UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Ray-Tracing mit CUDA

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von
Hanno Rabe

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	Dipl.-Inform. Oliver Abert (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im September 2008

Erklärung

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

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

.....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Schon die starre Fixed-Function-Pipeline früher Graphikchips bot gewisse Funktionalitäten, die vereinzelte Entwickler gewinnbringend in Szenarios einzusetzen wußten, für welche die Hardware ursprünglich nicht vorgesehen war; Veröffentlichungen hierzu stammen aus den Bereichen der Robotik [LRDG90, HICK+99], der künstlichen Intelligenz [Boh98] oder auch der Bildverarbeitung und Visualisierung [HE99a, HE99b]. Als im Jahr 2001 erstmals ansatzweise frei programmierbare Graphikhardware im Endkundenmarkt eingeführt wurde, hatte die Suche nach Wegen, sich auch die Rechenleistung der neuen Prozessoren für Zwecke zunutze zu machen, die nicht oder nur indirekt Graphikdarstellung in Echtzeit als Ziel verfolgen, bereits begonnen [TAS00].

Aus solchen zunächst hauptsächlich akademischen Ansätzen heraus entwickelte sich ein zunehmend ernstgenommenes Bestreben, die immer leistungsfähigere Technik auch in praxisrelevanten Szenarios für allgemeine Berechnungen einzusetzen. Die von Chipgeneration zu Chipgeneration erweiterte Flexibilität, von der hierbei profitiert werden konnte, wurde von den Herstellern jedoch lange Zeit ausschließlich im Hinblick auf bessere Möglichkeiten im Einsatz als Graphikbeschleuniger verwirklicht. Auch bei der Auswahl der Werkzeuge für die Einbindung der Prozessoren blieb man vorerst abhängig von Entwicklungen, die in aller Regel für den klassischen Verwendungszweck der GPU vorangetrieben wurden – selbst ausdrücklich für die allgemeine Programmierung ausgelegte Entwicklungshilfen[1] basierten zu dieser Zeit unter ihrer Oberfläche auf den klassischen zweckgebundenen Shading-Sprachen[2]. Damit konnten zwar gewisse hilfreiche Abstraktionen zum Beispiel für die Verarbeitung von Datenströmen nach dem Stream-Paradigma angeboten werden, realisiert werden mußten solche Lösungen jedoch immer noch innerhalb der architektonischen Rahmenbedingungen eines Graphikchips. Dessen zweckmä-

[1] zum Beispiel BrookGPU, Sh, RapidMind
[2] Assembler und die Hochsprachen CG, GLSL, HLSL

ßige Auslegung bedingte manche Eigenschaften, die für die neuen Anwendungsgebiete fundamentale Unzulänglichkeiten bedeuteten.[1] Im Zweifelsfall konnte dieses Erbe zum Verzicht auf die in den Graphikprozessoren vorhandenen Rechenkapazitäten veranlassen, was die Akzeptanz der GPU für allgemeine Berechnungszwecke insgesamt schmälerte.

Es sind also, während Preis und Marktdurchdringung seit langem keine Hürden mehr für Graphikprozessoren als solche darstellen und somit eine entsprechende Basis durchaus gegeben ist, zwei wesentliche Faktoren auszumachen, welche die endgültige Etablierung der GPU als allgemeiner Coprozessor bisher verhinderten: die weitestgehend einseitige Orientierung der Architektur auf echtzeitfähige Graphikdarstellung einerseits und der Mangel an hinreichend robusten und flexiblen Entwicklungswerkzeugen andererseits. Um die GPGPU[2]-Bewegung aus ihrer inhärenten bremsenden Abhängigkeit zu befreien, bedurfte es demnach der Initiative einer Instanz, die in beiden erwähnten Bereichen Lösungen zu präsentieren oder zumindest entsprechenden Einfluß geltend zu machen vermochte. Ohne die aktive Beteiligung eines großen Prozessorherstellers blieb es also unmöglich, diesen entscheidenden Fortschritt zu machen.

