht darin, dass es meist viel teurer ist durch einen Octreeknoten zu traversieren, als durch einen einzelnen komplett durchsichtigen Voxel entlang des Strahls hindurchzumarschieren.
- **Alle höheren Ebenen:** Für alle höheren Ebenen ist die Knotenbelegtheit das arithmetische Mittel der Belegtheit der Blattknoten, welche der Knoten

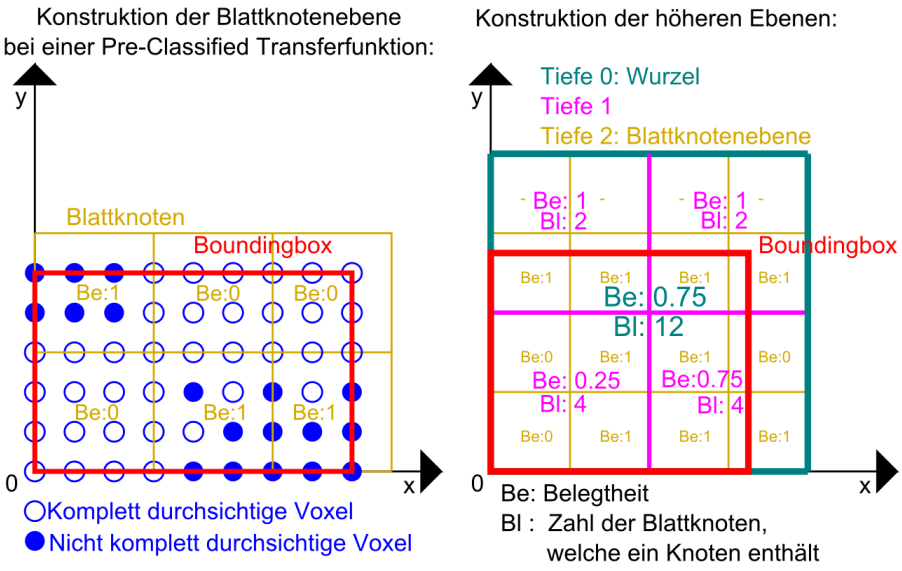


Abbildung 4.1: **Aufbau und Konstruktion des verwendeten Octrees.** Aus Gründen der leichteren Darstellung wird auf beiden Bildern der dem Octree entsprechende Quadtree gezeigt.

umfasst. Diese Belegtheit lässt sich auch aus der Belegtheit der direkten Kinder und der Zahl der Blattknoten, die diese Kinder umfassen, berechnen. So muss man bei der Belegtheitsberechnung nur maximal auf acht Knoten zugreifen. Der Zugriff auf die potentiell vielen Blattknoten, welche der Knoten, dessen Belegtheit es zu berechnen gilt, umfassen kann, bleibt einem dadurch erspart.

Diese Konstruktionsregeln werden noch einmal in Abbildung 4.1 gezeigt. Durch diese Definition ist die Konstruktion des Octrees peinlich parallel auf Knotenbasis innerhalb einer Ebene. Dadurch eignet sich dieser Octree um ihn, von der tiefsten Ebene an, Ebene für Ebene schnell auf der Graphikkarte zu bauen. Allerdings besitzen die höchsten Ebenen des Octrees nur wenige Knoten, so dass man die Parallelität der Graphikkarte nur schlecht ausnutzen kann. Dies spielt allerdings nur eine geringe Rolle, da die Zahl der Knoten und damit die benötigten Rechenoperationen für das Erstellen einer Ebene exponentiell mit der Octreetiefe anwachsen. Somit wünscht man sich vor allem auf den kostspieligen tiefen Octreeebenen eine hohe Parallelität, während eine geringe Parallelität auf den höheren Octreeebenen sich kaum negativ auf die Performance beim Erstellen des Octrees auswirkt.

Beispielhaft soll hierfür ein Volumenmodell mit 512^3 Voxel betrachtet werden. Fasst man jeweils 4^3 Voxel zu einem Blattknoten zusammen so werden auf der tiefsten

Ebene 128^3 Knoten benötigt. Erstellt man für jeden Knoten einen Thread so ergeben sich 2097152 Threads. Nun soll angenommen werden, dass eine moderate Occupancy von 32 Warps ausreichen soll um eine Geforce 580 GTX bei diesen Octreekonstruktionskernels auszulasten. Damit man auf dieser Graphikkarte diese Occupancy auf allen 16 Multiprozessoren erreichen kann, werden gerade einmal 16384 Threads benötigt. Unter der sehr vereinfachenden Annahme, dass die GPU für die Abarbeitung eines Blocks von 16384 Threads eine Zeiteinheit benötigt, benötigt sie für diese Ebene 128 Zeiteinheiten. Bei den beiden Ebenen darüber sind es 262144 und 32768 Knoten bzw. Threads. Beides reicht ebenfalls aus um die GPU gut auszulasten. Unter der selben vereinfachenden Annahme würde die GPU für diese beiden Ebenen 16 und 2 Zeiteinheiten benötigen. Erst bei den restlichen 5 darauf folgenden Ebenen, welche allerdings insgesamt nur noch 4681 Knoten beinhalten, reicht die Zahl der Knoten nicht mehr aus um die GPU auszulasten. Dadurch nährt sich die Zeit, welche man braucht um eine der höchsten Ebenen zu erstellen, einem konstanten Wert an. Nimmt man ebenfalls vereinfacht an, dass sofern weniger als 16384 Threads zur Verfügung stehen, die GPU für die Ausführung einer globalen ND-Range immer eine Zeiteinheit benötigt, so benötigt die GPU für das Erstellen der restlichen 5 Ebenen insgesamt nur noch 5 Zeiteinheiten. Somit ist die GPU bei der Erstellung des Octrees nur über einen Zeitraum von 5 Zeiteinheiten schlecht ausgelastet, während sie über einen Zeitraum von 146 Zeiteinheiten gut ausgelastet ist.

In Wirklichkeit ist die Abarbeitung eines Threads in der Blattknotenebene jedoch meist wesentlich teurer als die Abarbeitung eines Threads in einer höheren Ebene. So müsste ein Thread in dem soeben genannten Beispiel in der Blattknotenebene 64 Voxel betrachten, während er in einer höheren Ebene nur 8 Knoten betrachten müsste. Nimmt man deshalb zusätzlich an, dass die GPU für die Abarbeitung eines Blocks von 16384 Threads in der Blattknotenebene 8 Zeiteinheiten benötigt, während sie für einen gleich grossen Block in einer höheren Ebene nur eine Zeiteinheiten benötigt, so ist die GPU über einen Zeitraum von 1042 Zeiteinheiten gut ausgelastet und nur während einen Zeitraum von 5 Zeiteinheiten schlecht ausgelastet.

An diesem einfachen Beispiel wird deutlich, dass die geringere Parallelität auf den oberen Ebenen nur einen geringen Einfluss auf die Auslastung der GPU beim Erstellen des Octrees insgesamt hat. Hierzu könnte man auch eine Vielzahl von Benchmarks ausführen. Eine weitergehende Untersuchung würde den Rahmen dieser Arbeit überschreiten.

4.2 Traversierung

Nachdem nun die Konstruktion des Octrees erläutert worden ist, beschäftigt sich dieser Punkt mit der Traversierung des Octrees. Da OpenCL keine rekursiven Funktionsaufrufe unterstützt, muss man den Baum iterativ traversieren. Des Weiteren unterstützt es ebenfalls keine dynamischen Datenstrukturen im privaten Speicher, weshalb man den Speicher für den Octreestack vor der Traversierung statisch allokieren muss. Da auf diesen indexiert zugegriffen wird, befindet er sich im CUDA-lokalen Speicher. Die Implementierung des Raycasters mit Octree sieht dementsprechend, wie in Abbildung 4.2 gezeigt aus.

Die ersten Schritte sind hierbei identisch mit den entsprechenden Schritten des Stan-

```
ErstelleThreads();
ErstelleStrahl();
BerechneSchnittpunkteMitDerBoundingBox();
TransformationDesStrahlsInTexturkoordinaten();
AustrittsskalarStack[Stackgrösse];
AustrittsskalarStack[0] = AustrittsskalarDerBoundingBox;
OctreeDepth = 0;
Traversierungsskalar=BestimmeTraversierungsAnfang();

while(AustrittsskalarStack[0] > Traversierungsskalar)
  {
  while(Traversierungsskalar >= AustrittsskalarStack[OctreeDepth])
    {
    OctreeDepth--;
    }

  OctreeDepth++;
  Knoten = BerechneKnoten(OctreeDepth, Strahl, Traversierungsskalar);
  KnotenBelegtheit = LadeKnotenBelegtheitAusDemGlobalenSpeicher(Knoten);
  Austrittsskalar = BerechneAustittskalar(Knoten);

  if(KnotenBelegtheit > SchwellWert)
    {
    MarschiereMitStrahlVonTraversierungsskalarZuAustrittsskalar();
    Traversierungsskalar = Austrittsskalar;
    OctreeDepth--;
    }
  else if(KnotenBelegtheit == 0)
    {
    Traversierungsskalar = Austrittsskalar;
    OctreeDepth--;
    }
  else
    {
    AustrittsskalarStack[OctreeDepth] = Austrittsskalar;
    }
  }

SchreibeDasErgebnisInDenFramebuffer();
```

Abbildung 4.2: **Pseudocode der Octreetraversierung.**

dardverfahrens. Zuerst werden die Threads erstellt, dann die zu den Threads gehörenden Strahlen. Danach erfolgt die Schnittpunktberechnung und die Transformation in normalisierte Texturkoordinaten. Die Boundingbox wird bei der Schnittpunktberechnung allerdings etwas kleiner gewählt um sicherzustellen, dass das Eintrittsskalar und Austrittsskalar auf Grund von numerischen Ungenauigkeiten keinen Punkt ausserhalb der Boundingbox beschreiben. Denn in diesem Fall würde das spätere Nachschlagen des zum Punkt gehörenden Knotens innerhalb des Octrees fehlschlagen. Die Bestimmung des Startwerts für das Traversierungsskalar erfolgt analog wie beim Raymarching des Standardverfahrens. Ist die Kamera innerhalb der Boundingbox, so ist es null, ist sie ausserhalb, so ist es das bei der Schnittpunktberechnung bestimmte Eintrittsskalar.

An der untersten Stelle des Stacks speichert man das soeben berechnete Austrittsskalar aus dieser Boundingbox ab. Schreitet die Traversierung des Octrees über dieses Skalar, so hat der Strahl die Boundingbox verlassen und man kann die Traversierung beenden. Es ist wichtig hier anzumerken, dass die Wurzel selbst, wie in Abbildung 4.1 gezeigt, grösser als die Boundingbox sein kann. Auch können Knoten der tieferen Ebenen aus der Boundingbox herausragen. Deshalb wird als Abbruchkriterium für die Octreetraversierung nicht, wie sonst üblich, ein komplett leerer Stack herangezogen, sondern das Verlassen der Boundingbox. Aus diesem Grund muss man diese Abbruchbedingung nur einmal zu Beginn jedes Schleifendurchlaufs überprüfen und nicht, wie bei anderen Algorithmen üblich, jedes mal, wenn ein Knoten vom Stack genommen wird.

Nun kann mit der Traversierung selbst begonnen werden. Zu Beginn jeder Traversierungsiteration wird die soeben beschriebene Abbruchbedingung überprüft, um festzustellen ob die iterative Octreetraversierung zu beenden ist. Danach findet das Entfernen vom Stack statt. Bei diesem vergleicht man das Traversierungsskalar mit dem obersten Stackskalar. Ist das Traversierungsskalar grösser, so hat man den entsprechenden Knoten bereits verlassen und kann ihn vom Stack entfernen. Dadurch reduziert sich die zu betrachtende Octreetiefe jeweils um eins. Durch das Stackdesign benötigt das Entfernen vom Stack nur einen CUDA-lokalen Speicherzugriff, einen Vergleich und eine Subtraktion, wodurch es sehr günstig ist. Deshalb ist es möglich und sinnvoll alle bereits komplett traversierten Knoten in einem While-Loop auf einmal vom Stack zu nehmen, da so alle Threads eines Warps bei jedem Schleifendurchlauf einen Schritt in dem Baum herab gehen können. Dies wirkt sich insgesamt positiv auf die SIMD-Effizienz aus.

Als Nächstes geht man bei der Traversierung einen Schritt im Octree hinab und bestimmt den Knoten, welcher an der entsprechenden Position des Traversierungsskalars innerhalb der zu betrachtenden Octreetiefe ist. Anhand des Knotens wird nun dessen Speicheradresse berechnet und die prozentuale Belegtheit aus dem Speicher geladen. Danach wird das Austrittsskalar der Strahlengerade aus dem Würfel, welchen der Knoten umfasst, bestimmt. Hierfür verwendet man den selben Algorithmus wie in Kapitel 3.6.3 für den Fall, dass man nur das Austrittsskalar benötigt, und weiss, dass der Strahl die Boundingbox schneidet. Allerdings kann es passieren, dass das Austrittsskalar negativ ist, wenn der Punkt auf den Seitenebenen des Würfels liegt. In diesem Fall muss das Austrittsskalar auf null gesetzt werden. Da das Austrittsskalar ausserhalb des Knotens liegen muss, damit es auch zum Nachschlagen des nächsten

Knotens verwendet werden kann, wird noch ein sehr kleiner Wert dazu addiert. Anschliessend wird es auf die nächste ganze Zahl aufgerundet, um störende Artefakte an den Knotengrenzen zu vermeiden. Das Austrittsskalar darf jedoch nie grösser sein, als diejenigen Austrittsskalare, welche bereits auf den Stack liegen, weil sonst das Nehmen vom Stack fehl schlägt und man zusätzlich eventuell die Boundingbox beim Traversieren oder beim Raymarching verlässt. Deshalb wird dessen Grösse zusätzlich noch auf das an oberster Stelle auf dem Stack liegende Austrittsskalar begrenzt.

Abhängig von der aus dem Speicher geladenen Belegtheit und einer frei wählbaren Schwellwertkonstante gibt es bei jedem Knoten drei verschiedene Fälle wie weiter verfahren wird:

- Die Belegtheit ist null. Hierbei ist der Knoten leer und wird übersprungen. Dem Traversierungsskalar wird der Wert des zuvor berechneten Austrittsskalars zugewiesen und die zu betrachtende Octreetiefe um eins verringert.
- Die Belegtheit des Knotens ist grösser als die Schwellwertkonstante. Hier marschiert man mit dem Strahl durch den Knoten zwischen Traversierungsskalar und Austrittsskalar. Das Raymarching selbst verläuft wiederum analog zu dem in dieser Arbeit vorgestellten Standardverfahren. Danach wird wieder dem Traversierungsskalar der Wert des zuvor berechneten Austrittsskalars zugewiesen und die zu betrachtende Octreetiefe um eins verringert.
- Die Belegtheit des Knotens ist kleiner als die Schwellwertkonstante. In diesem Fall muss man als Nächstes die Kinder des Knotens untersuchen und das Austrittsskalar wird oben auf dem Stack abgelegt.

Die Idee hinter dieser Vorgehensweise und der Schwellwertkonstante stammt aus [CM10] und erklärt sich wie folgt:

- Ist die Belegtheit des Knotens sehr gross so beinhaltet er nur sehr wenig leeren Raum. In diesem Fall würde das Traversieren mehr Performance kosten, als durch den leeren Raum zu marschieren.
- Ist die Belegtheit des Knotens aber sehr klein, so beinhaltet der Knoten viel leeren Raum. Das Traversieren wäre aus diesem Grund günstiger als das Raymarching.

Mit dieser Konstante selbst kann man etwas Finetuning betreiben. Ihre optimale Wahl ist von vielen anderen Parametern, wie die Schrittgrösse oder den Volumendaten selbst, abhängig.

Nach dem Ausführen einer dieser drei genannten Möglichkeit kann mit der nächsten Traversierungsiteration begonnen werden. Ist die Traversierung abgeschlossen so wird wiederum das Ergebnis in den Framebuffer geschrieben. Die hier beschriebene Octreetraversierung ist noch einmal beispielhaft in Abbildung 4.3 visualisiert worden.

Generell hat dieser vollständige Octree den Nachteil, dass man ihn bis zur tiefsten Ebene hin komplett speichern muss. Dadurch benötigt er meist mehr Speicherplatz,

als ein unvollständiger Octree mit Zeigern. Allerdings fällt dies nicht weiter ins Gewicht, da die Volumendaten, je nachdem wie viele Voxel man zu einem Blatt zusammenfasst, meist ein Vielfaches an Speicherplatz verbrauchen. Jedoch wäre es dennoch möglich den Speicherverbrauch des Octrees zu komprimieren, da man nur drei Zustände für einen Knoten speichern muss: Knoten überspringen, Kinder des Knotens untersuchen und Inhalt des Knotens zeichnen. Deshalb müsste man nicht für jeden Knoten die Belegtheit durch einen Float abspeichern, sondern man könnte je zwei Knoten auf drei Bit komprimieren. Diese Verbesserung wurde allerdings hier nicht mehr durchgeführt.

4.3 Kombination der Traversierung und des Raymarchings

4.3.1 Allgemeines

Das Ziel dieses restlichen Kapitels ist es eine Kombination für das Raymarching und die Traversierung des Octrees zu finden, welche eine möglichst hohe SIMD-Effizienz und dadurch eine gute Performance auf einer GPU bietet. Die Betrachtungen im Folgenden sind zwar speziell auf den verwendeten Octree bezogen, sind aber allgemein für jede Empty-Space-Skipping-Datenstruktur gültig, welche sich als eine unregelmässige Folge von zu zeichnenden und zu überspringenden Knoten entlang des Strahls darstellen lässt. Hierunter fallen sämtliche Baumstrukturen und Proximity-Clouds. Letztere werden in [AK05] erläutert.

Ein einzelner Raymarchingschritt durch einen Knoten und das Springen zum nächsten Knoten werden im Folgenden aus Gründen der Einfachheit als eine einzige abstrakte Operation betrachtet, deren SIMD-Effizienz sich nach der in Punkt 2.2.3 vorgestellten Standardformel berechnet. Diese abstrakten SIMD-Effizienz gilt es aus Performancegründen zu erhöhen. Beide Operationen enthalten jedoch dynamische Sprungbefehle. So kann es passieren dass wenn die SIMD-Effizienz einer Operation erhöht wird, die zusätzlich teilnehmenden Threads innerhalb dieser Operationen zusätzlich Warpdivergenz verursachen. Definiert man allerdings die SIMD-Effizienz eines Programmes als das arithmetisches Mittel der SIMD-Effizienz der elementaren Instruktionen, kann es deshalb vorkommen, dass, obwohl man die SIMD-Effizienz dieser beiden abstrakten Operationen erhöht, man die gesamte SIMD-Effizienz des Programms verringert. Dieser Effekt soll in folgenden Überlegungen vernachlässigt werden.

Für die Messungen in diesem Kapitel wurde immer der CT-Scan eines Bonsai aus der Standardansicht mit einer Transferfunktion, welche das Rauschen in den leeren Bereichen beseitigte, gerendert. Diese Standardansicht ist in Abbildung 4.4 dargestellt. Ansonsten wurden, falls nicht anders angegeben, die Standardeinstellungen des Programms verwendet. Einstellungen wie die Schrittgrösse, welche die Graphikqualität beeinflussen, wurden in den Standardeinstellungen so gewählt, dass die Graphikqualität hoch ist. Auch wurde eine Pre-Classified Transferfunktion und keine Deep-Shadow-Map verwendet.

Diejenigen Einstellungen wie die Octreeknotengrösse, welche nur die Performance beeinflussen, wurden so gewählt, dass die Performance bei der besten Kombination, welches dieses Kapitel hervorbringen wird, - Buffered-For-For-With-Flags - gut

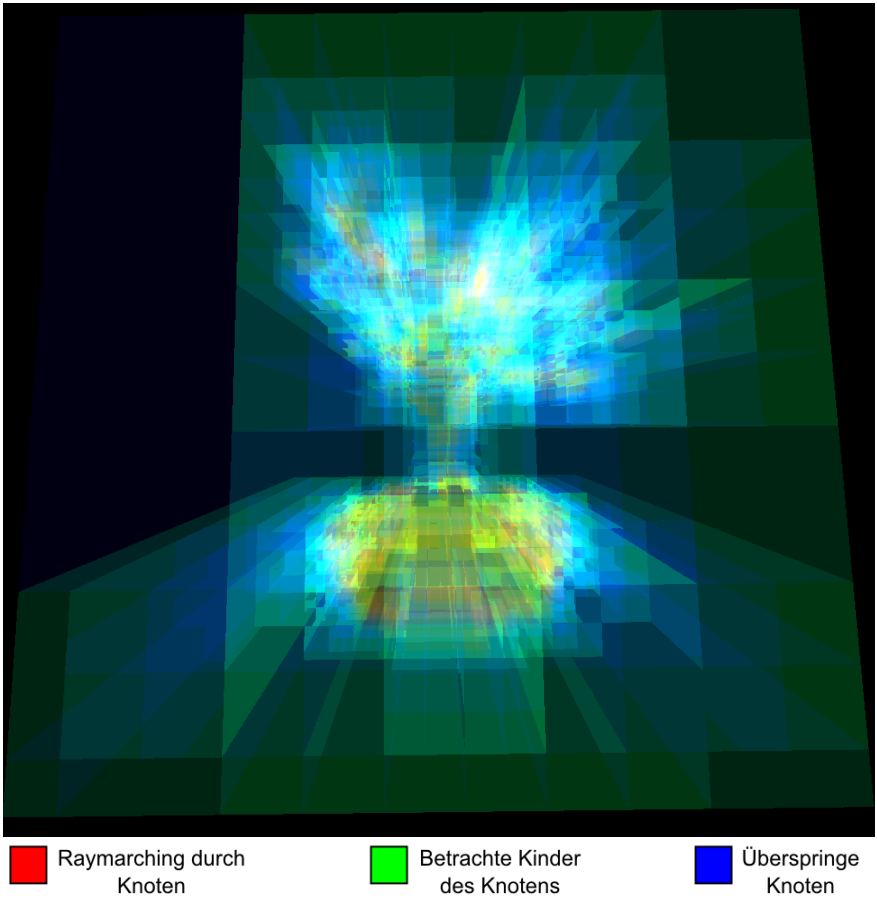


Abbildung 4.3: **Visualisierung des zum Bonsai gehörenden Octrees.** Die verschiedenen Knotenoperationen entlang eines Strahls wurden hier zur Gesamtfarbe aufaddiert. Man kann deutlich erkennen, dass die Farbe in der Abbildung in den Bereichen der Äste deutlich heller ist als im übrigen Bild. Entlang dieser Strahlen werden also wesentlich mehr Octreeknoten traversiert als in den übrigen Bildbereichen.

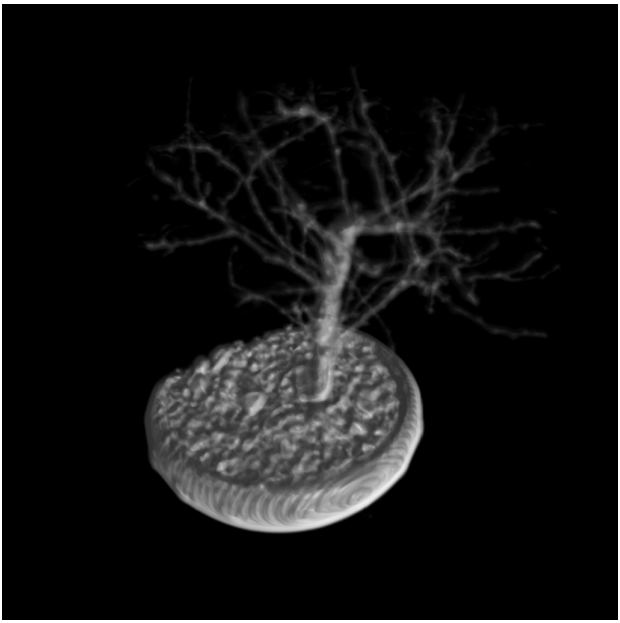


Abbildung 4.4: **Standardansicht des Bonsais.**

ist. Dennoch kann es sein, dass es im Folgenden bei vielen Kombinationen bessere Parameter gibt. Allerdings würde es zu weit führen diese jedes mal zu suchen.

Für die Messungen wurde eine Geforce 580 GTX verwendet. Der gemessene FPS-Gewinn durch die Optimierungen kann sich von Graphikkarte zu Graphikkarte unterscheiden. Jedoch sind die gemessenen abstrakten SIMD-Effizienzen bei der Traversierung und dem Raymarching unabhängig von der Wahl der Graphikkarte, sofern die gewählte Graphikkarte ebenfalls eine Warpgrösse von 32 besitzt. Dies gilt jedoch nicht für die SIMD-Effizienz des gesamten Programmes, da verschiedene Graphikkarten unterschiedliche Befehlssätze haben und dementsprechend sich die Zahl der Maschinenbefehle des kompilierten Programmes unterscheiden kann.