In jüngerer Vergangenheit mehrten sich die Anzeichen für entsprechende Bestrebungen seitens aller hierfür in Frage kommenden Unternehmen:

Der ehemalige Graphikkartenentwickler ATI gab im Jahr 2006 im Rahmen einer GPGPU-Initiative die Zusammenarbeit mit dem Forschungsprojekt Folding@Home der Universität Stanford bekannt und veröffentlichte einen durch die Rechenkapazitäten ihrer Graphikchips beschleunigten Client. Kurz darauf wurde das Unternehmen von dem Halbleiterspezialisten AMD übernommen, der das GPGPU-Konzept aufgriff und heute unter dem Begriff »AMD Stream Computing« fortführt. In der Produktlinie FireStream bietet das Unternehmen explizit für die Stream-Verarbeitung optimierte Varianten seiner Graphikkarten an, und auf verschiedenen Abstraktionsebenen werden entsprechende Entwicklungswerkzeuge zur Verfügung gestellt.[3]

Konkurrent NVIDIA hatte bereits im Jahr 2004 für seine Serie professioneller Graphikbeschleuniger die Offline-Rendering-Software Gelato veröffentlicht und damit die Verwendbarkeit seiner Prozessoren abseits der traditionellen Rasterungstechnik demonstriert.[4] Einen noch größeren Schritt weg von klassischer Graphikbeschleunigung bedeutete die 2006 bekanntgegebene Zusammenarbeit mit Havok, einem Spezialisten für die Simulation physikalischer Effekte. Die Software Havok FX wurde entwickelt, um die in Computerspielen zunehmend anspruchsvollen Physikberech-

[1] Instruktionslimits, keine dynamischen Verzweigungen etc.

[2] »General-Purpose computation on Graphics Processing Units«: Allzweckberechnung auf Graphikprozessoren

[3] `http://ati.amd.com/technology/streamcomputing`

[4] `http://www.nvidia.com/object/IO_12820.html`

nungen vollständig auf der GPU auszuführen.[1] Später in diesem Jahr stellte NVIDIA mit der Prozessorgeneration G80 nicht nur seinen erwarteten neuen Graphikchip vor, sondern das Unternehmen untermauerte mit demselben Produkt auch seine Ambitionen, die starre Zweckgebundenheit der GPU auf breiter Ebene aufzulösen. Wesentlicher Bestandteil dieser Strategie ist neben der Hardware die gleichzeitig präsentierte CUDA-Entwicklungsumgebung, die eine freie Programmierung der GPU als Coprozessor ermöglicht. Die anschließende Einführung der spezialisierten Tesla-Produktreihe dokumentiert die Bereitschaft und das Bestreben NVIDIAs, bei Verwendung nahezu derselben Technik den Schwerpunkt der Graphikbeschleunigung gänzlich fallenzulassen zugunsten einer Akzeptanz als ernstzunehmende Alternative zu Server-Clustern.[2]

Der Halbleiterspezialist Intel spielt in wirtschaftlicher Hinsicht auf dem Markt der Graphikchips eine sehr bedeutende Rolle, was der großen Verbreitung seiner in Chipsätzen integrierten Graphiklösungen geschuldet ist. Diese sind jedoch traditionell im unteren Leistungssegment angesiedelt und konkurrieren daher auch nur auf dieser Ebene mit den Produkten der erwähnten Entwickler dedizierter Graphikchips. Mit seinen bisherigen Lösungen also kann und will Intel nicht denselben Weg wie die anderen Chiphersteller beschreiten, aber die Verschiebung, die der wichtige HPC[3]-Markt durch die beschriebenen Entwicklungen erfährt, veranlaßt auch dieses Unternehmen zu einer entsprechenden Ausrichtung seiner Forschungen. Zum Ausdruck kommt dies in dem Projekt Tera-Scale, das in seinem noch frühen Stadium als Vorausschau darauf interpretiert werden darf, in welche Richtung sich die Prozessortechnik generell entwickeln wird.[4] Bereits von den hierbei gewonnenen Erkenntnissen beeinflußt zeigt sich Larrabee, eine Chiparchitektur, mit der Intel das Konzept der hochparallelen Berechnung umsetzt. Sie wird zunächst in leistungsfähigen Graphikprozessoren Anwendung finden, hier jedoch eine flexible Programmierbarkeit in den Mittelpunkt stellen, so daß neben der konventionellen Rasterungstechnik ausdrücklich auch andere Verfahren, wie zum Beispiel Ray-Tracing, bei der Bilderzeugung zum Einsatz kommen oder auch allgemeine Berechnungen ohne den Zweck der Graphikbeschleunigung ausgeführt werden können.[5] Der Markteintritt dieser Technik steht noch bevor und wird voraussichtlich im Jahr 2009 oder 2010 erfolgen.[6] Intel wird also mittelfristig ebenfalls im GPU-, GPGPU- und HPC-Segment vertreten sein.