Leider sind im Folgenden keine direkten Vergleiche zwischen CPU und GPU mehr möglich, da das Programm in neueren Versionen nicht mehr auf der Intel-OpenCL-CPU-Implementierung lauffähig ist. Genaue direkte Performancevergleiche sind jedoch auch aus dem Grund nicht sinnvoll, da man für eine vernünftige Bewertung das Programm ebenfalls auf die CPU optimieren müsste. Eine weitergehende Untersuchung würde den Rahmen dieser Arbeit überschreiten. Tests mit einer frühen Version ergaben jedoch, dass sich mit der CPU ohne Octree ungefähr 0.4 FPS und mit dem First-Approach-Octree (erklärt in 4.3.2) 0.8 FPS auf einem Intel Core i7 860 bei dem Bonsai erzielen liessen. Da die Geforce 580 GTX, wie später gemessen, bereits ohne

```
while(OctreeNichtKomplettTraversiert())
{
  Knoten K = SpringeZumNächstenKnoten();
  if(BelegtheitVonKnoten(K) > Schwellwert)
    MarschiereMitStrahlDurchKnoten(K);
}
```

Abbildung 4.5: **Pseudocode der First-Approach-Kombination**

	Standardverfahren ohne Octree	First-Approach-Kombination
SIMD-Traversierung	-	0.86
SIMD-Raymarching	0.99	0.29
FPS	10.3	17.5

Abbildung 4.6: **SIMD-Effizienzen der First-Approach-Kombination**

Octree 10.3 FPS erzielt, wird deutlich, dass die GPUs bei diesem Verfahren wesentlich mehr Performance als eine CPU bieten.

4.3.2 First-Approach-Kombination

Im Punkt 4.1 wurde beispielhaft Octreetraversierung an einer einfachen Kombination erläutert. Diese Kombination entstand ohne auf die Eigenschaften der GPU einzugehen. Deren Pseudocode ist in stark abstrahierter Form in Abbildung 4.5 gezeigt. Auf einer CPU lieferte diese Kombination bereits einen guten Performancegewinn, während ihre Leistung auf der GPU enttäuschend und in vielen Fällen sogar schlechter als ohne Octree ist. Eine der ersten Vermutungen war, dass dies auf die niedrige SIMD-Effizienz beim Raymarching zurückzuführen ist. Um diese Vermutung zu überprüfen wurde die SIMD-Effizienz gemessen und die Ergebnisse dieser Messung wurden in Abbildung 4.6 eingetragen. Somit bestätigt das Messergebnis die Vermutung der geringen SIMD-Effizienz.

Diese niedrige SIMD-Effizienz beim Raymarching lässt sich wie folgt erklären: Die verschiedenen Threads eines Warps werden den Octree wahrscheinlich unterschiedlich durchlaufen und deshalb bei einem bestimmten Schleifendurchlauf jeweils verschiedene Knoten betrachten. Manche Threads wollen den betrachteten Knoten überspringen, und diejenigen Threads, welche durch den jeweiligen Knoten hindurchmarschieren wollen, werden dies entlang eines unterschiedlich langen Geradensegments, als Schnitt zwischen Strahl und Octreeknoten, tun wollen. Die für das Marschieren benötigte Zahl der Iterationen kann so innerhalb der Threads eines Warps, insbesonders da sich die durchzumarschierenden Octreeknoten stark in ihrer Grösse unterscheiden können, ebenfalls stark schwanken. Wegen des Warpdivergenzverhaltens bei Schleifen auf der GPU führt nun der Warp beim Marschieren so viele Schritte aus, bis all seine Threads die Schleife verlassen haben. Da sich die Anzahl der Schleifendurchläufe innerhalb eines Warps stark voneinander unterscheidet ist hier, bei dem rechenauf-

wendigen Marschieren, die SIMD-Effizienz niedrig und es geht viel der Rechenleistung verloren. Dies ist beispielhaft noch einmal in Abbildung 4.7 und 4.8 erläutert.

4.3.3 For-For-Kombination

Um die SIMD-Effizienz aus dem First-Approach zu verbessern soll im Folgenden eine Lösung aus [NVd] für ein ähnliches Problem, nämlich die Kombination von Schnittpunktberechnung und Datenstrukturtraversierung beim Raytracing, an die Gegebenheiten des Volumenraycastings angepasst werden. Die beiden in [NVd] vorgestellten Kombinationen If-If und While-While sind in Abbildung 4.9 und 4.10 zitiert und wurden in Abbildung 4.11 und 4.12 in den Volumenraycaster übertragen.

Beide Lösungsansätze eignen sich jedoch nicht für den Volumenraycaster. Denn bei einem Raytracer ist es optimal, wenn die Knoten der dort verwendeten hierarchischen Datenstruktur klein sind und wenige Primitive enthalten, da man dadurch weniger Schnittpunktberechnungen durchführen muss. Im Gegensatz dazu ist es beim Volumenraycaster vorteilhaft, wenn die Knoten möglichst gross sind, da man dadurch potentiell Rechenaufwand beim Traversieren einsparen kann. Dementsprechend nimmt auch die Zahl der Schritte bei einem durchzumarschierenden Knoten ein viel grösseres Werteintervall als die Zahl der Primitive im Raytracer an.

Bei If-If führt dies bei dem Volumenraycaster nun dazu, dass die Threads eines Warps jeweils nach unterschiedlich vielen Schritten mit dem Marschieren des Knotens abschliessen und danach Traversieren wollen. Da die Threads meist nach wenigen Traversierungsschritten ihren nächsten Knoten gefunden haben, schliessen sie das Traversieren ab, bevor viele andere Threads ebenfalls traversieren wollen. Dadurch wird der Baum bei einer Iteration des Mainloops immer nur von wenigen Threads gleichzeitig durchlaufen, wodurch Warpdivergenz entsteht und Rechenzeit verloren geht.

Bei While-While tritt ebenfalls auf Grund der stark schwankenden Zahl der Iterationsschritte bei dem Marschieren innerhalb eines Warps das selbe Problem auf wie beim Marschieren im First-Approach. Die Schleife wird hier ebenfalls so oft durchlaufen, bis alle Warpthreads fertig sind, wodurch wegen der stark schwankenden Zahl der Schleifendurchläufe innerhalb eines Warps viel Rechenzeit verloren geht.

Um diese Probleme zu umgehen, wurde die If-If und While-While Kombinationen zu einer For-For-Kombination, deren Pseudocode in Abbildung 4.13 zu sehen ist, verallgemeinert. Die If-If und While-While Kombination stellen hierbei Spezialfälle dar, welche sich bei maximal einem oder bei maximal unendlich vielen For-Schleifendurchläufen ergeben. Die For-For-Kombination wird ebenfalls in [NVd] kurz erwähnt. So wird dort ausgesagt, dass man in Kombination mit Persistenten Threads die While-While Schleifenorganisation vermeiden sollte und man bessere Performance dadurch erzielen kann, indem man die maximale Zahl an Schleifendurchläufen bei While-While begrenzt. Dies entspricht somit der For-For-Kombination.

Durch diese For-For-Kombination soll nun erreicht werden, dass im Gegensatz zu If-If, die Threads nicht mehr weitestgehend einzeln mit dem Marschieren abschliessen, und danach alleine traversieren, wodurch die SIMD-Effizienz beim Traversieren erhöht wird. Zusätzlich wird, im Gegensatz zu While-While, in dieser For-For-Kombination durch Begrenzung der maximalen Anzahl der Schleifendurchläufe die Performance

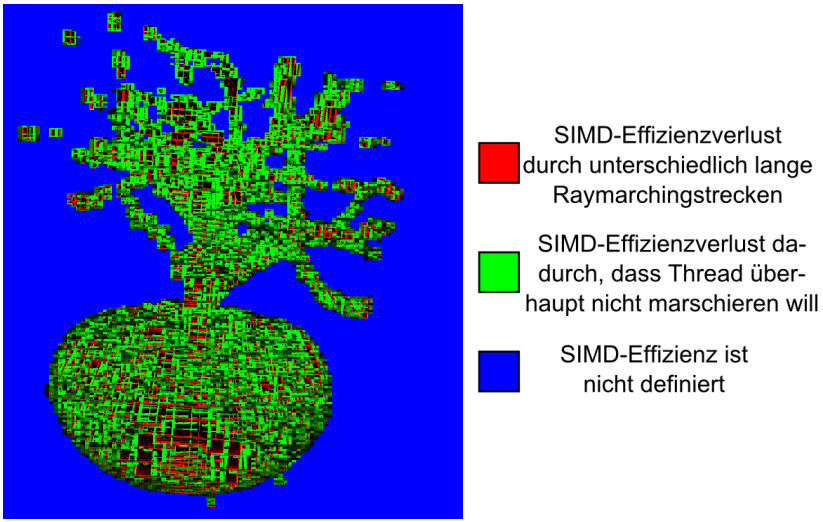


Abbildung 4.7: **Visualisierung der Raymarching SIMD-Effizienz der First-Approach-Kombination durch einen bestimmte Knoten.** Die blaue Farbe bedeutet, dass kein Thread des zum jeweiligen Thread bzw. Pixel gehörenden Warps einen Knoten nach der 15. Octreetraversierungsiteration mehr zeichnen will. Der rote Farbkanal und der grüne Farbkanal beziehen sich auf die erste Octreetraversierungsiteration nach der 15. Octreetraversierungsiteration des Warps, bei der mindestens ein Thread durch einen Knoten marschieren will. Es wurde so eine späte Traversierungsiteration gewählt damit es wahrscheinlich ist, dass sich die Threads des Warps bereits beim Durchlaufen des Octrees desynchronisiert haben. Die grüne Farbe demonstriert, dass der Thread des Pixels den Inhalt seines Knotens nicht rendern will. Die Intensität des Grüns ist hierbei direkt proportional zu der Zahl der Raymarchingschritte des zum Thread gehörenden Warps für diese Iteration. Die maximale grüne Intensität besagt, dass der gesamte Warp gezwungenermassen 30 oder mehr Schritte bei diesen Knoten machen wird. Der rote Farbkanal zeigt, dass der Thread zwar marschieren will, jedoch nicht derjenige Thread des Warps mit der maximalen Anzahl an Raymarchingiterationen durch seinen Knoten ist. Die Intensität des roten Farbkanals ist dabei direkt proportional zum Prozentsatz der Raymarchingschritte des Warps, bei denen der Thread untätig ist. Die maximale rote Intensität gibt hierbei an, dass der Thread nur die Hälfte oder weniger der maximalen Schritte des Warps nehmen will. Dementsprechend ist an dunklen Stellen des Bildes der Beitrag des Threads zu SIMD-Effizienz des Warps hoch und an grünen und roten Stellen gering. Bei diesem Bild wird durch das Vorherrschen der grünen Farbe deutlich, dass an den meisten Stellen SIMD-Effizienz dadurch verloren geht, dass der Thread überhaupt nicht marschieren will. Nur an wenigen Stellen geht SIMD-Effizienz verloren, da sich die Längen der Marschwege voneinander unterscheiden.

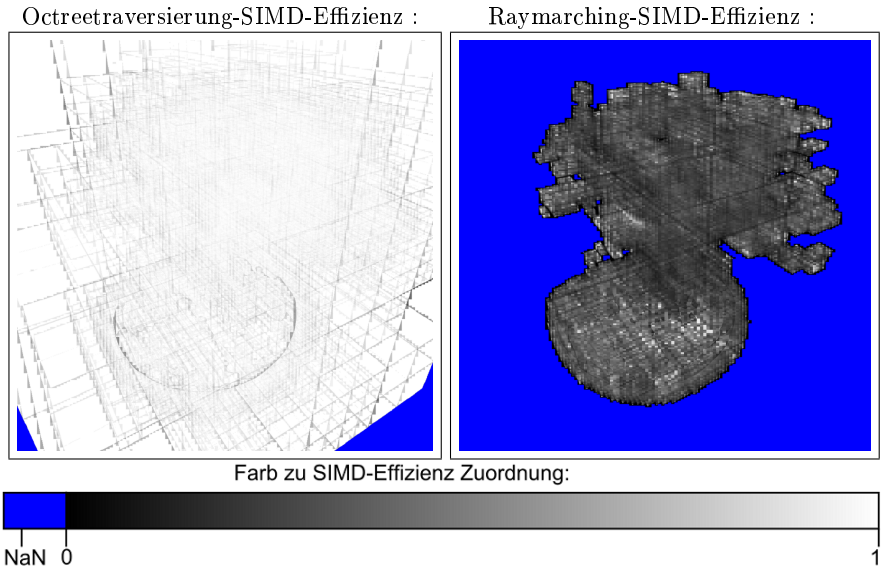


Abbildung 4.8: **Visualisierung der SIMD-Effizienzen der First-Approach-Kombination.** In dem rechten Bild ist die SIMD-Effizienz nicht mehr auf das Raymarching durch einen bestimmten Knoten, sondern auf das Raymarching entlang des gesamten Strahls bezogen. Die linken Teile des Bonsais, also der linke Schalenrand und die linken Blätter, welche nahe an der Kamera sind haben beim Raymarching noch eine relativ hohe SIMD-Effizienz. Je weiter sich die Teile des Bonsais jedoch von der Kamera entfernen desto niedriger wird die SIMD-Effizienz. Daraus lässt sich schliessen, dass sich die Threads bei der Traversierung zunehmend desynchronisieren und die SIMD-Effizienz dadurch reduziert wird. Die hohe Helligkeit im Bild der Octreetraversierung verdeutlicht, dass die SIMD-Effizienz bei der Octreetraversierung bereits sehr gut ist.

```
while ray not terminated
  if node does not contain primitives
    traverse to the next node
  if node contains untested primitives
    perform a ray-primitive intersection test
```

Abbildung 4.9: **Pseudocode von der If-If-Kombination für einen Raytracer.**

```
while ray not terminated
  while node does not contain primitives
    traverse to the next node
  while node contains untested primitives
    perform a ray-primitive intersection test
```

Abbildung 4.10: **Pseudocode von der While-While-Kombination für einen Raytracer.**

```
Knoten K;
bool Raymarching = false;
while(true)
{
  if(!Raymarching)
  {
    if(OctreeKomplettTraversiert())
      return;
    K = SpringeZumNächstenKnoten();
    if(BelegtheitVonKnoten(K) > Schwellwert)
      Raymarching=true;
  }

  if(Raymarching)
  {
  MarschiereEinenSchrittDurchKnoten(K);
  if(KomplettDurchKnotenMarschiert(K))
    Raymarching = false;
  }
}
```

Abbildung 4.11: **Pseudocode von der If-If-Kombination für einen Volumenraycaster.**

verbessert. Denn nun wird die Schleife auf jeden Fall nach maximal einer bestimmten Anzahl von Durchläufen abgebrochen, wodurch der gesamte Warp nicht mehr von dem zuvor am längsten iterierenden Thread aufgehalten wird.

Auch im Vergleich zur First-Approach-Kombination soll diese For-For-Kombination Vorteile mit sich bringen. Durch das mehrfache Ausführen der Traversierung soll erreicht werden, dass nun mehr Threads eines Warps einen Knoten gefunden haben, wenn der Warp mit dem Marschieren anfängt. Zudem soll durch die Begrenzung der maximalen Raymarchingschritten ebenfalls erzielt werden, dass der Warp nicht mehr von den am längsten iterierenden Thread aufgehalten wird. Allerdings wird wohl beim For-For im Vergleich zum First-Approach die SIMD-Effizienz der Octreetraversierung verringert, da die Warpthreads bei der For-For-Kombination entweder nicht marschieren werden oder nach unterschiedlich vielen Traversierungsschritten einen

```
Knoten K;
while(true)
{
  while(true)
  {
    if(OctreeKomplettTraversiert())
      return;
    K = SpringeZumNächstenKnoten()
    if(BelegtheitVonKnoten(K) > Schwellwert)
      break;
  }
  while(NichtKomplettDurchKnotenMarschiert(K))
    MarschiereEinenSchrittDurchKnoten(K);
}
```

Abbildung 4.12: **Pseudocode von der While-While-Kombination für einen Volumenraycaster.**

```
Knoten K;
bool Raymarching = false;
while(true)
{
  if(!Raymarching)
    for(int i = 0; i < MaximaleZahlAnTraversierungsSchritten; i++)
    {
      if(OctreeKomplettTraversiert())
        return;
      K = SpringeZumNächstenKnoten();
      if(BelegtheitVon(K) > Schwellwert)
      {
        Raymarching = true;
        break;
      }
    }
  if(Raymarching)
    for(int i = 0; i < MaximaleZahlAnRaymarchingSchritten; i++)
    {
      MarschiereEinenSchrittDurchKnoten(K);
      if(KomplettDurchKnotenMarschiert(K))
      {
        Raymarching = false;
        break;
      }
    }
}
```

Abbildung 4.13: **Pseudocode vom der For-For-Kombination.**

	First-Approach	If-If	While-While	For(2)-For(10)
SIMD-Traversierung	0.86	0.31	0.51	0.65
SIMD-Raymarching	0.29	0.86	0.63	0.68
FPS	17.5	18.5	22.2	26.3

Abbildung 4.14: **SIMD-Effizienzen der First Approach-, If-If-, While-While- und For-For-Kombinationen.** In den Klammern hinter den beiden Fors ist die maximale Anzahl an Forschleifendurchläufen angegeben. If-If und While-While wurden bei dieser Messung durch For(1)-For(1) bzw. durch For(unendlich)-For(unendlich) simuliert.

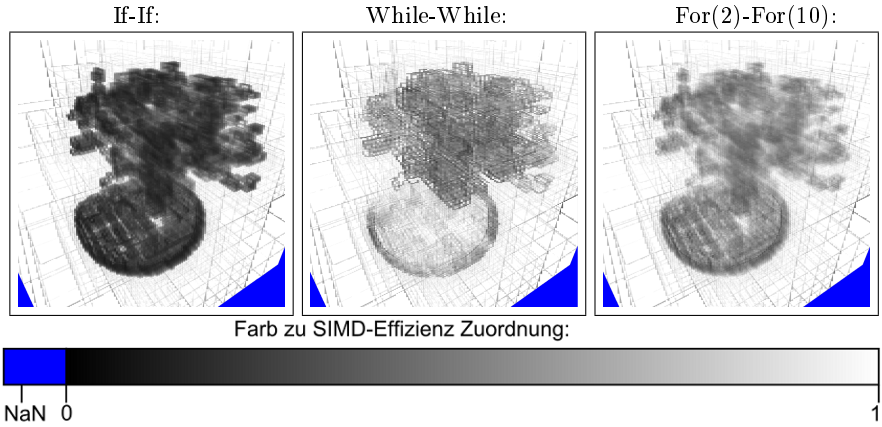


Abbildung 4.15: **Visualisierung der SIMD-Effizienzen beim Traversieren bei der If-If-, While-While- und For-For-Kombination.** Deutlich zu erkennen ist die sehr niedrige SIMD-Effizienz beim If-If. Die SIMD-Effizienz beim While-While schwankt, da die Schleifendurchläufe nicht limitiert sind, stark. For(2)-For(10) liefert diesmal das beste Ergebnis.

durchzumarschierenden Knoten finden werden.

Um den Vorteil dieser For-For-Kombination gegenüber den anderen Kombinationen zu überprüfen wurden nun für die verschiedenen Kombinationen die SIMD-Effizienz und die Performance gemessen. Die If-If- und While-While-Kombination wurden dabei durch die For-For-Kombination simuliert. Bei For-For wurden die beiden For-Schleifenparameter so gewählt, dass die Performance maximal war. Die Messungen ergaben die in Abbildung 4.14 angegebenen Werte. Die SIMD-Effizienzen bei der Traversierung und beim Raymarching wurden wieder in Abbildung 4.16 und 4.15 visualisiert.

Aus diesen Messungen folgt, dass die For-For-Kombination ihre Erwartungen erfüllt, und die Performance durch sie erhöht wird. Sie besitzt zwar bei der optimalen Wahl

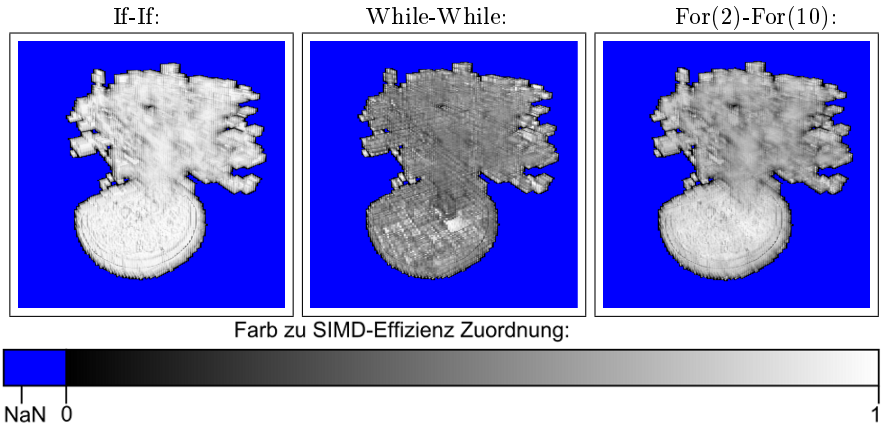


Abbildung 4.16: **Visualisierung der SIMD-Effizienzen beim Raymarching bei der If-If-, While-While- und For-For-Kombination.** Deutlich ist zu erkennen, dass If-If die höchste SIMD-Effizienz besitzt. Beim While-While ist die SIMD-Effizienz am niedrigsten und schwankt wieder am stärksten. Die SIMD-Effizienz beim For(2)-For(10) ist im Vergleich zu den beiden anderen Kombinationen mittelmässig.

ihrer Parameter bislang weder die höchste SIMD-Effizienz beim Octreetraversieren noch die höchste SIMD-Effizienz beim Raymarching, aber da beides mittelmässig ist, ist dennoch die Performance insgesamt am höchsten.

Ein Performance kostendes Problem bei dieser For-For-Kombination ist allerdings, dass nur ein Zwischenspeicherplatz für einen Knoten existiert, um Informationen von der Traversierung zum Raymarching zu schicken. Deshalb muss der Thread dort mit der Traversierung abbrechen, sobald er einen Knoten für das Raymarching gefunden hat, selbst wenn der Warp weiter traversiert. Auch kann er nicht mit dem Traversieren anfangen, wenn das Raymarching durch den zuletzt gefundenen Knoten noch nicht abgeschlossen worden ist. Ähnlich verhält es sich mit dem Raymarching. So lange das Octreetraversieren keinen neuen Knoten zum Marschieren gefunden hat, kann das Raymarching nicht fortgesetzt werden, selbst wenn der Warp weiter marschieren wird. Beides wirkt sich negativ auf die SIMD-Effizienz aus. Zusätzlich ist es nachteilig, dass die SIMD-Effizienz und damit Performance sehr von der korrekten Wahl der maximalen Schleifendurchläufe der beiden For-Schleifen abhängig sind.

4.3.4 Buffered-For-For-Kombination

Diese Probleme der soeben erläuterten For-For-Kombination lassen sich durch das Einfügen eines Buffers für Knoten grösstenteils beseitigen, wodurch die SIMD-Effizienz noch einmal verbessert wird. Die Buffered-For-For-Kombination ist angelehnt an die Speculative-Verfahren, welche ebenfalls in [NVd] beschrieben werden. Bei diesen Speculative-Verfahren wird das Raytracing-If-If oder While-While-Verfahren

ebenfalls um einen Buffer erweitert. Zusätzlich nehmen alle Threads eines Warps am Traversieren teil, wenn auch nur ein Thread des Warps traversieren will. Wird dann beim Traversieren ein neuer Knoten mit Primitiven gefunden, so wird dieser hinten in den Buffer eingefügt. Schliesst ein Thread mit dem Schnittpunktberechnen bei den Primitiven eines Knotens ab, so wird als Nächstes der vorderste Knoten aus dem Buffer genommen und bei diesem die Schnittpunktberechnungen fortgesetzt. Das zusätzliche Traversieren verursacht gemäss der SIMD-Definition keine zusätzliche Rechenzeit. Allerdings wird durch dieses Verfahren die benötigte Zahl Speicherzugriffe erhöht, da Threads weiterhin, obwohl sich ein Primitiv mit Schnittpunkt innerhalb ihres Buffers befindet, weiter traversieren und demnach auch weiter Daten vom globalen Speicher anfordern. Dadurch kann die Laufzeit erhöht werden. Zuerst wurde der Buffer dieser Optimierung in die für den Volumenraycaster bereits optimierte For-For-Kombination eingebaut. Der Pseudocode dieser Optimierung wird in Abbildung 4.17 gezeigt.

Der Buffer wurde als einen Ringbuffer realisiert, welcher als Array im CUDA-lokalen Speicher liegt. Für jeden Knoten wird in dem Buffer nur das Eintrittsskalar und das Austrittsskalar als Start und als Endpunkt für das Raymarching abgespeichert. Dadurch benötigt der Buffer nur acht Byte pro Knoteneintrag, wodurch man ihn relativ gross wählen kann, ohne viel des CUDA-lokalen Speicher zu belegen.

Eine weitere einfache Optimierung besteht darin, sollte beim Einfügen eines neuen Knotens in den Buffer dieser neue Knoten an den hintersten Knoten im Buffer angrenzen, diesen hintersten Knoten um den einzufügenden Knoten zu verlängern. Dabei wird das Austrittsskalar des hintersten Knotens auf das Austrittsskalar des einzufügenden Knotens gesetzt und es wird kein neuer Buffereintrag vorgenommen. Da es bei den meisten Datensätzen sehr wahrscheinlich ist, dass mehrere durchzumarschierende Knoten in Folge aneinander angrenzen, wird auf diese Weise die effektive Zahl der Knoteneinträge im Buffer stark erhöht. Dadurch kann man bei nahezu allen Strahlen alle Raymarching-Knoten in dem Buffer abspeichern, ohne dass er voll wird.

Des Weiteren stellte sich hier das Problem, dass es im Vergleich zu einem einzigen Raymarchingschritt relativ kostspielig ist, denjenigen Fall zu behandeln, dass man komplett durch den vordersten Knoten im Buffer marschiert ist und deshalb den Knoten aus dem Buffer entfernen will. Da die meisten Warpthreads jedoch bei unterschiedlichen Iterationen einen Knoten aus den Buffer entfernen wollen, besitzt die Behandlung dieses Falls eine niedrige SIMD-Effizienz und wird dadurch vermutlich bei fast jedem Raymarchingschritt ausgeführt. Aus diesem Grund wurde ein einfache Optimierung angewandt: Es hat keine Auswirkung auf die Farbe eines Pixels, ob man aus einem im Buffer befindlichen Knoten hinausmarschiert. Dies gilt jedoch nur, wenn man dafür sorgt, dass diese zu viel marschierte Strecke, falls sie ebenfalls in anderen durchzumarschierenden Knoten enthalten ist, auf keinen Fall ein zweites Mal marschiert wird. Deshalb fasst man nun mehrere Raymarching-Schritte noch einmal zusammen. Ein Thread marschiert diese zusammengefassten Schritte direkt in Folge, bevor er überprüft, ob einer oder eventuell sogar mehrere Knoten aus dem Buffer zu entfernen sind. Auf diese Weise wird die Wahrscheinlichkeit erhöht, dass mehrere Threads gleichzeitig die abgeschlossenen Knoten aus dem Buffer entfernen wollen, wodurch die SIMD-Effizienz dieser Operation erhöht wird. Zusätzlich wird dadurch der Buffer seltener abgefragt. Beides wirkt sich positiv auf die Performance aus.

```
while(true)
{
  if(BufferIstNichtVoll() && OctreeNichtKomplettTraversiert())
      for(int i = 0; i < MaximaleZahlAnTraversierungsSchritten; i++)
      {
           if(OctreeKomplettTraversiert())
             break;
           Knoten K = SpringeZumNächstenKnoten();
           if(BelegtheitVonKnoten(K) > Schwellwert)
           {
             if(HintersterKnotenImBufferGrenztAnKnoten(K))
               VerlängereHintersten KnotenImBufferUmKnoten(K));
             else
             {
               FügeKnotenHintenInBufferEin(K);
               if(BufferIstVoll())
                 break;
             }
           }
      }
  if(BufferIstNichtLeer())
    {
    for(int i = 0; i < MaximaleZahlAnRaymarchingSchritten
        /ZusammenGefassteRaymarchingSchritte; i++)
        {
        if(KomplettDurchVorderstenKnotenMarschiert())
        {
          EntferneDieKomplettDurchMarschiertenKnotenAusBuffer();
          if(BufferIstLeer())
            break;
        }
        for(int j = 0; j < ZusammenGefassteRaymarchingSchritte; j++)
          MarschiereDenNächstenSchrittDurchVorderstenKnotenImBuffer();
        }
    }
  if(BufferIstLeer() && OctreeIstKomplettTraversiert())
    return;
}
```

Abbildung 4.17: **Pseudocode der Buffered-For-For-Kombination.**

	For(2)-For(10)	Buffered-For(10)-For(40)
SIMD-Traversierung	0.65	0.87
SIMD-Raymarching	0.68	0.87
FPS	26.3	30.4

Abbildung 4.18: **SIMD-Effizienzen der For-For-Kombinationen und der Buffered-For-For-Kombination.** Beim Buffered-For-For wurden für diese Messung jeweils acht Raymarchingschritte zusammengefasst, bevor der Buffer abgefragt wird. Dementsprechend gibt der Wert in den zweiten Klammern beim Buffered-For-For nicht die maximale Zahl der Raymarching For-Schleifendurchläufe, sondern die maximale Zahl an Raymarchingschritten bei der Raymarching For-Schleife an.

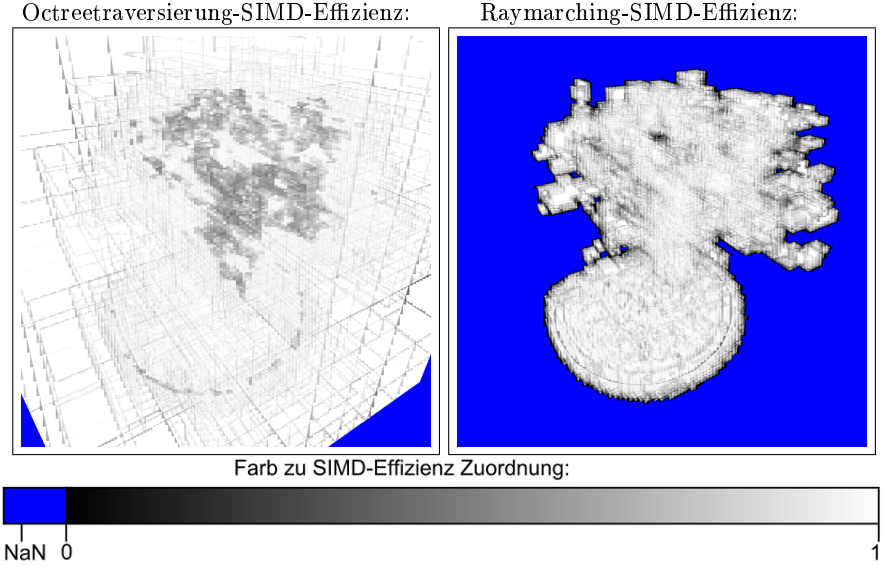


Abbildung 4.19: **Visualisierung der SIMD-Effizienzen der Buffered-For-For-Kombination.** Deutlich sind im linken Bild die fein strukturierten Bereiche, in denen der Buffer vollläuft, als dunklere Gebiete zu erkennen. Abgesehen davon sind beide Bilder bereits hell, woraus folgt, dass die SIMD-Effizienz hoch ist.

Um den Vorteil der Buffered-For-For-Kombination gegenüber der For-For-Kombination zu überprüfen, wurden wieder die SIMD-Effizienz und die FPS gemessen. Die Ergebnisse sind in Abbildung 4.18 eingetragen. Ebenfalls wurden diese SIMD-Effizienzen in Abbildung 4.19 visualisiert.

Bei den Messungen zeigt sich, dass diese Buffered-For-For-Kombination ihre Erwartungen erfüllt und die Performance weiterhin verbessert. Allerdings erhöht sich die

```
while(true)
{
  if(SetzeUndPrüfeFlagsObWarpTraversierenSollte())
    BufferedForOctreetraversierung();
  else
    BufferedForRaymarching();
}
```

Abbildung 4.20: **Pseudocode der Buffered-For-For-With-Flags-Kombination.**

SIMD-Effizienz im Vergleich zur For-For Version ca. um 30 Prozent, während sich die FPS nur um 13 Prozent erhöht. Eine Ursache hierfür sind die zusätzlichen Rechenoperationen, welche durch die Benutzung des Buffers entstehen. Auch die Tatsache, dass man durch das seltene Abfragen des Buffers überflüssige Raymarchingschritte ausführt, lindert die Performance. Weitere mögliche Gründe wären allerdings auch eine bereits eintretende Limitierung der Performance durch Teile der GPU und der Overhead, welcher dadurch entsteht, das fertig gerenderte Bild im Fenster anzuzeigen. Auch kann zusätzliche Warpdivergenz innerhalb des Raymarchings und der Octreetraversierung dazu führen, dass die Performance nicht gleichermassen zunimmt.

Des Weiteren stellt beim Buffered-For-For wiederum die richtige Wahl der drei For-Schleifen-Konstanten ein Problem dar. Fasst man zu viele Raymarchingschritte zusammen, so dass der Buffer nur selten überprüft wird, so kann es vorkommen, dass man durch zu viel leeren Raum marschiert. Wählt man sie zu klein, so erzielt man wiederum eine niedrige SIMD-Effizienz beim Entfernen der Knoten. Ist die maximale Anzahl an Octreetraversierungsschritten zu gross gewählt, so füllt sich der Buffer zunehmend. Dadurch werden potentiell mehr Knoten traversiert, als wegen der Early-Ray-Termination nötig wären. Des Weiteren wird der Buffer dadurch schneller komplett voll werden, wodurch ein Teil der Warpthreads nicht mehr mit dem restlichen Warp traversieren kann. Ist die maximale Anzahl an Raymarchingschritten zu gross gewählt, so ist der Buffer beim Beginn des Raymarchings bei vielen Threads nicht ausreichend gefüllt. Dadurch können diese Threads gemeinsam mit dem restlichen Warp keine oder nicht alle Raymarchingschritte gehen. All das wirkt sich wieder negativ auf die SIMD-Effizienz aus.

4.3.5 Buffered-For-For-With-Flags

Deshalb ist es für die SIMD-Effizienz bei dieser Buffered-For-For-Kombination, ähnlich wie bei dem Speculative-If-Verfahren, wichtig Strategien einzuführen, wann der Buffer durch die Octreetraversierung zu füllen und wann der Buffer durch das Raymarching zu leeren ist. Dies soll über Flags für jeden Warp verwirklicht werden. Diese Flags werden einmal bei jeder While-Mainloop-Iteration gesetzt und danach abgefragt. Dies ist in Abbildung 4.20 in Pseudocode dargestellt.

Folgende Flags existieren:

- **FlagA: Buffer-Underrun-Flag.** Ein Thread setzt diese Flag seines Warps, wenn die im Buffer enthaltenen Knoten zu wenig Raymarchingschritte besitzen, um die maximale Zahl an Raymarchingschritten auszuführen. Ist die Octreetraversierung bereits abgeschlossen, so setzt ein Thread diese Flag allerdings nicht.
- **FlagB: Buffer voll und Buffer-Underrun-Flag.** Es kann vorkommen, dass bei vielen kleinen Knoten der Buffer voll wird, ohne dass die Knoten genügend Raymarchingschritte für eine komplette For-Schleife enthalten. In diesem Fall wird diese Flag gesetzt.

Folgende Regeln für das Raymarching und für das Traversieren gelten:

- Ist keine der beiden Flags gesetzt, wird vom gesamten Warp nur das Raymarching ausgeführt.
- Ist nur FlagA gesetzt, so führt der gesamte Warp nur die Traversierung aus.
- Ist FlagB gesetzt, so führt der gesamte Warp nur das Raymarching aus.

Durch diese Regel wird sicher gestellt, dass die im Buffer befindlichen Knoten genügend Raymarching-Schritte enthalten, damit es bei dem Raymarching selbst zu keiner Warpdivergenz kommt. In zwei Fällen kommt es hierbei jedoch dennoch zur Warpdivergenz:

- Am Schluss, da sich die Zahl der Raymarchingschritte der Warpthreads insgesamt, allerdings meist nur minimal, unterscheidet.
- Bei gesetzter FlagB. Hierbei muss der Buffer geleert werden, obwohl er nicht genügend Schritte für eine komplette Raymarching-For-Schleife beinhaltet, wodurch Warpdivergenz auftritt.

Das Volllaufen des Buffers wird ebenfalls durch die Flags vermieden. Dadurch wird die SIMD-Effizienz beim Octreetraversieren verbessert. Denn sobald der Buffer genügend Raymarchingschritte für eine komplette Raymarching-For-Iteration umfasst, wird dieser geleert. In zwei Fällen kommt es hier auch wieder zur Warpdivergenz:

- Am Schluss, da sich die Zahl der insgesamt traversierten Octreeknoten innerhalb eines Warps, allerdings ebenfalls meist nur minimal, unterscheidet.
- Bei einem vollen Buffer. Denn hier können die Warpthreads nicht mehr bei der kompletten For-Traversierungsschleife mitmachen.

Insgesamt stellen die Flags eine einfache Kommunikationsmöglichkeit innerhalb des Warps dar, anhand welcher der Warp bestimmt, ob es besser ist den Octree zu traversieren oder das Raymarching auszuführen, wodurch die SIMD-Effizienz erhöht wird. Dadurch wird ebenfalls die Abhängigkeit der Performance des Verfahrens von der korrekten Wahl der beiden For-Schleifen-Parametern reduziert. Dennoch bleibt die Performance davon abhängig. So werden bei zu kleinen Parametern die Flags selbst

	Buffered-For(10)-For(40)-	Buffered-For(10)-For(40) With-Flags
SIMD-Traversierung	0.87	0.94
SIMD-Raymarching	0.87	0.90
FPS	30.4	30.7

Abbildung 4.21: **SIMD-Effizienzen der Buffered-For-For-Kombination und der Buffered-For-For-With-Flags-Kombination.**

zu oft gesetzt und abgefragt. Hierdurch entsteht ein zu grosser Overhead. Auch wird bei diesem Verfahren aus dem Grund, dass sich der Buffer nur noch ausreichend füllt damit der gesamte Warp an der kompletten For-Raymarching-Schleife teilnehmen kann, vermieden, dass im Falle der Early-Ray-Termination unter Umständen viele nicht mehr benötigte Octreeknoten traversiert werden. Wählt man allerdings den Parameter der For-Schleifen zu gross, so bleibt dieses Problem bestehen.

Um die Performanceänderungen durch diese Kombination zu überprüfen wurden wieder Messungen durchgeführt. Deren Ergebnisse wurden in die Abbildung 4.21 eingetragen. Anschliessend wurden diese SIMD-Effizienzen wieder, wie in Abbildung 4.22 gezeigt wird, visualisiert. Wie erwartet zeigt sich, dass bei guten Parametern beide Kombinationen in etwa gleichwertig sind. Selbst wenn die Flags die SIMD-Effizienz weiter erhöhen, so erhöht sich die Performance kaum. Eine Ursache hierfür ist, dass die Flags zusätzlichen Rechenaufwand verursachen. Auch sind wiederum die selben Gründe wie im letzten Punkt möglich.

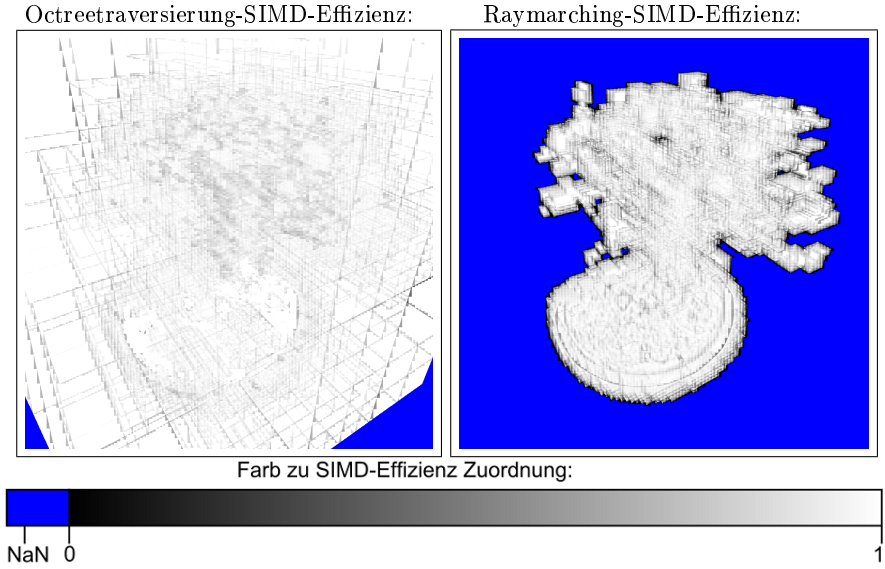


Abbildung 4.22: **Visualisierung der SIMD-Effizienzen der Buffered-For-For-With-Flags-Kombination.** Da bei dieser Version das Überlaufen des Buffers vermieden wird, sind die im Buffered-For-For dunklen Bereiche bei der Octreetraversierung verschwunden. Abgesehen von kleineren Schwankungen, welche insbesondere beim Raymarching am Rand des Bonsais auftreten, sind die beiden Bilder sehr hell, woraus eine hohe SIMD-Effizienz folgt.

5 Implementierung von persistenten Threads

In diesem Kapitel soll das Buffered-For-For-With-Flags um persistente Threads erweitert werden. Diese Idee selbst stammt wieder aus dem Raytracing aus [NVd]. Bei den Versuchen die SIMD-Effizienz durch eine geschickte Kombination von Raymarching und Octreetraversierung zu erhöhen, ist man beim Buffered-For-For-With-Flags, in demjenigen Fall, dass kein Bufferüberlauf eintritt, bereits am Optimum angelangt. Denn hier wird die SIMD-Effizienz nur noch dadurch reduziert, dass die Strahlen der Threads eines Warps auf dem Weg durch die Volumendaten insgesamt eine unterschiedliche Anzahl von Octreeknoten traversieren und unterschiedlich viele Raymarchingschritte ausführen wollen. Ein weiteres Problem ist, dass aus einem ähnlichen Grund die Warps einer Workgroup eine unterschiedliche Laufzeit besitzen. Dies führt dazu, dass die Workgroup am Schluss nur noch wenige unterminierte Warps beinhaltet, wodurch die Achieved-Occupancy reduziert wird. Eine Möglichkeit um die SIMD-Effizienz und die Achieved-Occupancy weiter zu erhöhen stellen persistente Threads dar. Bei diesem Verfahren startet man nicht für jeden Strahl einen Thread, sondern nur für jeden Multiprozessor eine Workgroup. Die Workgroupgrösse wird dabei so gewählt, dass bei dem gegebenen Registerverbrauch die Occupancy maximal ist. Jeder Thread bearbeitet nun nicht mehr einen Strahl bzw. Pixel und terminiert danach, sondern viele Strahlen bzw. Pixel nacheinander. Die Abarbeitung selbst verläuft weitestgehend nach der Buffered-For-For-With-Flags-Kombination. Dieses Verfahren ist in Pseudocode in Abbildung 5.1 dargestellt.

Durch diese Schleifenorganisation wird sicher gestellt, dass der Warp in ziemlich regelmässigen Abständen seinen Threads eine Möglichkeit gibt, sofern sie ihren Strahl abgeschlossen haben, einen neuen Strahl zu erstellen. Dadurch soll erreicht werden, dass alle Warpthreads und alle Warps eine sehr ähnliche Laufzeit besitzen, wodurch wiederum die SIMD-Effizienz und die Achieved-Occupancy erhöht werden soll.

Bei den persistenten Threads kann es jedoch zur Warpdivergenz bei dem Bestimmen der Pixelposition und der Strahlenerstellung selbst kommen. Letzteres ist bei den Verfahren ohne persistente Threads nur der Fall, wenn Teile des Rechtecks einer Workgroup über den Bildschirmrand hinausragen. Da das Bestimmen der Pixelposition und die Strahlenerstellung aufwändig sind, gilt es hier wieder die SIMD-Effizienz zu erhöhen.

Deshalb wird die Entscheidung, ob es sinnvoller für einen Warp ist das Raymarching auszuführen oder den Octree zu traversieren, ebenfalls durch eine For-Schleife mehrmals ausgeführt. Dadurch wird erzielt, dass mehrere Warpthreads bei einem While-Mainloop-Durchlauf gleichzeitig ihren Strahl beenden und einen neuen Strahl erstellen wollen. Möglich wäre es ebenfalls gewesen, wie in [NVd] vorgeschlagen, neue Strahlen nicht obligatorisch, sondern ebenfalls auf Grund einer Warpentscheidung zu erstellen. Dies wurde ebenfalls testweise implementiert und brachte keine Performancebesserung.

Auch musste der Entscheidungsprozess, ob es sinnvoller für einen Warp ist das Raymarching auszuführen oder den Octree zu traversieren, angepasst werden. Denn ein erster Versuch die Buffered-For-For-Flags bei den persistenten Threads zu verwenden scheiterte, da diese zu sehr die SIMD-Effizienz beim Raymarching optimierten. Denn die Flags führen das Raymarching im Optimalfall erst aus, wenn sich bei allen Warp-

```
bool StrahlTerminiert = true;
while(true)
{
  if(StrahlTerminiert)
  {
    if(KeineNeuenPixelMehrVorhanden())
      return;

    Pixelposition = BestimmePixelposition();
    StrahlTerminiert = false;
    ErstelleStrahl(Pixelposition);
  }

  for(int i = 0; i < MaximaleZahlAnEntscheidungen; i++)
      if(EntscheideObWarpTraversierenSollte())
        ForOctreetraversierung();
      else
        ForRaymarching();

  if(ErgebnisSollInDenFrameBufferGeschriebenWerden())
  {
    SchreibeErgebnisInDenFrameBuffer();
    StrahlTerminiert = true;
  }
}
```

Abbildung 5.1: **Pseudocode der persistenten Threads-Verfahren.**

threads genügend Raymarchingschritte dafür im Buffer befinden. Ansonsten wird die Traversierung ausgeführt. Werden nun allerdings neue Strahlen erstellt, während andere Warpthreads die Octreetraversierung bereits abgeschlossen haben und nur noch die Knoten in ihrem Buffer abarbeiten müssen, so traversiert der Warp so lange, bis der Buffer bei allen Threads genügend Raymarchingschritte beinhaltet. Dies kann allerdings sehr lange dauern, insbesonders, da viele Strahlen niemals marschieren werden. Dadurch erzielte man eine Raymarching SIMD-Effizienz von beinahe 100 Prozent, während die SIMD-Effizienz bei der Octreetraversierung lediglich ungefähr 10 Prozent betrug. Das Traversieren des Octrees wurde so kostspielig, dass die Performance sehr gering war.

Die einzelnen Threads bestimmen bei diesem modifizierten Entscheidungsprozess, analog wie bei den Flags, ob sie das Raymarching oder die Octreetraversierung ausführen wollen. Anhand der Entscheidung seiner Threads stellt der Warp nun per Mehrheitsentscheid fest, ob das Raymarching oder das Traversieren sinnvoller ist. Dies hat allerdings immer zur Folge, dass Strahlen, bei welchen nur noch das Raymarching ausgeführt werden muss, unter Umständen sehr lange vom Warp beim Traversieren mitgeschleppt werden, bis andere Threads ebenfalls das Raymarching ausführen wollen. Dadurch wird die SIMD-Effizienz reduziert. Alternativ wurde es ebenfalls versucht das persistente Threads-Verfahren ähnlich wie das Buffered-For-For ohne Entscheidungsprozess zu implementieren. Dies brachte allerdings kaum einen

Unterschied mit sich. Manchmal wurde die Performance situationsabhängig ein wenig besser, manchmal ein wenig schlechter.

Prinzipiell gibt es eine Vielzahl von Möglichkeiten die grundlegende Idee der persistenten Threads zu implementieren. Die verschiedenen im praktischen Teil der Arbeit implementierten Verfahren unterscheiden sich im Wesentlichen nur durch die Methode BestimmePixelposition():

- Interleaved: Bei dieser Version bearbeitet der Thread nacheinander eine Folge von Pixeln, wobei die einzelnen Pixel, welche ein Thread nacheinander bearbeitet, einen regelmässigen Abstand haben. Dieses Verfahren verwendet keinerlei Synchronisation der Threads untereinander. Dadurch werden sich alle Threads mit der Zeit desynchronisieren und zu einem bestimmten Zeitpunkt über ein grosses Volumen verstreut den Octree traversieren oder durch die Volumendaten marschieren. Dies kann sich schlecht auf das Caching und auf die SIMD-Effizienz auswirken.
- Strahlenpool per Warp: Es gibt für jeden Warp einen Strahlenpool. Dieser Strahlenpool entspricht einem Rechteck auf dem Bildschirm. Die Threads des Warps nehmen Strahlen aus dem Pool und bearbeiten diese. Ist der Pool leer wird er mit einem neuen Rechteck aus einem globalen Rechteck-Pool aufgefüllt. Dadurch befinden sich die Pixel der Warpthreads näher beieinander, was potentiell die Warpdivergenz verhindert. Das Problem, dass sich die Warpthreads räumlich gesehen in der Tiefe desynchronisieren und damit zu einem bestimmten Zeitpunkt über grösseres Volumen als ohne persistente Threads verstreut sind bleibt allerdings bestehen. Dadurch kann wiederum die SIMD-Effizienz reduziert werden.
- Strahlenpool per Workgroup: Es gibt für jede Workgroup einen Strahlenpool. Dieser verhält sich analog zu dem Strahlenpool per Warp. Vorteilhaft hierfür sind die höhere Lokalität der verschiedenen Warps des Multiprozessors. Dies erhöht die Cache-Effizienz des Texturcaches und des L1-Caches. Die Pixel der Threads eines Warps liegen allerdings potentiell weiter auseinander als bei dem Strahlenpool per Warp. Dadurch sind die Threads eines Warps zu einem gewissen Zeitpunkt potentiell über ein grösseres Volumen verstreut. Dies kann ebenfalls der SIMD-Effizienz schaden.

Abschliessend soll die Performance und die SIMD-Effizienzen dieser Optimierungen betrachtet werden. Die Messergebnisse wurden in 5.2 eingetragen. Ebenfalls wurden die SIMD-Effizienzen der verschiedenen persistenten Threads-Implementierungen in Abbildung 5.3 und Abbildung 5.4 visualisiert.