[1] http://www.nvidia.com/object/IO_30478.html

[2] http://www.nvidia.com/object/IO_43499.html

[3] »High-Performance Computing«: in diesem Zusammenhang hochgradig parallele Berechnungen zum Beispiel im Bereich des wissenschaftlichen Rechnens

[4] http://www.intel.com/research/platform/terascale

[5] http://softwarecommunity.intel.com/UserFiles/en-us/File/larrabee_manycore.pdf

[6] http://www.intel.com/pressroom/archive/releases/20080804fact.htm

Der Vollständigkeit halber sei in dieser Übersicht auch die ebenfalls seit dem Jahr 2006 in Form des Cell-Prozessors verfügbare »Cell-Broadband-Engine«-Architektur der STI-Allianz, bestehend aus Sony, Toshiba und IBM, aufgeführt. Hierbei handelt es sich um eine gleichsam hybride Prozessorarchitektur, welche die universelle Verwendbarkeit einer CPU und die spezielleren parallelen Verarbeitungsmechanismen einer GPU in sich vereint. Da in diesem Fall jedoch nicht mehr von GPGPU gesprochen werden kann – so besitzt die Multimediakonsole PLAYSTATION 3 neben einem Cell-Chip als Hauptprozessor zusätzlich einen dedizierten Graphikchip von NVIDIA –, wird an dieser Stelle nicht näher auf dieses Konzept eingegangen.

Es existieren also auf seiten aller namhafter Hersteller entweder bereits Lösungen oder zumindest entsprechende Ansätze, um die einstmals so exotische GPGPU-Entwicklung als vollwertige und gleichberechtigte Verwendungsmöglichkeit der Prozessoren zu etablieren bzw. den hierdurch neu definierten HPC-Sektor durch alternative Methoden und Produkte zu erschließen. Daß Rechenleistung, die vorher nur von Server-Clustern bereitgestellt werden konnte, nun unter anderem in Form von Zusatzkarten für den herkömmlichen PC verfügbar ist, verlagert schon die Forschungs- und Entwicklungstätigkeit vieler Unternehmen und Institutionen verschiedener Branchen.[1] Dennoch bleiben jene Anwendungsbereiche, in denen seit je her GPGPU-Techniken eingesetzt werden und die dadurch diese Entwicklung auch mitgetragen haben, wohl am besten dazu geeignet, den tatsächlichen Fortschritt zu bewerten und zu dokumentieren. Eine dieser schon traditionellen GPGPU-Anwendungen ist Ray-Tracing. Von den frühen Experimenten Purcells et al. [PBMH02] und Carrs et al. [CHH02] ausgehend, die übergreifend für die gesamte GPGPU-Bewegung von großer Bedeutung waren, begleiteten stetig neue Ideen, wie die zahlreichen Elemente des großen Themenkomplexes Ray-Tracing auf die GPU portiert werden könnten, und entsprechende Implementationen bis heute den Werdegang der allgemeinen GPU-Programmierung. Dabei wurden aus lange Zeit nur in der Theorie vorhandenen Vorteilen gegenüber ebenfalls zunehmend leistungsoptimierten CPU-basierten Lösungen schließlich reale Geschwindigkeitsgewinne, die neben der puren Leistungssteigerung der Hardware auch und gerade den zuvor beleuchteten neuen Möglichkeiten bei ihrer Programmierung zu verdanken sind [HSHH07, PGSS07, GPSS07].