Die Messungen zeigten, dass sich in sämtlichen Implementierungen der persistenten Threads weder die Performance noch die SIMD-Effizienz verbessert haben. Dies lässt sich auf folgendes Problem zurückführen: Die Strahlen eines Warps haben beim Buffered-For-For-With-Flags insgesamt bereits eine sehr ähnliche Laufzeit. Ersetzt man nun terminierte Strahlen mit neuen Strahlen, so kommt es vor, dass Threads mit alten Strahlen nicht mehr traversieren können, jedoch noch Knoten in ihren Buffer bearbeiten müssen. Bei neuen Strahlen ist dies genau andersherum. Egal ob der Warp

	Buffered-For(10)-For(40)-With-Flags	Persistent-Threads Interleaved	Persistent-Threads Pool Per Warp	Persistent-Threads Pool Per Workgroup
SIMD-Traversierung	0.94	0.59	0.84	0.82
SIMD-Raymarching	0.90	0.43	0.70	0.66
FPS	30.7	8.7	11.4	19.1

Abbildung 5.2: **SIMD-Effizienzen und FPS der Buffered-For-For-With-Flags Kombination und der persistenten Threads.**

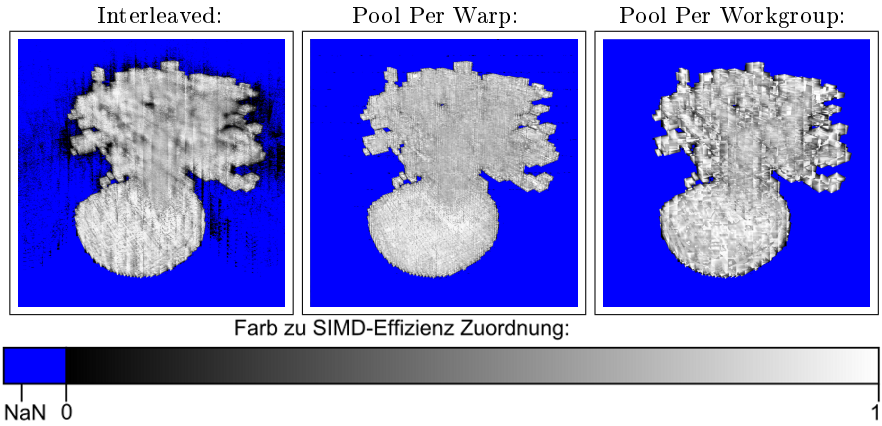


Abbildung 5.3: **Visualisierung der SIMD-Effizienzen beim Raymarching bei den persistenten Threads.**

sich in solchen Fällen für das Octreetraversieren oder für das Raymarching entscheidet, es geht SIMD-Effizienz verloren. Diese Beobachtung deckt sich auch mit [NVd]. So wurde dort ebenfalls analog beobachtet, dass sich beim Raytracing bei Strahlen mit einer bereits hohen SIMD-Effizienz durch persistente Threads nur ein Performanceverlust erzielen lässt.

Um dieses Problem zu vermeiden, müsste man mehrere aktive Strahlen pro Thread zulassen. Alternativ könnte man auch versuchen, wie in [NVd] vorgeschlagen worden ist, nicht mehr die aktiven Strahlen fest einem Thread zuzuweisen, sondern einem Warp 63 aktive Strahlen zu geben. Denn bei dieser Anzahl gibt es immer mindestens 32 Strahlen für das Raymarching oder für die Traversierung. Teilt man nun dynamisch die Strahlen auf die 32 Warpthreads auf, so liesse sich theoretisch eine SIMD-Effizienz bei den beiden Aktionen von beinahe 100 Prozent erzielen.

Auch könnte man versuchen die SIMD-Effizienz bei den persistenten Threads zu erhöhen indem man weitere Regeln, welche bestimmen ob ein Warp traversieren oder

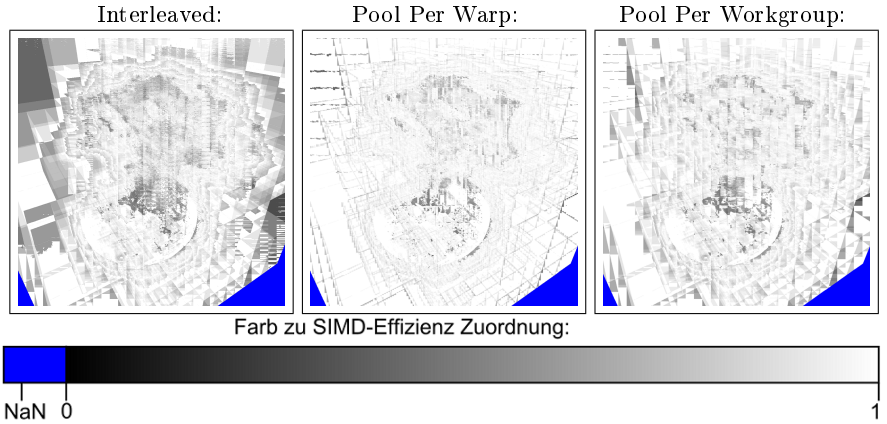


Abbildung 5.4: **Visualisierung der SIMD-Effizienzen bei der Octreetraversierung bei den persistenten Threads.**

marschieren sollte, ausprobiert.

Insgesamt scheint es also, dass die höhere Synchronität der Strahlen innerhalb eines Warps, wie man sie bei den Verfahren ohne persistenten Threads hat, ebenfalls zu einer höheren SIMD-Effizienz und damit zur höheren Performance führt. So könnte man noch eine weitere Optimierung durch persistente Threads ausprobieren, welche ebenfalls in [NVd] vorgeschlagen wird. Bei diesem persistenten Thread-Verfahren versucht man lediglich die Achieved-Occupancy zu erhöhen. Dieses Verfahren verhält sich ähnlich wie das in diesem Kapitel implementierte Verfahren mit dem Strahlenpool per Warp. Jedoch findet das Nehmen aus dem Pool für alle Warpthreads gemeinsam statt und zwar erst dann, wenn alle Warpthreads ihren Strahl abgeschlossen haben. Dadurch würde sich die SIMD-Effizienz jedoch nicht erhöhen.

Die Implementierung weiterer Optimierungen und anschliessende Untersuchungen in diesem Bereich würden allerdings den Umfang dieser Arbeit überschreiten.

6 Untersuchungen

6.1 Allgemeines

In den letzten beiden Kapiteln hat sich die Buffered-For-For-With-Flags-Kombination als bestes Verfahren erwiesen. Dadurch liess sich die FPS beim Bonsai von 10.3 auf 30.7 FPS im Vergleich ohne Octree verbessern. Dadurch wird deutlich, dass man das Verfahren von einer stark ruckelnden in eine flüssige Darstellung durch den Octree überführt hat. Während die letzten beiden Kapitel hauptsächlich den FPS-Wert verwendeten, um einen Eindruck darüber zu vermitteln, wie sehr die Darstellung echtzeitfähig ist bzw. wird, so wird bei den Untersuchungen in diesem Kapitel hauptsächlich der Wert für die Ausführungszeit des Kernels t_{Ausf} bei den Betrachtungen verwendet. Denn beim FPS-Wert besitzt man einen Overhead für das Anzeigen des Bildes, welcher bei der Ausführungszeit nicht mit enthalten ist. Dieser Overhead könnte die Betrachtungen im Folgenden verfälschen.

In diesem Kapitel soll die Ausführung dieses Verfahrens auf der GPU näher untersucht und diskutiert werden. Besonderen Wert wird bei dieser Diskussion darauf gelegt, wie gut das Verfahren die unterschiedlichen Rechenkerne der GPU ausnutzt und auslastet, wie gut das Verfahren skaliert, und wodurch die Performance des Verfahrens letztendlich limitiert wird. Auch wird bei den verschiedenen Rechenkernen ebenfalls abgeschätzt, wie viel Rechenleistung durch sie erzielt worden ist.

Jedoch besitzt diese Buffered-For-For-With-Flags-Kombination viele Parameter, deren Wahl alle soeben genannten Punkte beeinflussen könnte. Deshalb ist es nicht praktikabel für viele Parameterkombination jede Messreihe durchzuführen. Aus diesem Grund wurden wieder die Standardeinstellungen des Programmes, sofern sie für eine bestimmte Messreihe nicht explizit verändert wurden, verwendet. Bei diesen Standardeinstellung sind folgende Gegebenheiten erwähnenswert:

- Die Octreeblattgrösse betrug 8 auf 8 auf 8 Voxel.
- Die Workgroupgrösse betrug 16 auf 16 Workitems bzw. 2 auf 1 Warpkacheln.
- Auch erfolgten die Messungen immer aus der selben Kameraposition.
- Es wurde eine Pre-Classified Transferfunktion verwendet. Zusätzlich wurde die Transferfunktion wieder so eingestellt, dass die niedrigsten 35 Opazitätswerte auf null abgebildet werden, da es sonst auf Grund des Rauschens keine komplett leeren Bereiche in den Aufnahmen gegeben hätte.
- Es wurde keine Deep-Shadow-Map verwendet.
- Die Register des Kernels wurden nicht begrenzt. So benötigt das Buffered-For-For-With-Flags-Kernel auf der verwendeten Geforce 580 GTX die maximal mögliche Zahl von 64 Registern und besitzt zusätzlich leichtes Registerspilling.

Die Messungen werden im Folgenden bei drei verschiedenen Modellen durchgeführt: Einem Sparschwein, einem Bonsai und einem Teddy. Diese sind in Abbildung 6.1 gezeigt.

Für die Untersuchungen wurde folgendes Testsystem verwendet:

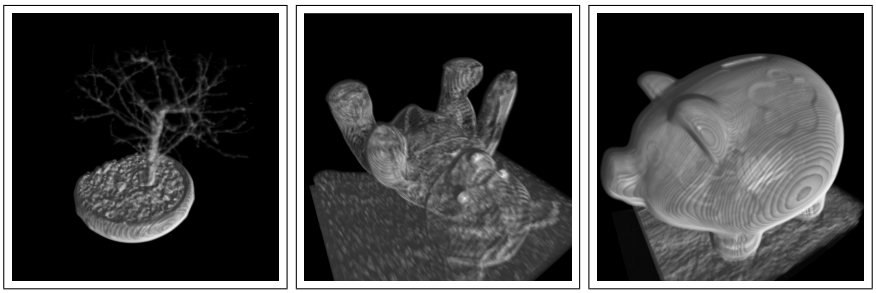


Abbildung 6.1: **Standardansichten vom Bonsai, vom Sparschwein und vom Teddy.**

- CPU: Intel Core i7 860
- Graphikkarte: Phantom Geforce 580 GTX
  - Chiptakt: 783 MHZ (siehe Anmerkung 6.1)
  - Speichertakt: 1008 MHZ
  - Speicherausbau: 3 GB
  - Treiberversion: 301.32
- Betriebssystem: Windows 7

Anmerkung 6.1: Die Geforce 580 GTX besitzt zwei unterschiedliche Taktdomänen innerhalb des Chips. Die Warpscheduler takten dabei nur halb so schnell wie die CUDA-Cores oder die SFUs. Mit Chiptakt ist hier immer der Takt der Warpscheduler gemeint.

6.2 Messungen

6.2.1 Laufzeitabhängigkeit vom Chiptakt

Später soll die Skalierbarkeit des Verfahrens in Abhängigkeit von diversen Hardwareparametern diskutiert werden. Zuerst wird für die spätere Diskussion der Einfluss des Chiptakts auf die Performance gemessen. Der Chiptakt wurde mit Hilfe des Übertaktungsprogammes MSI-Afterburner modifiziert und danach wurde die Laufzeit t_{Ausf} für die verschiedenen Modelle gemessen. Abschliessend wurden die Ergebnisse der Messung in Abbildung 6.2 eingetragen.

6.2.2 Laufzeitabhängigkeit vom Speichertakt

Als Nächstes soll die Abhängigkeit der Laufzeit vom Speichertakt untersucht werden. Diese ist ebenfalls für die Skalierbarkeit des Verfahrens interessant. Hierfür wurde

Chiptakt in MHZ	Bonsai- t_{Ausf} in s	Teddy- t_{Ausf} in s	Sparschwein- t_{Ausf} in s
405	0.0538	0.2044	0.1128
500	0.0438	0.1659	0.0918
600	0.0367	0.1381	0.0773
700	0.0316	0.1193	0.0665
800	0.0282	0.1052	0.0589
900	0.0251	0.0936	0.0525

Abbildung 6.2: **Performancemessung in Abhängigkeit von dem Chiptakt der Graphikkarte.**

Speichertakt in MHZ	Bonsai- t_{Ausf} in s	Teddy- t_{Ausf} in s	Sparschwein- t_{Ausf} in s
1700	0.0287	0.1076	0.0650
1800	0.0286	0.1074	0.0635
1900	0.0286	0.1071	0.0615
2000	0.0285	0.1070	0.0599

Abbildung 6.3: **Performancemessung in Abhängigkeit von dem Speichertakt der Graphikkarte.**

wieder das Übertaktungsprogramm genommen um die Laufzeit in Abhängigkeit des Speichertaktes der Graphikkarte zu messen. Die Ergebnisse dieser Messung wurden in Abbildung 6.3 eingetragen.

Leider war es hier nur möglich einen kleinen Taktbereich zu messen. Denn sobald man versuchte einen Speichertakt von unter 1700 MHZ einzustellen, fiel der Speichertakt sofort auf 200 MHZ ab oder blieb konstant auf 2000 MHZ. Nach oben wird der gemessene Taktbereich durch den Standardtakt begrenzt um keine Beschädigungen an der Graphikkarte zu riskieren.

6.2.3 Laufzeitabhängigkeit von der Multiprozessorzahl

Die Abhängigkeit der Performance von der Multiprozessorzahl ist ebenfalls für die Skalierbarkeit interessant. Leider bietet die Geforce 580 GTX in OpenCL 1.1 keine einfache Möglichkeit die Zahl der für eine Berechnung zu verwendenden Multiprozessoren einzustellen.

Deshalb wurde dafür das Kernel modifiziert. Die Idee hinter den Modifikationen ist, eine Zahl an Multiprozessoren festzulegen, welche abzuschalten sind. Diese Multiprozessoren werden dann im Kernel damit beschäftigt, so lange aktiv darauf zu warten bis die anderen Multiprozessoren das Raycasting komplett abgeschlossen haben. Dadurch können die abzuschaltenden Multiprozessoren nicht am Zeichenprozess teilnehmen.

Zuerst wurden für diese Modifikation des Kernels zwei globale Zähler im Kernel hinzugefügt: Einer für die Zahl der Multiprozessoren, welche abgeschaltet werden sollen, und ein weiterer für die Zahl der Workgroups, welche bereits gezeichnet worden sind.

Das Kernel wird nun mit einer möglichst grossen Workgroupgrösse gestartet, so dass ein Multiprozessor nur maximal eine Workgroup gleichzeitig bearbeiten kann. Zuerst dekrementiert die Workgroup einmalig atomar den Zähler für die abzuschaltenden Multiprozessoren. Aus dem Ergebnis dieser Operation wird bestimmt ob der Multiprozessor abgeschaltet werden soll. Ist dies der Fall so führt die Workgroup kein Raycasting aus, sondern wartet aktiv darauf, bis keine neuen Workgroups mehr zur Verfügung stehen. Dies wird durch Abfragen des globalen Zählers der bereits komplett gezeichneten Workgroups innerhalb einer Schleife realisiert. Damit die abzuschaltenden Multiprozessoren durch das permanente Abfragen dieses globalen Zählers nicht die mit den anderen Multiprozessoren geteilten Ressourcen zu sehr belasten, führen die abzuschaltende Multiprozessoren zwischen zwei Zählerabfragen viele Berechnungen aus. Dadurch wird dieser Zähler nur selten abgefragt. Stehen keine neuen Workgroups mehr zur Verfügung, so terminiert die Workgroup des abzuschaltenden Multiprozessors. Des Weiteren wird, wenn eine Workgroup das Raycasting abgeschlossen hat, der Zähler für die bereits gezeichneten Workgroups atomar um eins erhöht. Dadurch werden die abzuschaltenden Multiprozessoren während des kompletten Rendervorgangs in einer Warteschleife gefangen, und können nicht am Zeichenprozess teilnehmen.

Jedoch hat dieses Verfahren zusätzlich noch das Problem, dass diejenigen Workgroups, welche dazu dienen die Multiprozessoren abzufangen, nicht gezeichnet werden. Dadurch nehmen die für ein Bild benötigten Rechenoperationen mit der Anzahl der abgeschalteten Multiprozessoren ab. Dies würde die Performancemessung verfälschen. Deshalb wurde zuerst beobachtet, dass sich diejenigen Workgroups, welche dazu dienen den Multiprozessor abzuschalten, immer in der unteren linken Bildschirmecke befinden. Ihre genaue Position ist allerdings etwas zufällig. Um dies zu beheben, wird bei allen Workgroups in der unteren linken Bildschirmecke das Raycasting nicht ausgeführt, so dass die Zahl der für das Raycasting durchzuführenden Berechnungen unabhängig von der Zahl der abgeschalteten Multiprozessoren ist. Auf diese Art wurde nun die Laufzeit t_{Ausf} in Abhängigkeit von der Zahl der aktiven Multiprozessoren gemessen. Die Ergebnisse dieser Messungen wurden in Abbildung 6.4 eingetragen.

6.2.4 Laufzeitabhängigkeit von der Workgroupgrösse

Auch ist es von Interesse, wie sich die Performance abhängig von der Workgroupgrösse und der Occupancy verändert. Diese Occupancy lässt sich allerdings nur indirekt über die Workgroupgrösse einstellen. Jedoch bearbeitet ein Multiprozessor, wenn der Registerspeicher und der Shared-Memory dafür ausreichen, immer bis zu acht Workgroups gleichzeitig. Da beim Ressourcenverbrauch des Kernels der Multiprozessor bis zu 16 Warps beherbergen kann, kann man die Performance nicht ohne Workaround für eine Occupancy von eins bis acht Warps messen. Generell sind bei diesem Verfahren nur Workgroupgrössen möglich, welche sich mit Warpkacheln auffüllen lassen. Die Warpkacheln wurden dabei so aneinander gelegt, dass die Workgroups möglichst quadratisch sind. Die Ergebnisse dieser Messung wurden in Abbildung 6.5 eingetragen.

Multiprozessoren	Bonsai- t_{Ausf} in s	Teddy- t_{Ausf} in s	Sparschwein- t_{Ausf} in s
1	0.514227	1.746438	1.036438
2	0.258316	0.873113	0.519166
3	0.172498	0.585156	0.348801
4	0.129662	0.438728	0.262995
5	0.104851	0.355082	0.210871
6	0.086777	0.294927	0.176142
7	0.074442	0.251496	0.150701
8	0.065029	0.221995	0.133806
9	0.060506	0.197657	0.118510
10	0.057341	0.176468	0.107983
11	0.051169	0.160423	0.098621
12	0.047576	0.147007	0.088678
13	0.042805	0.143644	0.081767
14	0.037672	0.126622	0.076017
15	0.035209	0.118409	0.070991
16	0.033067	0.110698	0.066509

Abbildung 6.4: **Performancemessung in Abhängigkeit von der Zahl der Multiprozessoren.**

Warps per Workgroup	Workgroups per Multi-prozessor	Occupancy	Bonsai- t_{Ausf} in s	Teddy- t_{Ausf} in s	Sparschwein- t_{Ausf} in s
1	8	8	0.047566	0.186193	0.102364
2	8	16	0.028598	0.107085	0.060045
3	5	15	0.030038	0.111539	0.062705
4	4	16	0.029282	0.107143	0.061126
5	3	15	0.030968	0.112345	0.065241
6	2	12	0.035824	0.127898	0.075182
7	2	14	0.033651	0.118841	0.069558
8	2	16	0.030486	0.107268	0.062542
9	1	9	0.047381	0.162741	0.097157
10	1	10	0.043955	0.148661	0.089516
11	1	11	0.043096	0.146311	0.087606
12	1	12	0.038322	0.131536	0.078572
13	1	13	0.039034	0.132302	0.078909
14	1	14	0.035269	0.119076	0.071613
15	1	15	0.033831	0.114733	0.068570
16	1	16	0.032513	0.110510	0.065854

Abbildung 6.5: **Performancemessung in Abhängigkeit der Workgroupgrösse.**

Occupancy	Bonsai-t_{Ausf} in s	Teddy-t_{Ausf} in s	Sparschwein-t_{Ausf} in s
1	0.326094	1.195471	0.679808
2	0.168814	0.607141	0.350026
3	0.117403	0.416136	0.243721
4	0.091131	0.320091	0.188705
5	0.076229	0.263786	0.210871
6	0.086777	0.294927	0.157321
7	0.058825	0.200880	0.120851
8	0.120851	0.175616	0.105384
9	0.047379	0.162625	0.097210
10	0.043736	0.148491	0.089495
11	0.043031	0.146273	0.087628
12	0.038339	0.131166	0.078622
13	0.038867	0.132253	0.078949
14	0.035254	0.119263	0.071581
15	0.033632	0.114688	0.068390
16	0.032408	0.110452	0.065609

Abbildung 6.6: **Performancemessung in Abhängigkeit von der Occupancy.**

6.2.5 Laufzeitabhängigkeit von der Occupancy

Ein Workaround, um dennoch die Performancemessungen bei einer Occupancy von einem bis acht Warps durchzuführen, besteht darin den Shared-Memory-Verbrauch soweit zu erhöhen, dass der Multiprozessor nur eine Workgroup gleichzeitig beherbergen kann. Dadurch wird allerdings der L1-Caches des Multiprozessors von 48 kB auf 16 kB verkleinert. Die Warpkacheln wurden bei diesem Benchmark wieder so aneinandergelegt, dass die Workgroup möglichst quadratisch ist. Dies wurde in den Messungen, deren Ergebnisse in Abbildung 6.6 eingetragen worden sind, untersucht.

Interessanterweise ändert sich die Laufzeit in dem Bereich von neun bis 16 Warps, trotz des verringerten L1-Caches, im Vergleich zur vorherigen Messung in Punkt 6.2.4 kaum. Daraus kann man schliessen, dass die Messung kaum durch die Verringerung des L1-Caches verfälscht wird.

6.2.6 Visual-Profiler-Messungen

Nvidia stellt mit Visual-Profiler ein Programm zur Verfügung, mit welchen man die Ausführung eines OpenCL-Programms auf der Graphikkarte näher untersuchen kann. Ein Kernel löst bei der Ausführung auf der Graphikkarte viele verschiedene Hardware-Events, wie zB. Cache-Hit-Events oder Instruktion-Ausgeführt-Events, aus. Der Visual-Profiler fragt diese Events nach der Kernelausführung ab und berechnet aus ihnen Metriken, wie zB. die Cache-Hitrate oder die ausgeführten Instruktionen pro Takt. Aus diesen Metriken kann man sehr gut schliessen, wie ein Kernel bei der Ausführung die Hardware auslastet und ausnutzt. Deshalb wurden die verschiedenen Metriken bei der Ausführung des Programms damit aufgezeichnet. Da Nvidia aus dem Visual-Profiler in der Version 5.0 den OpenCL-Support entfernt hat, wurde

die Version 4.2 verwendet. Jedoch sind in der Version 5.0 neue Metriken über die L2-Cache-Hitrate hinzugekommen. Um diese dennoch zu berechnen, wurden die L2-Cache-Events aufgezeichnet und die L2-Cache-Hitraten durch die in der Hilfe der 5.0 Version angegebenen Formeln berechnet. Die Ergebnisse dieser Messungen wurden in Abbildung 6.7 eingetragen.

Erklärungen der Metriken:

- **Duration:** Ausführungszeit des Kernels. Dies entspricht somit t_{Ausf}.
- **Global-Load-Throughput:** Dies ist diejenige Bandbreite, welche dadurch entsteht, dass die LSUs bei globalen Speicherinstruktionen Daten in die Register des Multiprozessors laden. Also beinhaltet dieser Wert auch die entsprechenden Cache-Bandbreiten bei Cache-Hits.
- **Branch-Efficiency:** Verhältnis von divergenten Branches zu Branches insgesamt. Da bei Predication kein Branching entsteht, ist dies hier nicht enthalten.
- **IPC:** Herausgegebene Instruktionen pro Takt von den Warpschedulern ohne Replays. Die Tatsache, dass bei den IPCs die Replays nicht enthalten sind, folgt auch erst aus den Formeln für Metriken, welche erst in der Hilfe der 5.0er Version stehen.
- **Instruction-Replay-Overhead:** Unter diversen Umständen müssen Speicherinstruktionen mehrmals herausgegeben werden. Dies wird als Replay bezeichnet. Diese Metrik gibt an, welchen Anteil diese Instruktionen an der Gesamtzahl aller von den Warpschedulern herausgegebenen Instruktionen ausmachen. Wie sich dieser Wert genau zusammensetzt ist nicht dokumentiert. Dieser Wert enthält zumindest die Shared-Memory-Replays, Global-Memory-Replays, CUDA-Local-Memory-Replays und die Constant-Memory-Replays. Letztere werden im Visual-Profiler nicht gelistet und entstehen durch die Sequentialisierung durch indexierten Zugriff auf den Konstantenspeicher.
- **Multiprocessor-Efficiency:** Derjenige Anteil der Zeit in der ein Multiprozessor mindestens einen Warp beherbergt, gemittelt über alle Multiprozessoren.
- **Achieved-Occupancy:** Die normierte Achieved-Occupancy.
- **Shared-Memory-Replay-Overhead:** Anteil der mehrmals herausgegebenen Shared-Memory-Befehle wegen Shared-Memory-Bank-Conflicts an der Gesamtzahl aller herausgegebenen Instruktionen.
- **Global-Memory-Cache-Replay-Overhead:** Anteil der mehrmals herausgegebenen globalen Speicher Befehle durch L1-Cache-Misses an der Gesamtzahl aller herausgegebenen Instruktionen. (Siehe Anmerkung 6.2)
- **Local-Memory-Cache-Replay-Overhead:** Anteil der mehrmals herausgegebenen CUDA-lokalen Speicherbefehle durch L1-Cache-Misses an der Gesamtzahl aller Instruktionen. (Siehe Anmerkung 6.2)
- **Warp-Execution-Efficiency:** Dieser Wert entspricht der SIMD-Effizienz in dieser Arbeit.

	Bonsai	Teddy	Sparschwein
Zeiten:			
Duration in ms	28.111	106.7	59.418
Zeit zwischen zwei Kernelaufrufen in ms	3.1	3.1	3.1
Memory:			
DRAM-Read-Throughput in GB/s	3.62	1.46	3.911
DRAM-Write-Throughput in GB/s	9.48	1.79	4.69
Global-Load-Throughput in GB/s	13.67	3.14	8.98
Instruction:			
Branch-Efficiency	0.91	0.84	0.90
IPC	1.147	1.054	1.075
Instruction-Replay-Overhead	0.031	0.009	0.023
Shared-Memory-Replay-Overhead	0	0	0
Global-Memory-Cache-Replay-Overhead	0.002	0	0.001
Local-Memory-Cache-Replay-Overhead	0.01	0.002	0.005
Warp-Execution-Efficiency	0.80	0.76	0.70
Multiprocessor:			
Multiprocessor-Efficiency	1.00	1.00	1.00
Achieved-Occupancy	0.322	0.327	0.323
Cache:			
L1-Global-Hit-Rate	0.70	0.82	0.77
L1-Local-Hit-Rate	0.81	0.91	0.86
L1-L2-Read-Hit-Rate	0.91	0.88	0.83
L2-Total-Read-Hit-Rate	0.96	0.99	0.97
Texture:			
Texture-Cache-Hit-Rate	0.85	0.87	0.85
Texture-Cache-Throughput in GB/s	508	872	695
Texture-L2-Cache-Hit-Rate	0.97	0.99	0.98
L2 Read Events:			
l2 subp0 read sector queries	8043715	4815534	8363306
l2 subp1 read sector queries	8146695	4623663	8539929
l2 subp0 read sector hits	7261635	4045591	6883826
l2 subp1 read sector hits	7397134	4251766	7129192
L2 Texture Events:			
l2 subp0 tex sector queries	36424687	203542837	106687963
l2 subp1 tex sector queries	36292466	203524059	106747511
l2 subp0 tex sector hits	35362236	201483061	104957125
l2 subp1 tex sector hits	35490267	202663576	104859936

Abbildung 6.7: **Visual-Profiler-Messungen.**

- **L1-L2-Read-Hit-Rate:** Die Trefferrate bei Lesezugriffen auf den L2-Cache nach einem L1-Miss. Dies entsteht also sowohl durch den CUDA-lokalen Speicher als auch durch den globalen Speicher.
- **L2-Total-Read-Hit-Rate:** Die Trefferrate bei Lesezugriffen insgesamt. Dies beinhaltet Lesezugriffe durch Texturspeicherzugriffe, globale Speicherzugriffe und CUDA-lokale Speicherzugriffe.
- **Texture-L2-Cache-Hit-Rate:** Die Trefferrate des L2-Caches durch Texturspeicherzugriffe nach einem Texturcache-Miss.
- **Texture-Cache-Throughput:** Die erzielte Bandbreite des Texturcaches.
- **L2-Texture-Events:** Diejenigen Events, die bei L2-Zugriffen durch Texturzugriffe ausgelöst werden.
- **L2-Read-Events:** Diejenigen Events, die bei L2-Zugriffen durch globale oder CUDA-lokale Speicherbefehle ausgelöst werden.

Anmerkung 6.2: Wann genau CUDA-Local-Instruction-Replays und Global-Instruction-Replays auftreten ist nicht genau dokumentiert. In der 4.2er Version sind die Formeln für die Metriken nicht in der Hilfe angegeben. Die Metriken werden allerdings durch einen Tooltip erklärt als:

- **Global-Memory-Replays:**
  "Percentage of instruction issues due to replays for global memory cache misses"
- **Local Memory Replays:**
  "Percentage of instruction issues due to replays for local memory cache misses"

In der 5.0er Version sind die Formeln angegeben:

- **Global-Memory-Replays:**
  "100 * global_load_miss / inst_issued"
- **Local-Memory-Replays:**
  "100 * (local_load_miss + local_store_miss) / inst_issued"

Diese Events global_load_miss, local_load_miss, local_store_miss kann man allerdings in keiner der beiden Versionen aufzeichnen lassen. Laut Greg Smith, einem Nvidia Mitarbeiter auf Stack-Overflow, [GS12] handelt es sich bei den Events "global_load_hits" um "the number of L1 cache hits from global requests". Nimmt man an, dass es sich bei diesen nicht auffindbaren Events um die respektiven L1-Cache-Events handelt, so sind diese im Programm messbar. Aus den Formeln kann man dann dadurch folgern, dass diese Memory Replays bei L1-Cache-Misses auftreten.

	Bonsai	Teddy	Sparschwein
t_{Ausf} in s	0.0281	0.0107	0.0594
RGBA-Samples	99591670	811167286	348934260
A-Samples	131996264	692295536	313926328

Abbildung 6.8: **Messung der Samplerate beim Raycaster.**

	A-Textur	RGBA-Textur
t_{Ausf} in s	1.355	1.388
Textursamples	33554432000	33554432000
Gigasamples /s	24.9	24.2
Texturcachebandbreite in GB/s	1423	1495

Abbildung 6.9: **Messung der Samplerate in einem TMU-limitierten Benchmark.**

6.2.7 Messungen der Samplerate

Um die Auslastung der TMUs zu bestimmen wurde bei allen Modellen die Zahl der genommenen Samples bestimmt. Bei den Messungen wurde unterschieden, dass wegen der Pre-Classified Transferfunktion mehrere 3D-Texturen mit einer unterschiedlichen Anzahl an Kanälen verwendet worden sind:

- Eine einkanälige Textur bzw. A-Textur für den Opazitätswert.
- Zwei vierkanälige Texturen bzw. RGBA-Texturen für die Farbwerte und die Normalen.

Die Messungen ergaben diejenigen Werte, welche in Abbildung 6.8 zu sehen sind.

Zusätzlich wird die maximale Samplerate bei der trilinearen Interpolation benötigt. Diese wurde in einem TMU-limitierten Benchmark für eine RGBA- und A-Textur gemessen und die Ergebnisse der Messung wurden in die Tabelle in Abbildung 6.9 eingetragen.

Abweichungen von der Samplerate im Datenblatt der GPU von 49.41 Gigasamples/s sind darauf zurückzuführen, dass es sich bei dieser Samplerate um die bilineare Interpolation handelt. Denn modifizierte man das Benchmark so, dass statt einer 3D-Textur eine 2D-Textur gesamplet worden ist, so liessen sich ähnliche Sampleraten erzielen.

6.2.8 Messung der SIMD-Effizienzen

Für die spätere Diskussion ist es ebenfalls interessant, wie hoch die abstrakten SIMD-Effizienzen der verschiedenen Modelle bei dem Raymarching und bei der Traversierung sind. Diese wurden wie im letzten Kapitel für alle Modelle gemessen. Die Ergebnisse dieser Messung sind in Abbildung 6.10 zu sehen.

	Bonsai	Teddy	Sparschwein
SIMD-Traversierung	0.938	0.956	0.944
SIMD-Raymarching	0.899	0.952	0.931

Abbildung 6.10: **Messung der SIMD-Effizienzen.**

	Bonsai	Teddy	Sparschwein
Octeetraversierungsschritte	31606977	36826843	42755847
Raymarchingschritte	131996264	692295536	313926328

Abbildung 6.11: **Messung der Traversierungsschritte und der Raymarchingschritte.**

6.2.9 Messung der Raymarchingschritte und Octreetraversierungsschritte

Zu guter Letzt soll noch gemessen werden, wie viele Raymarching- und wie viele Traversierungsschritte bei jedem Modell für jeden Strahl ausgeführt worden sind. Die Ergebnisse dieser Messungen sind in Abbildung 6.11 zu sehen.

6.3 Diskussion der Messergebnisse

6.3.1 Allgemeines

Zu Beginn der Diskussion soll zuerst betrachtet werden, wie viele Raymarchingschritte und Octreetraversierungsschritte, welche in 6.2.9 gemessen worden sind, bei jedem Modell ausgeführt werden. Zur besseren Übersicht wurde die entsprechende Schrittzahlen jeweils auf die kleinste auftretende Traversierungsschrittzahl bzw. Raymarchingschrittzahl normiert. Auch wurde bei jedem Modell berechnet, wie viele Raymarchingschritte pro Octreetraversierungsschritt ausgeführt wurden. Die Ergebnisse dieser Berechnungen sind in 6.12 zu sehen.

Hierbei wird deutlich, dass beim Sparschwein am meisten Octreeknoten traversiert werden müssen. Dies ist auf die Höhlung im Inneren des Sparschweins zurückzuführen. Beim Traveriseren muss nun ein Strahl an der Vorderseite tief in den Octree hinabgehen. Nach der Vorderseite kann der Strahl den Innenraum des Sparschweins ebenfalls

	Bonsai	Teddy	Sparschwein
Normierte Octreetraversierungsschritte	1.00	1.17	1.35
Normierte Raymarchingschritte	1.00	5.25	2.38
Raymarchingschritte pro Octreetraversierungsschritt	4.17	18.8	7.34

Abbildung 6.12: **Traversierungsschritte und Raymarchingschritte.**

nur auf einer tiefen Octreeebene durchlaufen. Dadurch werden viele Iterationen nötig um den hohlen Innenraum zu überspringen. Erst nachdem er die Rückseite des Sparschweins erreicht hat, kann der Strahl von einer tiefen Ebene zurück zur Wurzel traversieren. Beim Teddy sind es mittelmässig viele Octreetraversierungsschritte. Der Teddy ist zwar sehr grob geformt, wodurch der Octree effektiv arbeitet. Allerdings finden beim Teddy selbst am meisten Raymarchingschritte statt. Dadurch umfassen diejenigen Octreeknoten, durch welche marschiert werden muss, das grösste Volumen, wodurch sie auch eine grosse Oberfläche haben. Da ein Strahl bei einer Oberfläche meist tief in den Octree hinabgehen muss, verursacht diese grosse Oberfläche viele traversierte Knoten. Beim Bonsai werden trotz seiner feinen Struktur am wenigsten Knoten traversiert. Dies ist auf das sehr kleine Volumen der durchzumarschierenden Octreeknoten und die damit verbundene kleine Oberfläche zurückzuführen. So werden beim Bonsai auch im Vergleich mit den anderen beiden Modellen am wenigsten Raymarchingschritte ausgeführt.

Während sich die Zahl der traversierten Octreeknoten nur ein wenig zwischen den verschiedenen Modellen unterscheidet, so unterscheidet sich die Zahl der Raymarchingschritte stark. Dies ist, wie soeben bereits erwähnt, hauptsächlich auf das unterschiedliche Volumen derjenigen Octreeknoten, durch welche marschiert werden muss, zurückzuführen. Deshalb ist es gemäss der Unterschiede zu erwarten, dass der Teddy dadurch mit Abstand die grösste Laufzeit hat, während das Sparschwein eine mittelmässige und der Bonsai eine niedrige Laufzeit hat. Dies entspricht soweit auch der im Punkt 6.2.6 gemessenen Laufzeit im Visual-Profiler.

Als Nächstes soll das Verhältnis von Raymarchingschritten pro Octreetraversierungsschritt diskutiert werden. Hierbei gilt es erst einmal zu berücksichtigen, dass das Raymarching und die Octreetraversierung die GPU unterschiedlich auslasten. So werden beispielhaft globale Speicherbefehle nur bei der Octreetraversierung benötigt, während Textursamplebefehle nur beim Raymarching benötigt werden. So werden Modelle mit einem kleinen Verhältnis eher diejenigen Teile der Graphikkarte, welche bei der Ausführung der globalen Speicherbefehle benötigt werden, auslasten, während Modelle mit einem grossen Verhältnis eher diejenigen Teile, welche für die Samplebefehle benötigt werden, auslasten werden. So tritt beim Teddy mit Abstand das grösste Verhältnis auf und gleichzeitig wird bei ihm auch mehr Texturcachebandbreite (gemessen in 6.2.6) verbraucht als bei den anderen beiden Modellen. Der Global-Load-Throughput (ebenfalls gemessen in 6.2.6) ist beim Teddy am niedrigsten. Beim Bonsai verhält es sich genau andersherum. Das Verhältnis ist am kleinsten. Dadurch wird beim Bonsai die geringste Texturcachebandbreite und am meisten Global-Load-Throughput benötigt. Ähnliches lässt sich auch im restlichen Kapitel bei den verschiedenen Auslastungen beobachten. Wird eine Auslastung durch das Raymarching verursacht, so ist sie beim Bonsai am kleinsten, beim Sparschwein mittelmässig und beim Teddy am grössten. Wird eine Auslastung durch die Octreetraversierung verursacht, so verhält es sich genau andersherum. Jedoch würde es zu weit führen auf diese Gegebenheit im Folgenden jedes mal explizit einzugehen.

Des Weiteren lässt sich aus der Zahl der RGBA-Samples und A-Samples (gemessen in Punkt 6.2.7) derjenige Anteil der Raymarchingschritte an allen Raymarchingschritten berechnen, bei welchen man durch nicht komplett durchsichtiges Volumen marschiert ist. Dies wurde in Abbildung 6.13 getan.

	Bonsai	Teddy	Sparschwein
Verhältnis	0.62	0.41	0.44

Abbildung 6.13: **Verhältnis von komplett durchsichtigen Raymarchingschritten zu allen Raymarchingschritten.**

Dadurch wird deutlich, dass trotz Octree durch viel komplett durchsichtiges Volumen marschiert wird. Dies hat als Ursache, dass die durchmarschierten Octreeknoten so gross sind, dass sie viele komplett durchsichtige Voxel umfassen. Eine weitere Ursache ist, dass ein Thread den Buffer der Octreeknoten beim Raymarching nur selten abfragt und deshalb auch oft aus Knoten hinausmarschiert. Die meisten Raymarchingschritte durch komplett durchsichtiges Volumen werden beim Bonsai und beim Sparschwein ausgeführt. Dies ist darauf zurückzuführen, dass wegen den feinen Ästen des Bonsais und dem Hohlraum innerhalb des Sparschweins man mit dem Octree dort den leeren Raum nicht gut entfernen kann. Dahingegen ist der Teddy sehr klobig. Dennoch wurde durch die Transferfunktion viel des flauschigen Plüsches der Füllung aus dem Teddy entfernt, so dass er ebenfalls viele kleine Hohlräume im Inneren besitzt. Diese sind jedoch zu klein um durch den Octree übersprungen zu werden, wodurch das Verhältnis beim Teddy ebenfalls gross ist.

Als Nächstes sollen kurz die gemessenen abstrakten SIMD-Effizienzen bei dem Raymarching und bei der Traversierung (gemessen in Punkt 6.2.8) betrachtet werden und mit der über die gesamte Programmausführung gemittelten SIMD-Effizienz (gemessen in Punkt 6.2.6) verglichen werden. So betragen die abstrakten SIMD-Effizienzen 90 bis 95 Prozent. Durch Branching und Predication, vornehmlich innerhalb der Octreetraversierung und des Raymarchings, reduziert sich die über die gesamte Programmausführung betrachtete SIMD-Effizienz auf 70 bis 80 Prozent. Somit verliert man durch dieses innere Branching und Predication in etwa 10 bis 25 Prozent der SIMD-Effizienz.

Schliesslich soll die in Punkt 6.2.6 gemessene Achieved-Occupancy betrachtet werden. Bei der gewählten Workgroupgrösse von zwei Warpkacheln kann ein Multiprozessor acht Workgroups bzw. 16 Warps des Kernels fassen. Dies resultiert in einer Occupancy von 16. Da laut den Visual-Profiler-Messungen die normierte Achieved-Occupancy bei allen Modellen in etwa 0.325 beträgt und ein Geforce 580 Multiprozessor maximal 48 Warps verwalten kann, beträgt die unnormierte Achieved-Occupancy 15.6 Warps. Dadurch wird deutlich, dass die Achieved-Occupancy nur ein wenig kleiner als die eigentliche Occupancy ist. So wird die Achieved-Occupancy und damit die Performance nur leicht durch die unterschiedliche Warplaufzeit reduziert.

6.3.2 Effizienz der Caches

In diesem Punkt sollen die Effizienzen aller Caches, welche in 6.2.6 gemessen worden sind, betrachtet werden. Hierbei soll besonders auf die verschiedene Caches als auch auf die verschiedenen Speicherbereiche in CUDA eingegangen werden.

So beträgt die L1-Cache Treffer-Rate für den CUDA-lokalen Speicher je nach Modell 80 bis 90 Prozent. Da die CUDA-lokalen Speicherdaten komplett in den L1-Cache pas-

sen sollten, folgt daraus, dass sie durch sich selbst oder durch globale Speicherzugriffe verdrängt werden. Dies könnte man komplett vermeiden, indem man den L1-Cache als Shared-Memory verwendet und sämtliche Objekte, welche in dem CUDA-lokalen Speicher liegen, also den Knotenbuffer und den Octreestack, im Shared-Memory abspeichert.

Die Lese-L1-Cache-Hitrate vom globalen Speicher ist mit ca. 70 bis 83 Prozent je nach Modell bereits gut. Nimmt man an, dass die Trefferrate für globale Lese-Speicherzugriffe beim L2-Cache in etwa der durchschnittlichen Trefferrate der Lese-L2-Cache-Anfragen bei einem L1-Cache-Miss entspricht, so fängt der L2-Cache weitere 88 bis 90 Prozent aller globalen Speicherzugriffe ab. Dadurch gehen lediglich 3 bis 4 Prozent aller globalen Speicherzugriffe auf den RAM über.

Bei den Texturspeicherzugriffen herrscht ein ähnliches Bild. Der Texturcache fängt bereits je nach Modell 85 Prozent bis 87 Prozent aller Zugriffe ab. Der bei einem Texturcache-Miss zunächst abgefragte L2-Cache besitzt bei diesen Texturzugriffen eine Trefferrate von 97 bis 99 Prozent. Deshalb wird nur bei 0.4 bis 0.9 Prozent aller Texturspeicherzugriffen der RAM abgefragt.

Dadurch wird deutlich, dass der L2-Cache bei diesem Verfahren wichtig ist, um die benötigte Speicherbandbreite zu reduzieren. So beträgt die Trefferrate des L2-Caches bei Lesezugriffen insgesamt je nach Modell 97 bis 99 Prozent. Durch Betrachtung der Cache Events wird ebenfalls deutlich, dass der L2-Cache wesentlich öfter wegen Texturcache-Misses als wegen L1-Cache-Misses durch Lesezugriffe abgefragt wird und gleichzeitig bei den Texturspeicherzugriffen eine hohe Hit-Rate besitzt. Deshalb ist der L2-Cache vor allem wegen den Texturspeicherzugriffen wichtig. Letztendlich lässt sich durch die Betrachtungen insgesamt sagen, dass das untersuchte Verfahren so lokal auf den Speicher zugreift, dass es die Caches sehr gut ausnutzt. Dies ist jedoch unter anderem auch auf die kleine Raymarchingschrittweite und auf die hohe Auflösung zurückzuführen. Bei kleineren Auflösungen und höheren Raymarchingschrittweiten wären die Speicherzugriffe weniger lokal, wodurch die Cache-Effizienz reduziert werden würde.

In den Messreihen 6.2.4 und 6.2.5 hat man in dem Occupancy-Bereich von 9 bis 16 Warps die selbe Messung nur mit unterschiedlichen L1-Cache-Grössen vorgenommen. So betrug in der ersten Messreihe die L1-Cachegrösse 48 kB und in der zweiten Messreihe 16 kB. Obwohl die L1-Cachegrösse auf ein Drittel schrumpfte, änderte sich die Performance kaum. Daraus kann man folgern, dass die L1-Cachegrösse nur einen sehr geringen Einfluss auf die Performance nimmt. Interessanterweise ist das Kernel mit dem kleineren L1-Cache oft ein klein wenig schneller als das Kernel mit dem grösseren L1-Cache. Da die Unterschiede in der Laufzeit alle kleiner als $200\mu s$ sind, ist dies vermutlich auf Messungenauigkeiten zurückzuführen. Des Weiteren besitzen beide Kernels einen leicht unterschiedlichen Quellcode. So wäre es ebenfalls möglich, dass der Compiler beide Kernels leicht unterschiedlich optimiert, wodurch solche kleinen Laufzeitunterschiede entstehen könnten.

6.3.3 Auslastung und Ausnutzung der Graphikkarte

6.3.3.1 Allgemeines

In diesem Punkt soll die Auslastung der verschiedenen Bestandteile der Graphikkarte, die Ausnutzung der Rechenwerke der GPU und die erzielte Rechenleistung diskutiert werden.

Eine hohe Auslastung soll hierbei angeben, ob ein Bestandteil der Graphikkarte die Performance limitiert. Da sich die Bestandteile der Graphikkarte in ihrem Ausführungsverhalten unterscheiden, ist es sinnvoll, die Definition der Auslastung selbst von Bestandteil zu Bestandteil der Graphikkarte unterschiedlich zu wählen. So ist es beispielhaft bei den CUDA-Cores und den SFUs sinnvoll, die Auslastung als denjenigen Prozentsatz der Zeit anzugeben, bei dem sie von einem Warp belegt werden. Da die TMUs von mehreren Warps gleichzeitig belegt werden können, ist diese Definition hier weniger sinnvoll. Deshalb wird bei den TMUs das Verhältnis zwischen erzielter und maximaler Textursamplerate als Auslastung gewählt. Bei Speicher- oder Cachebandbreiten ist es sinnvoll das Verhältnis von erzielter zu maximaler Bandbreite als Auslastung zu wählen. Generell lässt sich die Auslastung nur gemittelt über die gesamte Programmausführungszeit und über alle Multiprozessoren bestimmen.

Jedoch kann während der Ausführungszeit eines Programmes die Auslastung der Bestandteile der GPU stark variieren. Auch können die Ressourcen verschiedener Multiprozessoren sehr unterschiedlich ausgelastet werden, so dass die Performance dieser Multiprozessoren jeweils von einer anderen Ressource limitiert wird. Insbesondere letzteres kann bei sehr inhomogenen Kernels wie bei dem Volumenraycaster mit Octree der Fall sein. So lasten die Traversierung und das Raymarching die Ressourcen eines Multiprozessors sehr unterschiedlich aus. Traversiert ein Warp durch den Octree, so beinhaltet jeder Octreetraversierungsschritt viele CUDA-Core-Befehle, zwei SFU-Befehle, einen globalen Speicherbefehl und wenige CUDA-lokale Speicherbefehle. Marschiert ein Warp durch durchsichtige Volumendaten, so beinhaltet jeder Raymarchingschritt ein Textursample-Befehl und sehr wenige CUDA-Core-Befehle. Marschiert man durch volles Volumen, so benötigt man für jeden Schritt mittelmässig viele CUDA-Core-Befehle, drei Textursample-Befehle und zwei SFU-Befehle. Dadurch wird deutlich, dass ein Warp über einen kleineren Zeitraum betrachtet, je nachdem was er gerade dabei ist zu tun, einen Multiprozessor sehr unterschiedlich auslasten kann.

Deshalb sind, wenn alle Warps eines Multiprozessors gerade den Octree traversieren wollen, die TMUs des Multiprozessors untätig. In diesem Fall könnten theoretisch nur die übrigen Ressourcen, wie zB. die Befehlsrate der Warpscheduler, limitieren. Wollen jedoch alle Warps eines Multiprozessors gerade marschieren, so werden die TMUs so stark belastet, dass sie die Performance ebenfalls limitieren können. Limitieren die TMUs die Performance zu sehr, so werden die anderen Ressourcen nur wenig ausgelastet. Die Auslastung der Warpscheduler wäre also gering. Angenommen die Limitierung beim Raymarching durch die TMUs und die Limitierung bei der Octreetraversierung durch die Warpscheduler treten ein. Dann kann es vorkommen, obwohl die Warpscheduler und die TMUs die Performance eines Multiprozessoren jeweils zeitweise limitieren die Auslastung beider Rechenwerke über alle Multiprozessoren und über die Ausführungszeit gemittelt nur durchschnittlich ist.

	Bonsai	Teddy	Sparschwein
DRAM-Read-Throughput in GB/s	3.62	1.46	3.91
DRAM-Write-Throughput in GB/s	9.48	1.79	4.69
DRAM-Throughput in GB/s	13.1	3.25	8.60
Auslastung der Speicherbandbreite	0.068	0.017	0.045

Abbildung 6.14: **Auslastung der Speicherbandbreite.**

Auch bei homogenen Programmen, welche eine Ressource im zeitlichen Mittel nur mittelmässig oder gering auslasten, kann es zu einem bestimmten Zeitpunkt sein, dass diese Ressource die Performance limitiert. So sei beispielhaft ein Kernel genannt, bei dem die Warp-Scheduler im Mittel nur 1 IPC ihrer maximal 2 IPC herausgeben. Dennoch kann es oft vorkommen, dass bei einem bestimmten Takt 3 oder mehr Warps bereit werden, den nächsten Befehl auszuführen. In diesem Fall würde die maximale IPC-Rate nicht ausreichen und es würde Performance verloren gehen.