In der vorliegenden Arbeit soll dieser Weg nachvollzogen und die Implementierung eines zugleich leistungsfähigen und flexiblen GPU-basierten Ray-Tracing-Systems unter Verwendung der CUDA-Entwicklungsumgebung von NVIDIA dokumentiert werden. Ziel ist, den neuartigen Zu-

[1] `http://www.nvidia.com/object/IO_43499.html`,
siehe auch
`http://www.nvidia.com/object/cuda_home.html`

gang zur Programmierung der GPU ganzheitlich zu erfassen und in Bezug auf die verfügbare Dokumentation, die Erlernbarkeit, die bereitgestellten Mittel zur Fehleranalyse sowie die erreichbare Leistung zu evaluieren. Um zu ermitteln, inwiefern sich eine unter Verwendung von CUDA verwirklichte Lösung praxisgerecht in ein bestehendes System integrieren läßt, ist vorgesehen, den erstellten Ray-Tracer als Modul in die an der Universität Koblenz entwickelte Echtzeit-Ray-Tracing-Umgebung Augenblick einzubinden. Einer aussagekräftigen Einordnung der Leistungsfähigkeit des entwickelten Systems dient ein Vergleich mit dem in Augenblick verfügbaren CPU-basierten Renderer im Rahmen einer umfassenden Auswertung. Die gewonnenen Erkenntnisse sollen eine fundierte Einschätzung der Möglichkeiten zulassen, welche die Verwendung moderner GPUs für Ray-Tracing unter Zuhilfenahme aktueller Entwicklungswerkzeuge mit sich bringt.

Kapitel 2

NVIDIA CUDA[1]

CUDA steht als Akronym für »Compute Unified Device Architecture«. Der Hersteller NVIDIA bezeichnet damit eine Kombination aus Hard- und Software, die eine Programmierung und Verwendung der GPU als allgemeinen datenparallel ausgelegten Prozessor erlaubt. Unter den Bezeichnungen GPGPU oder auch GPU-Computing hat eine solche vermeintliche Zweckentfremdung des Graphikprozessors zwar schon lange vor Einführung der neuen Architektur stattgefunden, jedoch besteht zwischen diesen bisherigen Ansätzen und CUDA ein elementarer Unterschied, der in den jeweiligen Vorgehensweisen zu suchen ist: Die traditionelle Praktik ist, daß die GPU als allgemeiner Coprozessor genutzt, hierfür aber immer noch als ursprünglich zweckgebundene GPU mit Hilfe einer Graphik-API programmiert wird – sei es direkt durch den Entwickler der GPGPU-Anwendung oder indirekt durch ein Werkzeug, welches die nach außen hin angebotenen Abstraktionen unter der Oberfläche in die Befehlssprache einer Graphik-API umsetzt. CUDA hingegen ermöglicht die Nutzung der GPU für allgemeine Zwecke durch eine ebenso verallgemeinerte Programmierung, die bis auf die Ebene der Hardware hinabreicht und an keiner Stelle mehr auf eine Graphik-API abgebildet wird.

Der Aufbau dieses Kapitels sieht zunächst eine kurze Einführung in die verwendete Nomenklatur sowie eine grundlegende Beschreibung der Arbeitsweise und Implementation CUDAs vor. Die konkreten Mittel zur GPU-Programmierung mit CUDA werden anschließend vorgestellt, indem auf die für die vorliegende Arbeit wesentlichen Komponenten der Entwicklungsumgebung eingegangen wird. Besonderheiten und gewisse Stra-

[1] Sofern nicht explizit anders angegeben, basieren die Angaben in diesem und den folgenden Kapiteln auf dem offiziellen NVIDIA-CUDA-Programmierleitfaden in der Version 2.0 [NVI08a] und beziehen sich auf die CUDA-Spezifikation 1.0 sowie dieser Spezifikation entsprechende Hardware. Soweit dies jedoch möglich und sinnvoll ist, werden in den Beschreibungen versionsabhängige Implementationsdetails in Annotationen ausgegliedert, um hiervon losgelöst die wesentlichen Inhalte der CUDA-Technik in den Fokus zu rücken.

tegien bei der Programmierung mit CUDA finden im Abschluß dieses Kapitels Erwähnung.