Deshalb folgt zwar daraus, dass wenn die Auslastung eines bestimmten Graphikkartenbestandteils nahezu 100 Prozent ist, dieser Bestandteil limitiert. Jedoch lässt sich im Gegenzug aus einer geringen Auslastung eines Bestandteils nicht folgern, dass durch diesen Bestandteil die Performance nicht limitiert wird. Lediglich bei einer sehr geringen Auslastung ist eine Limitierung sehr unwahrscheinlich. Sofern eine bestimmte Ausführungseinheit für einen Befehl trotz niedriger Auslastung die Performance limitiert, lässt sich dieser Effekt durch eine höhere Occupancy reduzieren. Denn dadurch besitzen die Warpscheduler einen grösseren Befehlspool, den sie als Nächstes in Auftrag geben können. Dadurch sind sie seltener darauf angewiesen, dass sie als Nächstes die Befehle von bestimmten Ausführungseinheiten in Auftrag geben müssen.

Zusätzlich wird für jedes Rechenwerk eine Ausnutzung definiert. Diese Ausnutzung gibt an, welchen Prozentsatz der maximalen Performance des Rechenwerks das Programm im Endeffekt ausnutzt. So kann zum Beispiel die SIMD-Effizienz die Ausnutzung der CUDA-Cores oder der SFUs reduzieren.

6.3.3.2 Auslastung der Speicherbandbreite

Die maximale Speicherbandbreite der verwendeten Graphikkarte beträgt 194.0 GB/s. Diese berechnet sich aus:

$$\begin{aligned} Speicherbandbreite_{max} =& Busbreite * ÜbertrageneBitsPerTakt * RAM\text{-}Takt \\ =& 384Bit * 4\frac{Bit}{Takt} * 1008MHZ \\ =& 194GB/s \end{aligned}$$

Dadurch lässt sich über die Summe der im Punkt 6.2.6 gemessenen Schreib- und Lese-Speicherbandbreite die prozentuale Ausnutzung der Speicherbandbreite berechnen. Die Ergebnisse dieser Rechnung sind in Abbildung 6.14 gezeigt.

Da die mittlere prozentuale Auslastung mit 1.7 bis 6.8 Prozent sehr niedrig ist, limitiert die Speicherbandbreite nicht die Performance. Diese niedrige Lesebandbreite ist vor allem der guten Cache-Effizienz zu verdanken.

	Bonsai	Teddy	Sparschwein
IPC	1.15	1.05	1.08
Instruction Replay Overhead	0.03	0.009	0.023
Auslastung	0.59	0.53	0.55

Abbildung 6.15: **Auslastung der Warpscheduler**

Interessanterweise wird bei allen Modellen wesentlich mehr Schreibbandbreite als Lesebandbreite benötigt, obwohl der Algorithmus erwartungsgemäss viel aus dem RAM durch die Textur- und Octreezugriffe lesen und nur die Farbe des Strahls am Schluss in den RAM der Graphikkarte schreiben sollte. Diese wider Erwarten hohe Schreibbandbreite ist auf die Schreiboperationen in den CUDA-lokalen Speicher zurückzuführen. Wie genau diese Schreibbandbreite zu Stande kommt ist allerdings unklar, da Nvidia für genaue Erklärungsversuche das Cache-Protokoll seiner GPUs nicht ausreichend dokumentiert.

Diese Schreibbandbreite liesse sich senken, indem man stattdessen den Knotenbuffer und den Octreestack in den Shared-Memory verlegen würde. Auch könnte man das Registerspilling vermeiden indem man ebenfalls Daten in den Shared-Memory auslagert.

6.3.3.3 Auslastung der Warp-Scheduler

Die Geforce 580 GTX besitzt zwei Warpscheduler. Jeder der beiden Warpscheduler gibt, sofern es möglich ist, jeden Takt einen Befehl für einen Warp heraus. Dadurch lassen sich insgesamt maximal 2 IPC erzielen. Will man die Auslastung der Warpscheduler aus den in Punkt 6.2.6 durch den Visual-Profiler gemessenen IPC berechnen, so muss man zusätzlich berücksichtigen, dass dieser Wert nicht die ebenfalls in Punkt 6.2.6 gemessenen Replays enthält. Deshalb gilt hier für die Auslastung:

$$\begin{aligned} Auslastung &= \frac{IPC}{(1 - InstructionReplayOverhead) * IPC_{max}} \\ &= \frac{IPC}{(1 - InstructionReplayOverhead) * 2} \end{aligned}$$

Diese wird in der Tabelle 6.15 gezeigt.

So ist die mittlere Auslastung der Warpscheduler mit 53 bis 59 Prozent eher mittelmässig. Dies deutet darauf hin, dass die Occupancy zu niedrig ist um die GPU gut auszulasten, oder dass irgendein Bestandteil der GPU die Performance bereits limitiert. Bei der mittelmässigen Auslastung kann es dennoch vorkommen, dass die Performance eines Multiprozessors zeitweise durch die Befehlsrate der Warpscheduler limitiert wird. Allerdings betragen die Instruction-Replays je nach Modell 0.9 bis 3.1 Prozent. Somit könnte man man die Auslastung durch Vermeidung der Replays reduzieren, um eine etwaige Limitierung durch die Befehlsrate zu vermeiden:

- Man benötigt für die Octreetraversierung diverse Konstanten, welche im Konstantenspeicher liegen und auf welche indexiert zugegriffen wird. Hierunter fal-

len zum Beispiel der Startindex für eine Ebene im globalen Octreebuffer und die Grösse der Ebene. Da der Index für den Zugriff für alle Warpthreads meist nicht identisch ist, wird dieser Zugriff sequentialisiert, wodurch Replays entstehen. Dies könnte man lindern, indem man diese Konstanten in dem Shared-Memory abspeichert.

- Um die globalen Speicherreplays zu vermeiden, besitzt die Geforce 580 GTX prinzipiell die Möglichkeit, globale Speicherzugriffe nicht in den L1-Cache aufzunehmen. Damit würde auch vermieden werden, dass man durch globale Speicherzugriffe den CUDA-lokalen Speicher aus dem L1-Cache verdrängt. Dadurch liessen sich ebenfalls die CUDA-lokalen Speicherreplays reduzieren. Leider stellt Nvidia unter OpenCL diese Funktionalität nicht zur Verfügung.
- Auch könnte man die CUDA-lokalen Speicherreplays vermeiden, indem man stattdessen, wie im letzten Punkt vorgeschlagen, den Octreestack und den Knotenbuffer in den Shared-Memory verlegt.

6.3.3.4 Auslastung und Ausnutzung der CUDA-Cores

Leider gibt es bei Nvidia keine einfache Möglichkeit die durch die CUDA-Cores erzielte Zahl an Integer- und Floatingpoint-Operationen zu bestimmen. Deshalb soll hier eine einfache Schätzung mit Hilfe der im Punkt 6.2.6 durch den Visual-Profiler gemessenen IPC genügen. Für genauere Schätzungen müsste man die verschiedenen Befehle im Assembly des Programms abzählen und zusätzlich einen aufwändigen Simulator schreiben. Dies würde allerdings den Umfang dieser Arbeit überschreiten. Die CUDA-Cores arbeiten zwei unterschiedliche Befehle für je zwei verschiedene Warps jeden Takt ab. Dadurch reicht die Befehlsrate der Warpscheduler gerade dafür aus um die CUDA-Cores auszulasten, wodurch die CUDA-Cores die Performance nur in denjenigen Fall limitieren können, in dem ebenfalls die Befehlsrate limitiert. Unter der Annahme, dass sämtliche herausgegebenen Befehle an die CUDA-Cores gehen, so folgt für deren Auslastung:

$$\begin{aligned} Auslastung =& \frac{IPC_{gemessen}}{IPC_{max}} \\ =& \frac{IPC_{gemessen}}{2} \end{aligned}$$

Aus dieser Auslastung lässt sich nun mit Hilfe der im Visual-Profiler gemessenen SIMD- bzw. Warp-Effizienz die Ausnutzung berechnen:

$$Ausnutzung = SIMD\text{-}Effizienz * Auslastung$$

Über den theoretischen Spitzenwert der CUDA-Cores von 1604 GFLOPS-FMA (Fused multiply-add) lässt sich nun die tatsächlich erzielte Performance schätzen. Dieser Spitzenwert berechnet sich aus:

$$\begin{aligned} FLOPS_{max} =& FMAFaktor * FLOPsPerTakt * CUDACoreTakt * CUDACoreAnzahl \\ =& 2 * \left(1\frac{FLOP}{Takt}\right) * (783MHZ) * 512 \\ =& 1604GFLOPS \end{aligned}$$

	Bonsai	Teddy	Sparschwein
IPC	1.15	1.05	1.08
SIMD-Effizienz	0.89	0.76	0.81
Auslastung	0.57	0.53	0.54
Ausnutzung	0.46	0.40	0.43
Erzielte Performance in GFLOPS	723	637	686

Abbildung 6.16: **Auslastung und Ausnutzung der CUDA Cores.**

Da die Spitzenleistung der CUDA-Cores nur bei FMA erreicht wird, wird die Ausnutzung und die erzielte Rechenleistung zusätzlich dadurch reduziert, dass nicht alle im Programm enthaltenen Befehle FMA sind. Welcher Anteil der Befehle FMA ist, lässt sich jedoch schlecht abschätzen. Generell besitzt das Programm wohl beim Raymarching einen hohen FMA-Anteil, während es bei der Octreetraversierung wohl einen niedrigen FMA-Anteil besitzt. Da bei allen Modellen die Zahl der Raymarchingschritte ein Vielfaches der Octreetraversierungsschritte beträgt wird angenommen, dass viele der Befehle FMA sind. Deshalb wird der Anteil an nicht FMA-Befehlen vernachlässigt. Dadurch ergibt sich für die Ausnutzung und die erzielte Rechenleistung der verschiedenen Modelle diejenige Tabelle, welche in Abbildung 6.16 zu sehen ist.

Die Auslastung, Ausnutzung und die erzielte Performance der CUDA-Cores sind somit gering. Dieser Wert stellt allerdings wegen der Vereinfachung, dass sämtliche Befehle an die CUDA-Cores gehen und FMA-Befehle sind, eine Obergrenze dar. Die tatsächliche Auslastung der CUDA-Cores ist deshalb noch geringer. Die geringe erzielte Performance liegt interessanterweise nicht nur an der stark optimierten SIMD-Effizienz, sondern wird hauptsächlich durch den niedrigen IPC-Wert verursacht.

6.3.3.5 Auslastung der SFUs

Über die genaue Ausnutzung und Auslastung der SFUs kann man leider ohne weitreichende Untersuchungen wenig aussagen. Dennoch ist deren prozentuale zeitliche Belegtheit durch einen Warp interessant, um grob abschätzen zu können, ob diese Ausführungseinheiten die Performance limitieren.

Die SFUs benötigen vier Takte um einen Befehl für einen Warp abzuarbeiten. Um volle zeitliche Belegtheit bzw. Auslastung zu erreichen, würden sie 0.25 IPC der Warpscheduler benötigen. Da die Warpscheduler bei dem Algorithmus selbst, gemäss der Visual-Profiler-Metriken aus Punkt 6.2.6, etwa nur 1.1 IPC herausgeben, müsste in diesem Fall circa jeder vierte Befehl ein SFU Befehl sein. Da im Programm allerdings nur sehr wenig SFU-Befehle verwendet werden, kann man ausschliessen, dass die SFUs die meiste Zeit belegt sind und dass dadurch die Performance limitiert wird.

6.3.3.6 Auslastung der LSUs

Ähnlich fällt es schwer über die Auslastung der LSUs genaue Aussagen zu treffen. Da das Programm aber insgesamt nur wenige CUDA-lokale und globale Speicherbefehle verwendet, ist es wahrscheinlich, dass ihre Auslastung nur sehr gering ist und damit die Performance nicht durch sie limitiert wird.

	Bonsai	Teddy	Sparschwein
t_{Ausf} in s	0.0281	0.0107	0.0594
RGBA-Samples	99591670	811167286	348934260
A-Samples	131996264	692295536	313926328
Erzielte Samplerate in Gigasamples /s	8.23	14.2	11.2
Auslastung	0.34	0.59	0.46

Abbildung 6.17: **Auslastung der TMUs.**

6.3.3.7 Auslastung und Ausnutzung der TMUs

Da man die TMUs ebenfalls für Berechnungen verwendet, ist es auch sinnvoll zu bestimmen, wie sehr sie ausgelastet werden, ob das Programm durch die Sampleleistung der TMUs limitiert wird, und welche Rechenleistung der Algorithmus durch sie erzielt. Aus dieser erzielten Rechenleistung kann man dann berechnen, welchen Prozentsatz der maximalen Rechenleistung der TMUs der Algorithmus ausnutzt.

Als Erstes soll bestimmt werden, wie gut der Algorithmus die Sampleleistung der TMUs auslastet, um festzustellen ob der Algorithmus durch diese Sampleleistung limitiert wird. Aus den Messungen des Texturbenchmarks in Punkt 6.2.7 folgt, dass es für die Kosten des Texturbefehls nahezu egal ist, ob eine vierkanälige Textur oder eine einkanälige Textur gesamplet werden soll. So beträgt die maximal mögliche Samplerate der GPU in etwa 24 Gigasamples/s bei trilinearer Interpolation. Um die erzielte Samplerate zu berechnen werden die in Punkt 6.2.7 gemessenen Samples der einkanäligen Textur und der vierkanäligen Texturen aufaddiert und durch die Ausführungszeit t_{Ausf} geteilt. Dies wird anschliessend in die Relation zur maximalen Samplerate gesetzt. Die Ergebnisse sind in der Tabelle in Abbildung 6.17 zu sehen.

Daraus folgt, dass der Algorithmus im zeitlichen Mittel nur je nach Modell von 34 bis 59 Prozent der maximalen Samplerate bei der Programmausführung erzielt. Dies könnte als Ursache haben, dass die TMUs auf Grund von Speicherbandbreite oder Cachebandbreite nicht ihre volle Samplerate entfalten können. In Punkt 6.3.4 folgt aber, dass die Bandbreite des L2-Caches und des RAMs die Performance nicht oder nur sehr wenig limitieren. So wäre es dennoch möglich, dass die TMUs nicht ihre volle Performance entfalten können, weil die Bandbreite des Texturcaches limitiert. Es scheint jedoch sehr unwahrscheinlich, dass das Zugriffsmuster so ungünstig ist, dass die Texturcachebandbreite nicht für das lokale Samplen einer Textur ausreicht. Da die TMUs wie in Punkt 6.3.3.1 jedoch zeitlich und zwischen verschiedenen Multiprozessoren sehr unterschiedlich ausgelastet werden, ist eine Limitierung selbst bei der mittelmässigen Auslastung durch sie möglich.

Eine einfache Möglichkeit die Auslastung der TMUs zu senken ohne die Ausnutzung zu reduzieren, wäre es die Opazität nicht in einer eigenen Textur, sondern in dem A-Kanal einer der beiden RGBA-Texturen abzuspeichern. Denn auf diese Weise müsste man pro Raymarchingschritt ein Textursample weniger nehmen und hätte zusätzlich keine höheren Kosten, da das Samplen einer RGBA- und einer A-Textur in etwa gleich teuer ist.

Nun soll erörtert werden welche Rechenleistung sich durch die TMUs maximal erzielen

lässt und welche Rechenleistung vom Algorithmus erzielt worden ist. Anschliessend soll daraus die Ausnutzung berechnet werden. Um die erzielte Rechenleistung zu bestimmen muss zuerst betrachtet werden, welche Rechenoperationen eine TMU bei der trilinearen Interpolation durchführen muss. Da es nicht dokumentiert ist, wie die TMUs genau arbeiten, schätzt man die Rechenleistung der TMUs dadurch, wie viele Rechenoperationen man benötigen bzw. durchführen würde, wenn man die TMU in Software implementieren würde:

1. Umrechnen der Sampleposition von normalisierten Texturkoordinaten in unnormalisierte Texturkoordinaten ohne Offset. Dieser Schritt wird hier vernachlässigt. Denn einerseits reduziert es dank Hardwareimplementierung die Performance der TMU nicht und andererseits würde man, wenn man die trilineare Interpolation per Software implementieren würde, diese Softwareinterpolation direkt mit unnormalisierten Texturkoordinaten ohne Offset ansteuern.

2. Berechnen der Ecken der Boundingbox des Interpolationswürfels aus der Sampleposition in unnormalisierten Texturkoordinaten ohne Offset:

   $$\mathbf{C} = ceil(\mathbf{Sampleposition})$$
   $$\mathbf{F} = floor(\mathbf{Sampleposition})$$

   Dieser Schritt kostet 6 FLOPs.

3. Die Berechnung der benötigten Speicheradressen aus $\mathbf{C}$ und $\mathbf{F}$ für die Eckpunkte des Interpolationswürfels. Dies ist potentiell stark davon abhängig, wie genau die Textur im Speicher liegt und wird, da dies ebenfalls nicht dokumentiert ist, ignoriert. Zusätzlich werden die Daten des Interpolationswürfels für jeden Farbkanal der Textur aus dem Speicher angefordert:

   $$TB_0 = Textur(\mathbf{F}.x, \mathbf{F}.y, \mathbf{F}.z)$$
   $$TB_1 = Textur(\mathbf{F}.x, \mathbf{F}.y, \mathbf{C}.z)$$
   $$....$$
   $$TB_7 = Textur(\mathbf{C}.x, \mathbf{C}.y, \mathbf{C}.z)$$

4. Da die Daten im Speicher als Byte vorliegen müssen sie in Floats konvertiert werden:

   $$TF_0 = (float)TB_0$$
   $$TF_1 = (float)TB_1$$
   $$....$$
   $$TF_7 = (float)TB_7$$

   Dies kostet 8 FLOPs

5. Die Berechnung der Faktoren für die Gewichtungen der trilinearen Interpolation:

   $$\overrightarrow{SF} = \mathbf{Sampleposition} - \mathbf{F}$$
   $$\overrightarrow{SC} = \mathbf{C} - \mathbf{Sampleposition}$$

Dieser Schritt kostet 6 FLOPs.

6. Die Berechnung der Gewichtungen der trilinearen Interpolation:

$$
\begin{aligned}
G_0 =& \overrightarrow{SC}.x * \overrightarrow{SC}.y * \overrightarrow{SC}.z \\
G_1 =& \overrightarrow{SC}.x * \overrightarrow{SC}.y * \overrightarrow{SF}.z \\
& .... \\
G_7 =& \overrightarrow{SF}.z * \overrightarrow{SF}.y * \overrightarrow{SF}.z
\end{aligned}
$$

   Da bei jeweils zwei aufeinanderfolgenden G_i zwei Faktoren identisch sind, braucht man für jedes zweite G_i nicht 3 sondern 2 FLOPs. Dadurch benötigt man für die acht G_i insgesamt 20 FLOPs.

7. Für jeden Farbkanal die trilineare Interpolation selbst:

$$InterpolierterWert_{unnormiert} = \sum_{i=0}^{7} G_i * TF_i$$

   Dieser Schritt kostet 16 FLOPs pro Farbkanal.

8. Da die Werte TF_i alle im Bereich von 0 bis 255 sind und die trilineare Interpolation der TMU bei Byte-Texturen jedoch die Bytewerte auf das Intervall 0 bis 1 abbildet, muss man den interpolierten Wert noch durch 255 teilen:

$$InterpolierterWert = \frac{InterpolierterWert_{unnormiert}}{255}$$

   Dieser Schritt kostet 1 FLOP pro Farbkanal.

Aufsummiert ergeben sich 57 FLOPs bei einer A-Textur und 105 FLOPs bei einer RGBA-Textur. Multipliziert man dies mit der gemessenen maximalen Samplerate aus dem Texturbenchmark so ergeben sich 1420 GFLOPS bei einer A-Textur und 2530 GFLOPS bei einer RGBA-Textur. Letzteres entspricht somit in etwa der maximalen Rechenleistung der TMUs bei der trilinearen Interpolation.

Nun soll betrachtet werden, welche Rechenleistung man bei dem Raycaster effektiv durch die TMUs erzielt. So sind alle Schritte beim Sampling der Opazitätstextur noch sinnvoll, weshalb man durch jedes Opazitätstextursample 57 FLOPs erzielt. Beim Sampling der beiden RGBA-Texturen, also der Normalen und Opazitätstextur, wurden die Schritte zwei, fünf und sechs jedoch zuvor schon für die selbe Sampleposition beim Sampling der Opazitätstextur berechnet. Dadurch führt die TMU diese Berechnungen redundanterweise durch, weshalb sie nicht zur effektiv ausgenutzten Rechenleistung beitragen. Beim siebten Schritt werden beim Sampling einer RGBA-Textur immer vier Kanäle interpoliert. Der Algorithmus selbst benötigt jedoch nur die interpolierten Werte von drei Kanälen. Der überschüssige Wert wird verworfen. Deshalb sind die einzigen effektiven Rechenoperationen, welche bei jedem Normalen und Farbtextursample ausgeführt werden, das Konvertieren von den Bytewerten in Floatwerte aus dem vierten Schritt, die drei Kanalinterpolationen, die im Schritt

	Bonsai	Teddy	Sparschwein
Erzielte GFLOPS	470	808	636
Ausnutzung	0.19	0.32	0.25

Abbildung 6.18: **Ausnutzung der TMUs.**

sieben durchgeführt werden, und das Normieren, welches im letzten Schritt ausgeführt wird. Dies bringt insgesamt pro RGBA-Textursample ebenfalls nur 57 FLOPs ein. Daraus kann man nun aus der Zahl der A-Textursamples und der RGBA-Textur Samples aus dem Benchmark aus Punkt 6.2.7 die erzielte Rechenleistung bestimmen. Zusätzlich kann man dadurch, dass man die erzielte Rechenleistung durch die maximale Rechenleistung teilt die Ausnutzung bestimmen. Dies wurde in Abbildung 6.17 getan. Während man mit den CUDA-Cores in etwa 637 bis 722 GFLOPS erzielt, so erzielt man mit den TMUs in etwa 470 bis 808 GFLOPS. Dadurch wird deutlich, dass die Verwendung der TMUs sehr wichtig für die Performance des Programms ist.

6.3.4 Skalierbarkeit

In diesem Punkt soll die Skalierbarkeit des Verfahrens, in Abhängigkeit von der Multiprozessorzahl (gemessen in Punkt 6.2.3), dem Chiptakt (gemessen in Punkt 6.2.1) und dem Speichertakt (gemessen in Punkt 6.2.2) untersucht werden. Während man aus den Messwerten des Speichertakts direkt sagen kann, dass dieser im Messbereich einen vernachlässigbaren Einfluss auf die Performance hat, so sind die Messreihen für den Chiptakt und die Multiprozessorzahl weniger anschaulich. Deshalb scheint es sinnvoll, für beide Messreihen den Speedup und die Skalierungseffizienz zu berechnen und diesen in Diagrammen zu veranschaulichen.

Zuerst wird hierfür der Speedup für die verschiedenen Zahlen von Multiprozessoren berechnet. Diese Berechnungen sind in Abbildung 6.19 zu sehen. Da die Rechenleistung maximal mit der Zahl der Multiprozessoren skaliert, definiert man die Skalierungseffizienz als das Verhältnis zwischen Multiprozessorzahl und Speedup. Diese wurde in Abbildung 6.20 berechnet. Da der Speedup bei 16 Multiprozessoren für alle Modelle in etwa 15.6 oder die Skalierungseffizienz in etwa 98 Prozent beträgt, kann man sagen, dass das Verfahren sehr gut linear mit der Zahl der Multiprozessoren skaliert. So verliert man gegenüber optimaler Skalierung in etwa nur 2 Prozent Performance. Zur besseren Übersicht wurde diese Messung noch einmal in den Diagrammen in den Abbildungen 6.21 und 6.22 und visualisiert.

Als Nächstes wird ebenfalls der Speedup für den Chiptakt berechnet. Der Chiptakt wird hierfür auf den niedrigsten Chiptakt, für welchen die Messung ausgeführt worden ist, normiert. Das Ergebnis dieser Rechnung ist in Abbildung 6.23 zu sehen. Analog definiert man wieder eine Skalierungseffizienz als das Verhältnis von Speedup zu normierten Chiptakt und berechnet diese. Die Ergebnisse sind in Abbildung 6.24 zu sehen.

Bei dem höchsten gemessenen Chiptakt von 900 MHZ liesse sich maximal ein Speedup von 2.22 erwarten. Da man bei jedem Modell einen Speedup von in etwa 2.16 erzielt bzw. die Skalierungseffizienz in etwa 97 Prozent beträgt, folgt, dass das Verfah-

Multiprozessoren	Bonsai-Speedup	Teddy-Speedup	Sparschwein-Speedup
1	1.00	1.00	1.00
2	1.99	2.00	2.00
3	2.98	2.98	2.97
4	3.97	3.98	3.94
5	4.90	4.92	4.92
6	5.93	5.92	5.88
7	6.91	6.94	6.88
8	7.91	7.87	7.75
9	8.50	8.84	8.75
10	8.97	9.90	9.60
11	10.05	10.89	10.51
12	10.81	11.88	11.69
13	12.01	12.16	12.68
14	13.65	13.79	13.63
15	14.60	14.75	14.60
16	15.55	15.78	15.58

Abbildung 6.19: **Speedup in Abhängigkeit von der Zahl der Multiprozessoren.**

Multiprozessoren	Bonsai-Skalierungseffizienz	Teddy-Skalierungseffizienz	Sparschwein-Skalierungseffizienz
1	1.000	1.000	1.000
2	0.995	1.000	0.998
3	0.994	0.995	0.990
4	0.991	0.995	0.985
5	0.981	0.984	0.983
6	0.988	0.987	0.981
7	0.987	0.992	0.982
8	0.988	0.983	0.968
9	0.944	0.982	0.972
10	0.897	0.990	0.960
11	0.914	0.990	0.955
12	0.901	0.990	0.974
13	0.924	0.935	0.975
14	0.975	0.985	0.974
15	0.974	0.983	0.973
16	0.972	0.986	0.974

Abbildung 6.20: **Skalierungseffizienz in Abhängigkeit von der Zahl der Multiprozessoren.**

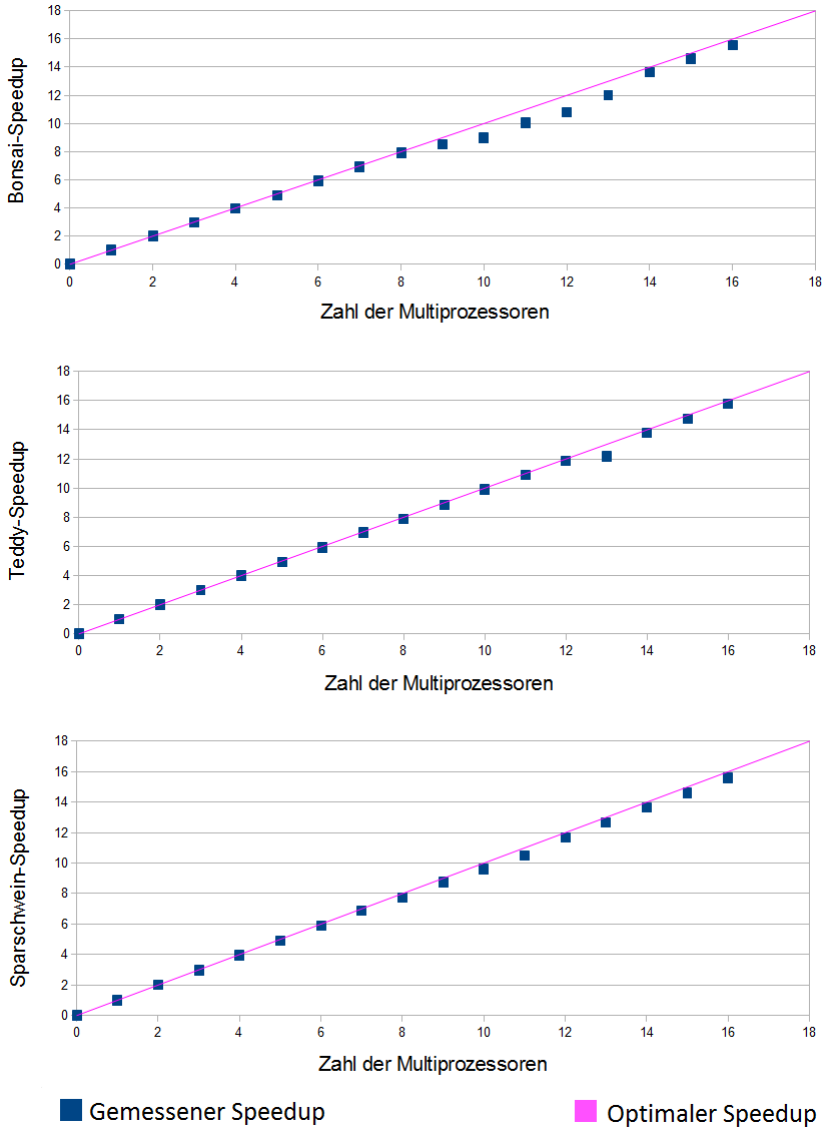


Abbildung 6.21: **Diagramm des Speedups in Abhängigkeit von der Zahl der Multiprozessoren.** Da diese Punkte für die verschiedenen Modelle fast identisch sind, wurde für jedes Modell ein eigenes Diagramm gezeichnet. Die Diagramme verdeutlichen noch einmal, dass das Verfahren nahezu linear mit der Zahl der Multiprozessoren skaliert.

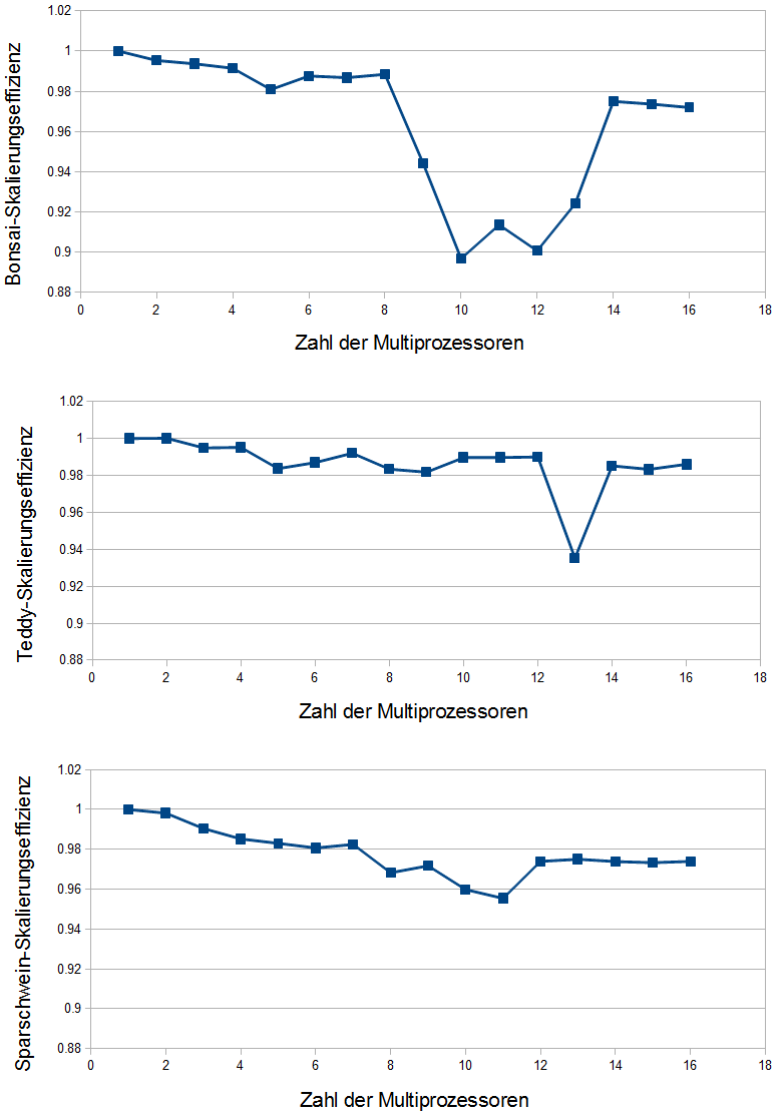


Abbildung 6.22: **Diagramm der Skalierungseffizienz in Abhängigkeit von der Zahl der Multiprozessoren.** Bei einigen Multiprozessorzahlen wird deutlich, dass die Skalierungseffizienz um bis zu 10 Prozent abfällt, wobei sie sich von dem Abfall später wieder erholt.

Chiptakt in MHZ	Normierter Chiptakt	Bonsai-Speedup	Teddy-Speedup	Sparschwein-Speedup
405	1.00	1.00	1.00	1.00
500	1.23	1.23	1.23	1.23
600	1.48	1.47	1.48	1.46
700	1.73	1.70	1.72	1.70
800	1.98	1.92	1.95	1.92
900	2.22	2.15	2.18	2.15

Abbildung 6.23: **Speedup in Abhängigkeit vom normierten Chiptakt.**

Chiptakt in MHZ	Bonsai-Skalierungs-effizienz	Teddy-Skalierungs-effizienz	Sparschwein-Skalierungs-effizienz
405	1.000	1.000	1.000
500	0.995	0.995	0.995
600	0.990	0.990	0.989
700	0.985	0.985	0.981
800	0.973	0.973	0.970
900	0.968	0.968	0.967

Abbildung 6.24: **Skalierungseffizienz in Abhängigkeit vom Chiptakt.**

ren innerhalb des Messbereichs fast linear mit dem Chiptakt skaliert. So verliert man gegenüber optimaler Skalierung nur in etwa 3 Prozent Performance. Zur Veranschaulichung werden sowohl der Speedup als auch die Skalierungseffizienz in Diagramme eingetragen. Dies ist in Abbildung 6.25 und 6.26 zu sehen.

Durch die Betrachtung der Messungen, der hier vorgenommenen Berechnungen und der Diagramme können zusammengefasst folgende Aussagen über die Skalierbarkeit innerhalb des Messbereichs für alle Modelle getroffen werden:

- Das Verfahren skaliert fast linear mit dem Chiptakt.
- Das Verfahren skaliert nicht mit dem Speichertakt.
- Das Verfahren skaliert fast linear mit der Zahl der Multiprozessoren. Hier traten jedoch bei manchen Multiprozessorzahlen Abweichungen auf.

Diese Skalierungen entsprechen auch soweit den bisherigen Erwartungen. Dennoch kann man daraus einige neue Schlüsse ziehen.

Die zuvor berechnete Ausnutzung der Speicherbandbreite ist sehr gering, da die Caches nahezu jeden Speicherzugriff abfangen. Somit entspricht es den Erwartungen, dass der Speichertakt und damit die Speicherbandbreite im betrachteten Messbereich keine Rolle spielen und dadurch nicht limitieren. Allerdings kann man auf diese

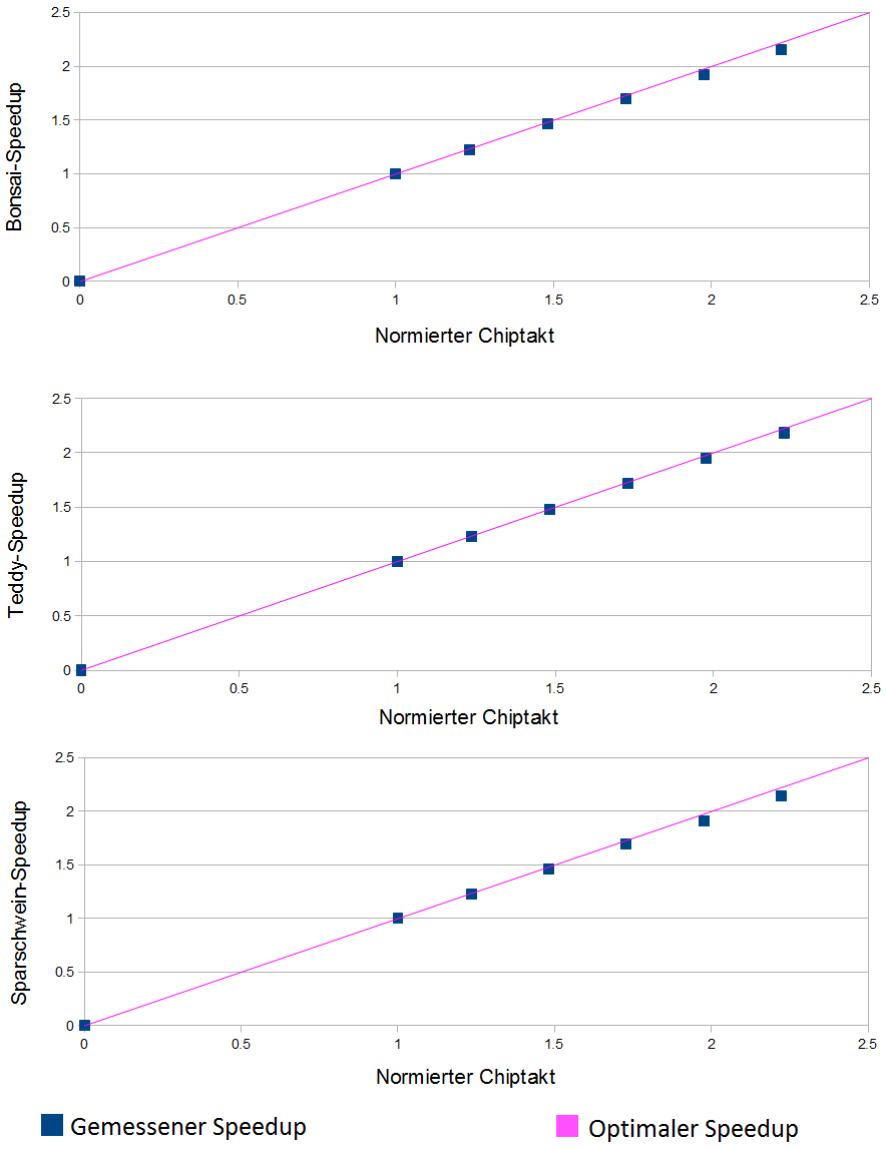


Abbildung 6.25: **Diagramm des Speedup in Abhängigkeit des Chiptakts.** Deutlich zu erkennen ist, dass das Verfahren in etwa linear mit dem Chiptakt bei allen Modellen skaliert.

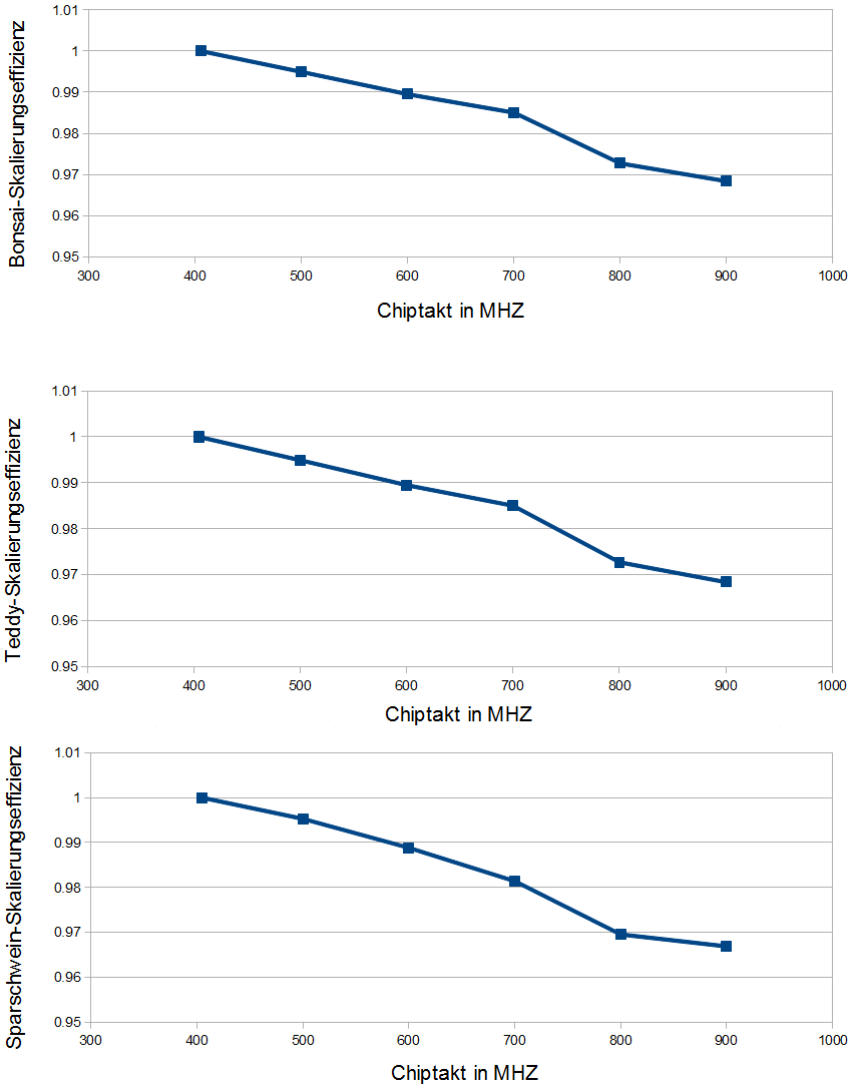


Abbildung 6.26: **Diagramm der Skalierungseffizienz in Abhängigkeit des Chiptakts.**

Art nur schlecht über die Limitierung durch die Speicherlatenzen eine Aussage treffen, da diese eventuell durch das BIOS der Graphikkarte nahezu konstant gehalten werden und sich durch den Speichertakt nur wenig verändern.

Des Weiteren folgt aus der linearen Skalierung mit dem Chiptakt ebenfalls, dass die Speicherbandbreite nicht limitiert. Zudem kann man aus dieser linearen Skalierung folgern, dass die Zugriffszeiten auf den RAM der Graphikkarte gut verborgen werden können und sich somit nicht auf die Performance auswirken. Dadurch können höchstens chipinterne Latenzen die Performance negativ beeinflussen.

Die lineare Skalierung mit der Multiprozessorzahl entspricht soweit ebenfalls den Erwartungen für ein Verfahren, welches peinlich parallel auf Pixelbasis bzw. auf Workgroupbasis ist. Der temporäre Abfall der Skalierungseffizienz ist hierbei ebenfalls leicht zu erklären. Denn am Schluss des Renderns, wenn keine neuen Workgroups mehr zur Verfügung stehen, müssen alle Multiprozessoren warten, bis auch der letzte Multiprozessor das Berechnen seiner aktuellen Workgroup abgeschlossen hat. Wäre man nicht gezwungen bei dieser Messung große Workgroups zu wählen, wäre die Arbeit feiner granuliert, wodurch sie sich besser auf die Multiprozessoren aufteilen liesse. Damit würden sich vermutlich diese Abweichungen noch einmal reduzieren. Zusätzlich kann man durch die Skalierung mit den Multiprozessoren aussagen, dass die Gesamtheit der Ressourcen, welche sich die Multiprozessoren untereinander teilen, durch die zusätzliche Auslastung nicht oder kaum die Performance reduziert. Hierunter fallen insbesonders die L2-Cache-Bandbreite, die L2-Cache-Trefferrate, die Speicherbandbreite und das chipinterne Interconnect Network.

6.3.5 Einfluss der Workgroupgrösse auf die Performance

Nun soll der Einfluss der Workgroupgrösse auf die Performance diskutiert werden. Diese Messung wurde in Punkt 6.2.4 vorgenommen. Zur besseren Übersicht wird aus ihr zuerst ein Diagramm erstellt. Für das Diagramm wurde die Laufzeit auf die minimale Laufzeit normiert. Das Diagramm ist in Abbildung 6.27 zu sehen.

In dem Diagramm erkennt man, dass bei einer grossen Workgroupgrösse die Performance im Allgemeinen geringer ist, als bei einer kleinen Workgroupgrösse, sofern bei beiden Workgroupgrössen der Multiprozessor die selbe Occupancy besitzt. Dies lässt sich damit erklären, dass bei der kleinen Workgroupgrösse die Arbeit feiner granuliert ist. Dadurch lässt sie sich besser auf die Multiprozessoren aufteilen. Zudem besteht bei grösseren Workgroups das Problem, dass die Warps eine unterschiedliche Laufzeit haben, weshalb die Workgroup so lange wie der am längsten laufende Warp den Multiprozessor belegt. Auf diese Art sind am Ende der Laufzeit einer Workgroup nur noch wenig Warps aktiv und Latenzen können schlechter überbrückt werden. Durch beides wird die Achieved-Occupancy reduziert. So ist es aus den vorgenommenen Messungen ersichtlich, dass die optimale Workgroupgrösse aus zwei quadratisch angeordneten Warpkacheln besteht. Dies lässt sich damit begründen, dass sie die kleinste Workgroupgrösse ist bei der der Multiprozessor die bei diesem Kernel höchste Occupancy hat. Sowohl die feine Granulation der Arbeit als auch die höchste Occupancy führen dazu, dass diese Workgroupgrösse die beste Achieved-Occupancy und damit Performance bietet.

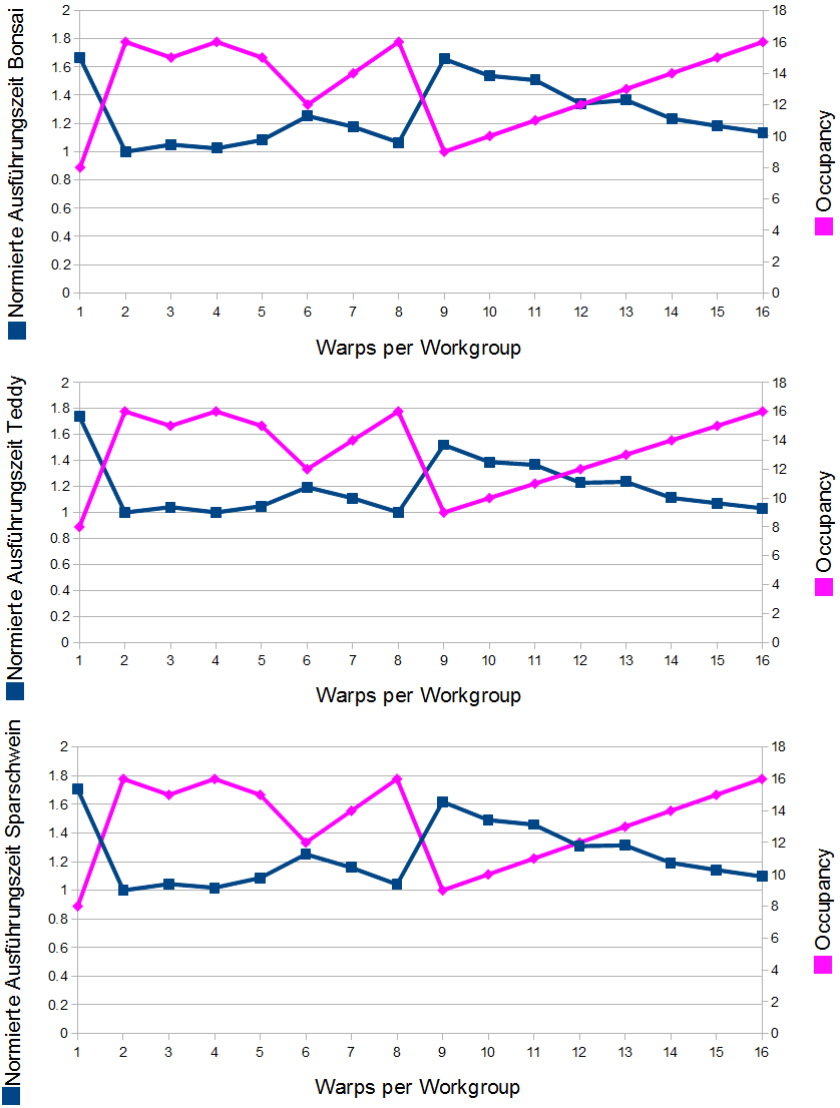


Abbildung 6.27: **Normierte Laufzeit in Abhängigkeit von der Workgroup-grösse.** In Magenta wurde zusätzlich die Occupancy eingezeichnet.

Occupancy	Bonsai-Speedup	Teddy-Speedup	Sparschwein-Speedup
1	1.00	1.00	1.00
2	1.93	1.97	1.94
3	2.78	2.87	2.79
4	3.58	3.73	3.60
5	4.28	4.53	4.32
6	5.06	5.36	5.12
7	5.54	5.95	5.63
8	6.36	6.81	6.45
9	6.88	7.35	6.99
10	7.46	8.05	7.60
11	7.58	8.17	7.76
12	8.51	9.11	8.65
13	8.39	9.04	8.61
14	9.25	10.02	9.50
15	9.70	10.42	9.94
16	10.6	10.82	10.36

Abbildung 6.28: **Speedup in Abhängigkeit von der Occupancy.**

6.3.6 Einfluss der Occupancy auf die Performance

Als Nächstes soll betrachtet werden, wie sehr sich die Occupancy, also die Zahl der beherbergten Warps eines Multiprozessors, auf die Performance auswirkt. Für diese Betrachtung wird die Messreihe aus Punkt 6.2.5 herangezogen. Diese Messreihe hat allerdings den Nachteil, dass die Workgroupgrösse sehr gross und damit wie in Punkt 6.3.5 beobachtet nicht optimal ist. Zusätzlich weicht die Workgroupgrösse von derjenigen Workgroupgrösse der übrigen Messreihen ab. Zur besseren Übersicht wird wieder der Speedup im Vergleich zu einem Warp berechnet. Dies ist in Abbildung 6.28 zu sehen und wurde wieder in ein Diagramm 6.29 eingetragen.

Aus dieser Tabelle und dem Diagramm erkennt man einen Trend, dass je höher die Occupancy ist, desto weniger Performancegewinn lässt sich durch einen weiteren zusätzlichen Warp erzielen. Dennoch erzielt man bis zur maximal möglichen Occupancy für jeden zusätzlichen Warp einen deutlichen Performancegewinn. So lässt sich aus der Skalierung mit der Occupancy aussagen, dass die maximale Occupancy von 16 Warps und die dadurch erzielte Achieved-Occupancy bei dieser grossen Workgroupgrösse sehr wahrscheinlich nicht ausreicht um die GPU auszulasten. Dies gilt ebenfalls aller Wahrscheinlichkeit nach auch für die optimale Workgroupgrösse von zwei Warpkacheln, obwohl die Achieved-Occupancy bei der kleineren Workgroupgrösse wahrscheinlich höher ist. Interessant sind im Diagramm die temporären Abfälle des Speedups bei den Primzahlen 7, 11 und 13. Dies ist darauf zurückzuführen, dass bei diesen Occupancies sich die Warpkacheln nur in einer Reihe anordnen lassen. Dadurch ist die Workgroupform weit von einem Quadrat entfernt, wodurch unter anderem die Achieved-Occupancy reduziert wird.