2.1 Grundlagen

Im Kontext CUDAs werden einige grundsätzlich beteiligte Elemente identifiziert und hierfür eine gewisse Nomenklatur definiert: Für die GPU bzw. die Graphikkarte wird der allgemeine Begriff *Device* eingeführt. Als Coprozessor wird das Device der CPU bzw. dem übrigen Computersystem zur Seite gestellt, welches demzufolge als *Host* dient und so bezeichnet wird. Funktionen, die zur Ausführung auf einem Device vorgesehen sind, stellen *Device-Code* dar, und analog handelt es sich bei Anweisungen für den Host um *Host-Code*. Da beim GPU-Computing die Berechnungen in aller Regel nicht für eine direkte graphische Darstellung durchgeführt und ihre Ergebnisse also nicht in den Framebuffer geschrieben werden, sondern für die weitere Verwendung auf dem Host vorgesehen sind, erhält hier insbesondere der Rücktransfer der Daten vom Device eine sehr viel größere Bedeutung als bei der konventionellen Graphikbeschleunigung. Ein essentielles Thema in CUDA sind also Kopiervorgänge zwischen System- und Graphikspeicher, weshalb auch diese Domänen in Form ihrer physikalischen Existenz als DRAM die entsprechenden Bezeichnungen *Host-Memory* und *Device-Memory* erhalten.

Im folgenden werden zunächst das logische Programmier- und Ausführungsmodell CUDAs und anschließend die konkrete Abbildung des Modells auf die Hardware vorgestellt. Diese Reihenfolge erlaubt es, früh ein Verständnis für das Konzept, auf dem CUDA basiert, aufzubauen und nachfolgend mit geschärftem Blick die für das GPU-Computing wesentlichen Elemente der Hardware zu identifizieren.

2.1.1 Konzept

Das CUDA zugrundeliegende Konzept ist die hochgradig parallele Ausführung eines Programms bzw. einer Funktion, in diesem Zusammenhang *Kernel* genannt, auf dem Device. Hierzu wird der Kernel von einer großen Anzahl sogenannter *Threads*, den wesentlichen Einheiten der Berechnung, simultan verarbeitet. Im Rahmen dieses allgemeinen Schemas existieren drei zentrale Abstraktionen, die das Programmier- und Ausführungsmodell CUDAs maßgeblich bestimmen:

Die erste Abstraktion stellt die hierarchische Organisation der elementaren Threads dar: Sie werden deterministisch zu gleichgroßen Bündeln oder Blöcken, den sogenannten *Thread-Blocks*, zusammengefaßt. Die Menge der Blocks bildet das sogenannte *Grid*. Dieses umfaßt alle Threads und repräsentiert so die parallele Ausführung des Kernels.