Es wäre auch interessant abschätzen zu können, wie viel Performance durch die zu niedrige Occupancy verloren geht. Hierfür müsste man die Steigung der Tabelle be-

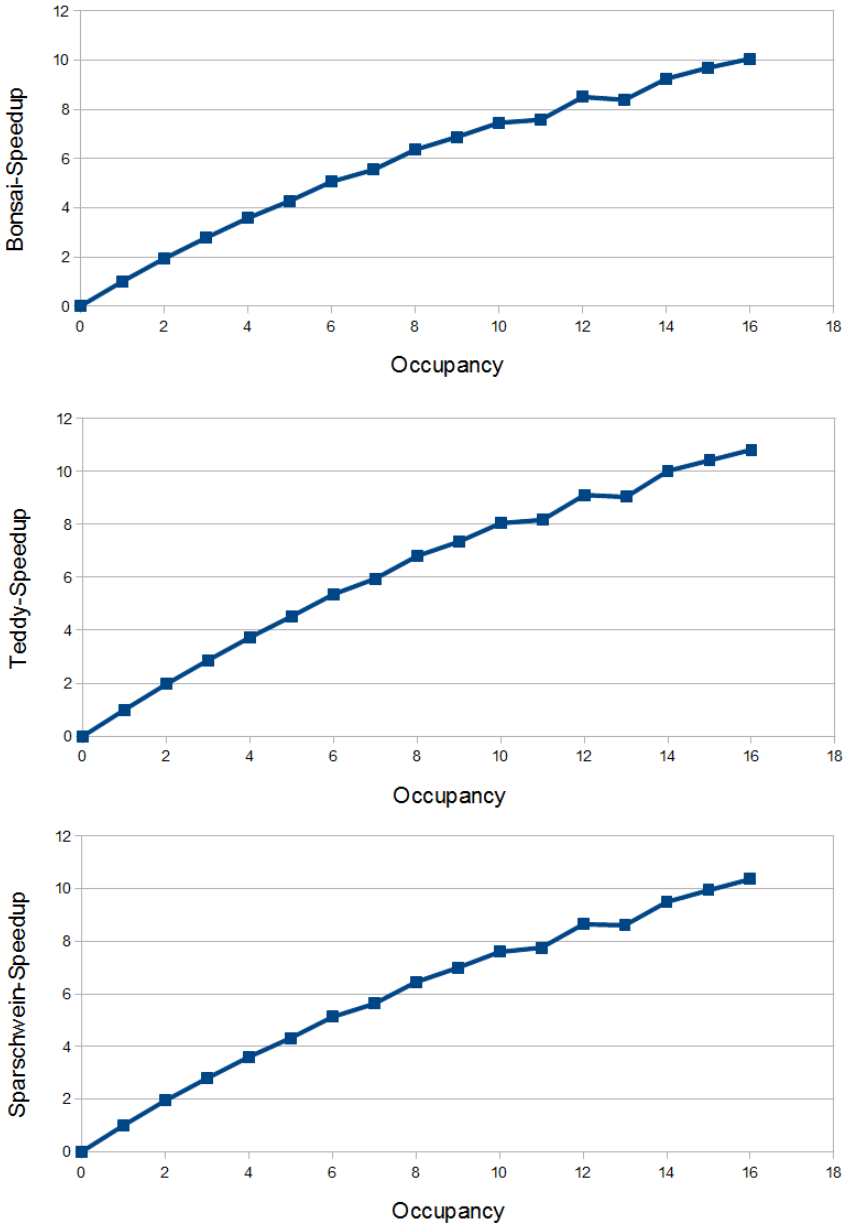


Abbildung 6.29: **Diagramm des Speedups in Abhängigkeit von der Occupancy.**

trachten und eventuell extrapolieren. Da man für die Extrapolation jedoch keinerlei Fehlermass angeben kann wird hier darauf verzichtet.

Da die Performance sehr wahrscheinlich durch eine zu niedrige Occupancy reduziert wird, könnte man versuchen, diese durch eine Erhöhung der Occupancy zu verbessern. Die einfachste Möglichkeit hierfür wäre, die Zahl der Register des Buffered-For-For-With-Flags-Kernels durch den entsprechenden Compilerbefehl zu begrenzen. Dies ist allerdings ein zweischneidiges Schwert, da dadurch zusätzliches Registerspilling entsteht, wodurch die Performance reduziert werden kann. Auch wurde das Kernel selbst nicht auf einen niedrigen Registerverbrauch hin optimiert. So könnte man versuchen den Registerverbrauch zu senken und dadurch die Occupancy zu erhöhen. So könnte man an diversen Programmstellen Ergebnisse nicht in den Registern zwischenspeichern, sondern diese jedes Mal aufs neue redundanterweise berechnen. Zudem könnte man versuchen, die Berechnungsreihenfolge leicht umzuändern, so dass man an manchen Programmstellen weniger Zwischenergebnisse in den Registern aufbewahren muss. Durch diesen niedrigeren Registerverbrauch liesse sich ebenfalls die Occupancy und damit die Performance erhöhen.

6.4 Limitierung der Performance

In den letzten Punkten zeigte sich, dass RAM-Latenzen und RAM-Bandbreite nur einen sehr geringen Einfluss auf die Performance haben. Auch lässt sich aussagen, dass diejenigen Ressourcen, welche sich von allen Multiprozessoren geteilt werden, wie die L2-Cache-Bandbreite oder das chipinterne Interconnect-Network, so wenig ausgelastet werden, dass sie dadurch die Performance nur unwesentlich beeinflussen.

Auch wurde berechnet, dass die mittlere Auslastung von allen Rechenwerken und den Warpschedulern nur mittelmässig oder gering ist. So betrug die mittlere Auslastung der Warpscheduler 53 bis 59 Prozent, der CUDA-Cores 54 bis 57 Prozent und der TMUs 34 bis 59 Prozent. Dennoch lässt sich die Limitierung, da der Algorithmus die TMUs und die Warpscheduler sehr unterschiedlich auslastet, wie in 6.3.3.7 erläutert, nicht ausschliessen. Des Weiteren zeigte sich, dass es wahrscheinlich ist, dass eine höhere Occupancy die Auslastung der GPU erhöhen würde. Wie sehr die Performance jedoch durch die zu geringe Occupancy limitiert wird, lässt sich mit diesem Buffered-For-For-With-Flags-Kernel nicht abschätzen.

Um zu bestimmen, inwieweit die TMUs und die Warpscheduler durch ihre ungleichmässige Auslastung die Performance zeitweise limitieren, wurde der Standard-Volumenraycaster ohne Octree verwendet um die Samplerate und die IPC beim Marschieren durch eine komplett leere und eine komplett volle Boundingbox zu bestimmen. Des Weiteren wurden der IPC-Wert und die Instruction-Replays bei einem Octreevisualisierungskernel anhand des Octrees des Bonsais gemessen. Beide Kernels haben den Vorteil, dass sie weniger Register als das Buffered-For-For-With-Flags-Kernel verbrauchen, wodurch sich auch höhere Occupancies simulieren lassen. Aus den Messergebnissen wurde dann jeweils die Auslastung der Warpscheduler und der TMUs berechnet sowie der Speedup, welcher durch die Erhöhung der Occupancy entstanden ist. Das Ergebnis dieser Messungen und Berechnungen ist in den Tabellen der Abbildung 6.30 und Abbildung 6.31 zu sehen. Dennoch müsste man berücksichtigen, dass beiden Kernels der Teilalgorithmen die zwischen den Multiprozessoren geteilten

Gezeichnetes Objekt Occupancy	Leere Box 16	Leere Box 21	Volle Box 16	Volle Box 21
Samples in Gigasamples	1.56	1.56	4.70	4.70
IPC	0.55	0.65	0.97	1.15
t_{Ausf} in s	0.0871	0.0732	0.271	0.223
Instruction-Replay-Overhead	0	0	0	0
Schedulerauslastung	0.27	0.32	0.48	0.58
TMU-Auslastung	0.73	0.88	0.71	0.86
Speedup durch Occupancy	1.00	1.19	1.00	1.21

Abbildung 6.30: **Raymarchingmessungen durch eine volle und eine leere Boundingbox.**

Gezeichnetes Objekt Occupancy	Octree 16	Octree 21
IPC	1.53	1.62
t_{Ausf} in s	0.00901	0.00863
Instruction-Replay-Overhead	0.06	0.06
Schedulerauslastung	0.81	0.86
Speedup durch Occupancy	1.00	1.06

Abbildung 6.31: **Octreetraversierungsmessung.**

Ressourcen sehr unterschiedlich auslasten können als das Kernel des Gesamtalgorithmus. So könnte zum Beispiel bei einem Teilalgorithmus die Speicherbandbreite limitieren, während beim Gesamtalgorithmus die Speicherbandbreite nur wenig ausgenutzt werden würde. Diese Effekte sollen in der folgenden Diskussion vernachlässigt werden.

So lassen sich aus diesen Tabellen diverse Schlüsse ziehen:

- Beim Raymarching erzielt man immer einen grossen Performancegewinn durch zusätzliche Occupancy. So beträgt der Speedup sowohl bei der vollen und bei der leeren Boundingbox jeweils ca. 1.2. Daraus folgt, dass die beim Buffered-For-For-With-Flags erzielte Occupancy für das Raymarching noch nicht ausreicht um die GPU auszulasten.
- Die TMUs sind beim Raymarching bei der niedrigeren Occupancy von 16 Warps mit 71 bis 73 Prozent bereits mittelmässig bis gut ausgelastet. Bei der höheren Occupancy von 22 Warps erhöht sich ihre Auslastung auf 86 bis 88 Prozent. Deshalb ist in beiden Fällen schon eine Reduzierung der Performance durch die Auslastung der TMUs, wie es in Punkt 6.3.3.1 beschrieben worden ist, möglich; bei der höheren Occupancy von 22 Warps sogar wahrscheinlich.
- Die Auslastung der Warpscheduler beim Marschieren durch den leeren Raum ist selbst bei der höheren Occupancy von 22 Warps mit 30 Prozent sehr gering. Dadurch ist es unwahrscheinlich, dass die Warp-Scheduler so sehr ausgelastet

sind, dass die Performance durch deren nicht ausreichende Befehlsrate merklich reduziert wird. Marschiert man jedoch durch vollen Raum, so erhöht sich ihre Auslastung bei der höheren Occupancy auf 60 Prozent. Hier könnte die Auslastung bereits die Performance reduzieren.

- Bei der Octreetraversierung fällt der Speedup durch zusätzliche Occupancy mit 1.06 gering aus. Dies ist darauf zurückzuführen, dass die Warpscheduler bei der niedrigeren Occupancy bereits mit 81 Prozent gut ausgelastet sind. So wurde ihre Auslastung durch die Erhöhung der Occupancy auf 86 Prozent erhöht. In beiden Fällen ist die Auslastung des Warpscheduler so hoch, dass wie in Punkt 6.3.3.1 beschrieben es sehr wahrscheinlich ist, dass die Performance durch deren zu niedrige Befehlsrate reduziert wird.

So zeigt es sich bei jedem Teilalgorithmus, dass die beim Gesamtalgorithmus erzielte Occupancy von 16 nicht ausreicht um die GPU im Teilalgorithmus auszulasten. Somit gilt dies auch für den Gesamtalgorithmus. Da die Warpscheduler und die TMUs in den Teilalgorithmen so sehr ausgelastet werden, dass eine eintretende Limitierung möglich bis wahrscheinlich ist, gilt dies ebenfalls für den Gesamtalgorithmus.

6.5 Fazit der Untersuchungen

Nun sollen kurz die Ergebnisse der Untersuchungen dieses Kapitels zusammengefasst werden. So zeigte es sich, dass die Cache-Effizienz bei diesem Algorithmus durch die Lokalität der Speicherzugriffe gut ist. Auch wird nur ein Bruchteil der maximalen Speicherbandbreite verbraucht, wodurch diese nicht limitiert. Deshalb skaliert das Verfahren nicht mit der Speicherbandbreite. Eine Limitierung durch eine zu niedrige Befehlsrate der Warpscheduler ist wahrscheinlich und eine Limitierung durch eine zu niedrige Samplerate der TMUs ist möglich. Ebenfalls wird die Performance durch die zu niedrige Occupancy reduziert. Die RAM-Zugriffszeiten können jedoch bereits so gut verborgen werden, dass sie keinen Einfluss auf die Performance nehmen. Dadurch wird die höhere Occupancy hier nicht benötigt um die RAM-Zugriffszeiten zu überbrücken. Auch zeigte sich, dass die L2-Cache-Bandbreite und die Bandbreite des Interconnect-Networks nicht die Performance limitieren. Auch konnte man folgern, dass die Rechenleistung der TMUs und der CUDA-Cores nur mittelmässig bis schlecht ausgenutzt wird. Des Weiteren skaliert das Verfahren in etwa linear mit dem Chiptakt der Graphikkarte und mit der Multiprozessorzahl.

Auf Grund dieser Beobachtungen kann man nun folgern, wie man die Hardware für eine höhere Performance verbessern könnte:

- Da eine zeitweise Limitierung durch die Warpscheduler wahrscheinlich ist, könnte man versuchen die Performance durch eine Verbesserung der Befehlsrate der Warpscheduler zu erhöhen. Eine erhöhte Befehlsrate würde jedoch dazu führen, dass die CUDA-Cores ebenfalls die Performance limitieren können. Unter der Annahme, dass die meisten Befehle CUDA-Core-Befehle sind, ist dies ebenfalls wahrscheinlich. Somit müsste man in diesem Fall für einen deutlichen Performancegewinn ebenfalls den IPC-Wert der CUDA-Cores erhöhen.

	Bonsai	Teddy	Sparschwein
Rechenleistung TMUs in GFLOPS	470	808	636
Rechenleistung CUDA-Cores in GFLOPS	723	637	686
Rechenleistung Insgesamt in GFLOPS	1193	1445	1492
Ausnutzung der Rechenleistung der GPU	0.29	0.35	0.36
Vielfaches der Rechenleistung der CPU	7.2	8.7	9.0

Abbildung 6.32: **Ausnutzung der Rechenleistung insgesamt und Vergleich mit der maximalen Rechenleistung einer CPU.**

- Da auch eine zeitweise Limitierung durch die TMUs möglich ist, könnte man versuchen die Performance des Programms zu erhöhen, indem man die Leistung der TMUs verbessert.
- Da die Occupancy nicht ausreicht um die GPU gut auszulasten, würde ein vergrösserter Registersatz die Performance verbessern.
- Auch könnte man die Performance weiter verbessern, indem man die GPU durch mehr Multiprozessoren stärker parallelisiert, oder die Komplexität eines Multiprozessors weiter erhöht, so dass dieser mit einem höheren Takt betrieben werden kann.

Zudem wurden in diesem Kapitel eine Vielzahl von weiteren Optimierungen für das Buffered-For-For-With-Flags-Kernel vorgeschlagen. Leider würde es den Rahmen dieser Arbeit überschreiten, diese Verbesserungen zu implementieren und noch einmal die Performance durch Messungen zu überprüfen.

Abschliessend soll kurz abgeschätzt werden, wie viel Rechenleistung man insgesamt durch die TMUs und CUDA-Cores erzielt. Diese Rechenleistung wird zusätzlich in Relation zu der maximal möglichen Rechenleistung beider Rechenwerke von 4134 GFLOPS gesetzt. Auch wird sie in Relation zur Peak-Performance von 166 GFLOPS einer modernen Intel Core i7 990X-CPU gesetzt. Dies wurde in Abbildung 6.32 berechnet. So zeigt sich, dass obwohl man bei diesem Algorithmus die Rechenleistung der GPU insgesamt nur mit 29 bis 36 Prozent mittelmässig bis schlecht ausnutzt, man dennoch das 7 bis 9 fache der maximalen CPU-Performance erzielt.

7 Schluss

In diesem Kapitel soll kurz der Erfolg dieser Arbeit bewertet werden. So eignet sich das im Kapitel 3 implementierte Verfahren bereits gut um Volumendaten anzuzeigen. Auch läuft das Verfahren bereits auf moderner Hardware in Bildwiederholraten, welche Interaktivität ermöglichen. Die Implementierung des Octrees in den Raycaster im Kapitel 4 ermöglichte bei Volumendaten mit viel leeren Raum bereits bei der First-Approach-Kombination einen Performancegewinn. Dieser Performancegewinn wurde in späteren Kombinationen durch eine erhöhte SIMD-Effizienz noch weiter verbessert. Das resultierte in der performantesten Kombination, welche als Buffered-For-For-With-Flags bezeichnet wurde. Der anschliessende Versuch im Kapitel 5 das Verfahren mit persistenten Threads zu verbessern wurde den Erwartungen nicht gerecht. So reduzierten sich bei diesem Verfahren sowohl die SIMD-Effizienz als auch die Performance. Schliesslich zeigte sich bei den Untersuchungen in Kapitel 6, dass das Verfahren nahezu linear mit der Multiprozessorzahl und dem Chiptakt skaliert, nur wenig Speicherbandbreite benötigt, und die Speicherlatenzen gut verborgen werden können. Zusätzlich zeigte sich, dass die Occupancy nicht ausreicht um die GPU auszulasten. Auch konnte man folgern, dass man die Rechenleistung der verschiedenen verwendeten Rechenwerke nur mittelmässig bis schlecht ausnutzt.

8 Anhang

8.1 Probleme mit OpenCL

Bei der Implementierung des praktischen Teils der Arbeit zeigte sich zusätzlich, dass Nvidias OpenCL-Implementierung in einem schlechten Zustand ist. Dadurch eignet sie sich immer noch nicht für ernsthafte Projekte. Beispielhaft seien folgende Bugs genannt:

- Ein "if(true)....else...." im Kernel-Quellcode führt zum Absturz des Compilers. Dies ist besonders frustrierend, da der Compiler automatisch konstante Terme im Quellcode zu true und false vereinfacht. Dadurch gestaltet sich das Suchen nach diesem Bug schwierig. Zusätzlich erschwerte dies zum Teil das Debuggen, da man das Prädikat des ifs nicht einfach auf true setzen konnte.
- Bei zu komplexen Schleifenstrukturen kommt es bei Sprungbefehlen zu Compilerbugs, so dass das Kernel nicht richtig funktioniert.
- Ein "write_imagef(FrameBuffer, PositionOnScreen,0.0f);" führte an einer Stelle dazu, dass in den Framebuffer, nicht wie das "0.0f" eigentlich bewirken sollte ein schwarzer Pixel, sondern ein weisser Pixel geschrieben wird.
- In älteren Treiberversionen bestand zu Projektbeginn das Problem, dass das Context-Sharing mit OpenGL nicht richtig funktionierte und sich Buffer in einem Kernel nicht auslesen liessen. Dieses Problem verschwand allerdings mit neueren Treiberversionen.
- Compilerfehlermeldungen, welche ebenfalls in einem Absturz des Compilers resultierten. Zum Beispiel:

  " Stored value type does not match pointer operand type! store i32* %11, i32 addrspace(3)** %Value, align 4 i32 addrspace(3)*Broken module found, compilation aborted! "
- Besitzt eine Funktion die falsche Anzahl an Parametern kommen Fehlermeldungen wie:

  "ptxas application ptx input, line 17674; error : Call has wrong number of parameters ptxas fatal : Ptx assembly aborted due to errors"

  Die in der Fehlermeldung genannte Zeile bezieht sich allerdings auf die Zeile in dem zuerst vom Compiler erzeugten PTXAS-Assembly, wodurch das Finden dieser Funktion im OpenCL Quellcode unnötig erschwert wird.
- Auf einer Geforce 670 GTX stürzt der Compiler beim Kompilieren des Volumenraycasterkernels nach dem Erzeugen des PTXAS-Assemblys beim Erzeugen des Maschinencodes ab.
- Da Nvidia seine eigene Interpretation des OpenCL-Standards verwendet läuft der Code in seiner finalen Version nur auf Nvidiakarten und erzeugt Fehler beim Versuch ihn auf anderen Plattformen zu kompilieren. In wie weit die Schuld hierbei nur bei Nvidia oder auch bei den anderen Herstellern liegt ist allerdings etwas unklar.

- Nvidia legt einen Computing-Cache auf der Festplatte für bereits kompilierte OpenCL Kernels an. Dadurch wird ein Kernel nicht erneut kompiliert, wenn es sich bereits in diesem Cache befindet. Allerdings bietet Nvidia die Möglichkeit, dass man sich diverse Kernelstatistiken bei der Kompilierung eines Kernels ausgeben lassen kann. Findet die Kompilierung jedoch wegen dem Cache nicht statt, so werden diese Statistiken obwohl man ihre Ausgabe befiehlt, ebenfalls nicht ausgegeben. In diesem Fall ist es nötig den Cache zu löschen oder das Kernel leicht zu modifizieren.

Da jedoch im Verlaufe der halbjährigen Arbeitszeit mehrere Treiberversionen verwendet worden sind, kann es sein, dass die Bugs zum Teil bereits behoben sind.

8.2 Praktische Arbeit

Die auf einer DVD beiliegende praktische Arbeit wurde mit QT 4.7.4 und OpenCL 1.1 programmiert. Zusätzlich wird Windows und Glew benötigt. Ausserdem funktioniert das Programm NUR mit Nvidias OpenCL-Implementierung. Denn sowohl bei Intels als auch bei AMDs OpenCL-Implementierung kommt es zu Fehlern bei der Kompilierung der Kernels. Wegen des zeitlich beschränkten Umfangs der Arbeit wurde darauf verzichtet den Code noch auf den anderen beiden anderen OpenCL-Implementierungen lauffähig zu machen. Des Weiteren wurde das Programm in seiner finalen Version nur auf einer Geforce 580 GTX getestet. So scheiterte der Versuch es auf einer Geforce 670 GTX zum Laufen zu bekommen, da der Nvidia Compiler nach dem Erstellen des PTXAS-Assembly und beim Erstellen des Maschinencodes einen Fehler erzeugte und abstürzte.

8.3 Quellenangaben

[AK05] *"Overview of Volume Rendering"*
Arie Kaufman und Klaus Mueller
Center for Visual Computing — Computer Science Department — Stony Brook University,2005

[SO98] *"Ray - Box Intersection"*
Scott Owen
Web-Release, `http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm`, April 1998

[KH11] *"The OpenCL Specification - Version: 1.1 "*
Khronos Group
Web-Release,
`http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, Januar 2011

[WPa] *"Volume rendering"*
Wikipedia - The Free Encyclopedia
Web-Release, `http://en.wikipedia.org/wiki/Volume_rendering`, Dezember 2012

[WPb] *"Phong-Beleuchtungsmodell"*
Wikipedia - The Free Encyclopedia
Web-Release, `http://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell`, Dezember 2012

[NVa] *"OpenCL Programming Guide"*
Nvidia Corporation
Teil des Nvidia Computing SDKs 4.2,
`https://developer.nvidia.com/CUDA-toolkit-42-archive`, Dezember 2012

[NVb] *"OpenCL Best Practices Guide"*
Nvidia Corporation
Teil des Nvidia Computing SDKs 4.2,
`https://developer.nvidia.com/CUDA-toolkit-42-archive`, Dezember 2012

[NVc] *"NVIDIA's Next Generation CUDA Compute Architecture: Fermi"*
Nvidia Corporation
Web-Release, `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`

[NVd] *"Understanding the Efficiency of Ray Traversal on GPUs"*
Timo Aila and Samuli Laine
Nvidia-Research, `http://www.nvidia.com/docs/IO/76976/HPG2009-Trace-Efficiency.pdf`, August 2009

[VOL] *"The Volume Library"*
Verschiedene
Web-Release,
`http://www9.informatik.uni-erlangen.de/External/vollib/`, Dezember 2012

[JM10] *"Exploiting Spatial and Temporal Coherence in GPU-Based Volume Rendering"*
Jörg Mensmann
Dissertation — Westfälische Wilhelms-Universität Münster,2010

[CM10] *"Interaktive Visualisierung"*
Christian Meß
Seminar,WS 06/07)

[RTVG06] *"Real-Time Volume Graphics"*
Klaus Engel, Markus Hadwiger,Joe M. Kniss, Christof Rezk-Salama and Daniel Weiskopf
A K Peters Ltd.,2006

[GS12] *"What is the difference: DRAM Throughput vs Global Memory Throughput"*
Greg Smith
Forumsdiskussion, `http://stackoverflow.com/questions/10901441/what-is-the-difference-dram-throughput-vs-global-memory-throughput`, Juli 2012

[ICP] *"KD-trees and icp implementations using graphics hardware"*
Maximilian Reischl
Bachelor-Thesis, Universität Bayreuth, Juni 2012

8.4 Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und ohne fremde Hilfe verfasst, keine neben den von mir angegebenen Hilfsmitteln und Quellen dazu verwendet und die den benutzten Werken inhaltlich oder wörtlich entnommenen Stellen als solche kenntlich gemacht habe. Des Weiteren versichere ich, dass ich diese Arbeit nicht bereits zur Erlangung eines akademischen Grades eingereicht habe.

Bayreuth, Januar 2013

_____

Tim Werner