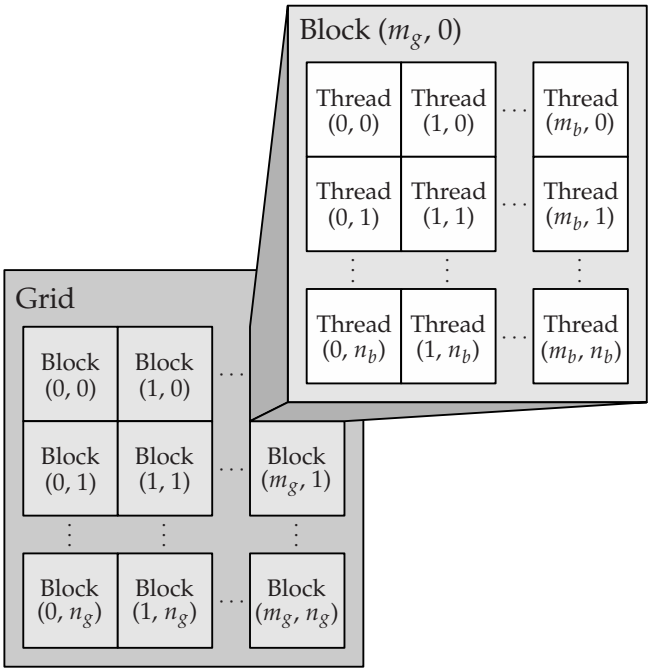


Abbildung 2.1: Thread-Hierarchie. In diesem Beispiel wird das Grid als zweidimensionales Feld der Breite m_g und Höhe n_g angenommen, das aus wiederum zweidimensionalen Thread-Blocks mit jeweils m_b Spalten und n_b Zeilen besteht. Aus dieser Konfiguration ergeben sich die angegebenen Indices der Blocks und Threads. Die Darstellung ist einer Abbildung im CUDA-Programmierleitfaden nachempfunden.

Größe und Layout des Grids, also Anzahl und Anordnung der Threads in einem Block sowie Anzahl und Anordnung der Blocks insgesamt, sind konstant für die Ausführung einer gesamten Kernel-Berechnung auf dem Device und bilden einen Teil der sogenannten *Konfiguration* der Ausführung. Das Grid wird wahlweise als ein- oder zweidimensionales Feld durch Angabe der Anzahl an Blocks in der jeweiligen Dimension angelegt. Die Gestalt der Blocks wiederum wird als ein-, zwei- oder auch dreidimensionales Feld durch die entsprechende Bestimmung der Anzahl an beinhalteten Threads festgelegt.[1] Jeder Thread-Block besitzt gemäß seiner Position im Grid einen dessen Dimensionalität entsprechenden Index und somit eine eindeutige Identifikationsnummer, die *Block-ID*. Diese ist im Fall eines eindimensionalen Grids mit dem Index identisch und läßt sich für ein zweidimensionales Grid aus dessen Ausmaßen und dem Index des Blocks be-

[1] Die Größe eines Grids beträgt in jeder Dimension maximal 65.535 Blocks. Die x- und y-Dimensionen eines Blocks können jeweils maximal den Wert 512, die z-Dimension maximal den Wert 64 zugewiesen bekommen. Unabhängig davon besteht ein Thread-Block aus maximal 512 Threads.

rechnen. Ebenso wird jeder Thread innerhalb eines Blocks eindeutig durch den Index seiner Position identifiziert, woraus sich schließlich analog seine *Thread-ID* ableiten läßt. Die Indices eines Threads und des ihn beinhaltenden Blocks sowie die Dimensionen des Grids und der Blocks sind während einer Kernel-Ausführung in Form vordefinierter Variablen verfügbar, so daß in einem Thread dessen eindeutige Position im Grid ermittelt werden kann. Abbildung 2.1 zeigt in schematischer Darstellung ein Beispiel für diese Organisation.

Die zweite wesentliche Abstraktion bezieht sich auf die verschiedenen Speicherräume, auf die bei der parallelen Berechnung zugegriffen werden kann. Sie unterscheiden sich hinsichtlich der Zugriffsstrategien und -optimierungen sowie in Bezug auf ihre Sichtbarkeit innerhalb der Thread-Hierarchie: Allen Threads gemeinsam, also gleichsam auf Grid-Ebene, stehen

Global Memory als Speicherraum für allgemeine Lese- und Schreibzugriffe an beliebigen Positionen,

Constant-Memory als größen[1]- und auf optimierte Lesezugriffe beschränkter Speicherraum für konstante Daten sowie

Texture-Memory als auf Lesezugriffe mit speziellen Optimierungen für Texturdaten beschränkter Speicherraum

zur Verfügung. Jeder Thread erhält weiterhin exklusiven Zugriff auf Ressourcen, die der Zwischenspeicherung temporärer Daten während einer Berechnung dienen. Dies sind

Register in limitierter Anzahl sowie

Local Memory als im Vergleich zur Menge der Register größerem, aber behäbiger angebundenem Speicherraum.

Hierarchisch genau zwischen diesen allgemeinen und exklusiven Speicherräumen ist auf Block-Ebene der *Shared Memory* angesiedelt; auf ihn haben alle Threads desselben Blocks Zugriff, während solche, die in unterschiedlichen Blocks organisiert sind, keine Berechtigung zum Lesen oder Beschreiben des jeweils in dem anderen Block zur Verfügung gestellten Shared Memorys besitzen.

Ein spezieller Synchronisationsmechanismus ist die dritte Schlüsselabstraktion CUDAs: Alle Threads desselben Blocks können durch das Setzen von Schranken synchronisiert werden. Zugriffe auf den Shared Memory lassen sich so koordinieren. Der damit einhergehende Aufwand ist ausdrücklich gering: Über die Zeit hinaus, die zwangsläufig vergeht, bis in

[1] Die Größe des Constant-Memorys beträgt 64 KiB.

allen betroffenen Threads die Synchronisationsschranke erreicht wird, entstehen keine nennenswerten Verzögerungen. Eine Option zur Synchronisation über die Grenzen eines Blocks hinweg für die Beeinflussung einer Kernel-Ausführung existiert hingegen nicht – Threads verschiedener Blocks werden unabhängig ohne Möglichkeit der Kommunikation untereinander bearbeitet.

Mit diesen drei zentralen Abstraktionen ist es CUDA möglich, seinem Ausführungsmodell ein Programmiermodell gegenüberzustellen, das sich einerseits in der größtmöglichen Weise unabhängig von der zugrundeliegenden Technik zeigt und andererseits deren spezielle Fähigkeiten aufgreift und an den Entwickler heranträgt. Zum Beispiel können die Blocks eines Grids in transparenter Weise je nach vorhandenen Ressourcen entweder parallel oder auch sequentiell verarbeitet werden, während auf der Ebene der Programmierung stets die Parallelität aller Berechnungen vorausgesetzt wird. Weiterhin können dank des differenzierten Speicher- und Synchronisationsmodells Probleme sinnvoll zerlegt werden in kleiner dimensionierte, von Threads kooperativ durchführbare Berechnungen, die sich wiederum unabhängig voneinander im größeren Maßstab des Grids zur Lösung des Gesamtproblems parallelisieren lassen.

Die Einsicht in das Konzept, das CUDA zugrundeliegt, ermöglicht im folgenden einen fokussierten Blick auf die Hardware, die diesem Modell zur Seite gestellt wird. Dabei bleiben zur Wahrung der Verständlichkeit verschiedene im GPU-Computing nicht relevante Aspekte der Funktionalität unbehandelt, da insbesondere die spezielle Aufgabe der Graphikbeschleunigung nicht Thema der vorliegenden Arbeit ist.

2.1.2 Die G80-Prozessorarchitektur[1]

Die Prozessorgeneration G80 des GPU-Spezialisten NVIDIA basiert auf einer Chiparchitektur, die sich von jener aller Vorgänger stark unterscheidet. Das bis dahin beständige Konzept der Aufteilung in verschiedene Berechnungseinheiten für spezielle Aufgaben, namentlich Vertex- und Pixel- bzw. Fragment-Shader, wird inhaltlich und hinsichtlich der Benennung fallengelassen zugunsten einer Vereinheitlichung, der sogenannten »Unified-Shader«-Architektur. Ersetzt werden die spezialisierten Shader-Einheiten durch einen universellen Prozessortyp, der alle zuvor verteilten Funktionalitäten in sich vereint und über einen entsprechend generalisierten Befehlssatz angesprochen wird. NVIDIA bezeichnet diese verallgemeinerte Form der wesentlichsten Recheneinheit der GPU als *Stream-Processor* (SP) und identifiziert damit die Bedeutung dieses Elements auch in Bezug auf die

[1] Sofern nicht explizit anders angegeben, beziehen sich konkrete Angaben zur Hardware in diesem und den folgenden Kapiteln auf die vollausgestattete Chipvariante des G80, wie sie auf Graphikkarten der Typen GeForce 8800 GTX und GeForce 8800 Ultra zum Einsatz kommt.

Verwendung der GPU für nicht-graphische Zwecke. Es handelt sich dabei um ALUs für Operationen auf Gleitkommazahlen[1,2] sowie erstmals auch Ganzzahlen in jeweils skalarer Ausführung. Dieses Layout bedeutet einen weiteren wesentlichen Unterschied zu Shadern, die in aller Regel als Vektoreinheiten, meist vierdimensional, konzipiert sind, jedoch de facto keinen Rückschritt hinsichtlich der Parallelität, die in dem neuen Design auf höherer Ebene hergestellt wird: Ein sogenannter *Streaming Multiprocessor* (SM) faßt eine definierte Anzahl der skalaren SPs zusammen und bildet hieraus einen Verbund, der in einem dem SIMD[3]-Prinzip entlehnten Layout organisiert ist, auf dessen genaue Bedeutung später in Kapitel 2.1.4 eingegangen wird. Auf dieser Hierarchieebene, also im Verbund eines SMs, befinden sich weiterhin spezielle für Berechnungen mit transzendenten Zahlen ausgelegte Rechenwerke sowie eine Befehlskontrolleinheit, welche die von den Prozessoren auszuführenden Instruktionen koordiniert. Eine schematische Darstellung des Aufbaus eines SMs zeigt Abbildung 2.2.[4]

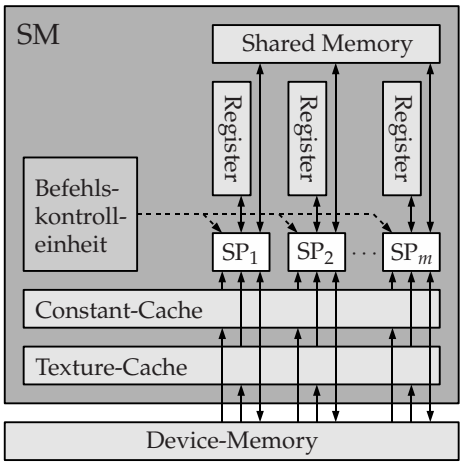


Abbildung 2.2: Streaming Multiprocessor. Im Mittelpunkt stehen die vielfältigen Speicheranbindungen der m SPs; keine Berücksichtigung finden hier die Spezialprozessoren für die Berechnung transzendenter Funktionen. Diese Darstellung ist einer Abbildung im CUDA-Programmierleitfaden nachempfunden.

[1] NVIDIA stuft die Behandlung von Gleitkommazahlen CUDA-fähiger Devices als konform mit dem Standard IEEE 754 [ICSI85] ein; Ausnahmefälle werden explizit angegeben. Diese klare Spezifikation ist eine wichtige Voraussetzung zum Beispiel für den von NVIDIA beworbenen Einsatz CUDAs im Bereich wissenschaftlicher Berechnungen.

[2] Bis CUDA-Spezifikation 1.2 werden Gleitkommazahlen nur in einfacher, ab Spezifikation 1.3 auch in doppelter Genauigkeit unterstützt.

[3] »Single Instruction, Multiple Data«: Derselbe Befehl wird parallel auf unterschiedlichen Daten ausgeführt.

[4] Der G80-Chip besitzt 16 SMs, in denen jeweils 8 SPs zu einem Verbund zusammengefaßt sind. Insgesamt stehen somit 128 SPs für parallele Berechnungen zur Verfügung.

Den Berechnungseinheiten des SMs steht Speicher aus zwei verschiedenen Domänen zur Verfügung: dem bereits erwähnten allgemeinen Device-Memory sowie einem vergleichsweise kleinen, in jedem SM lokal bereitgestellten Chipspeicher, dem sogenannten *On-Chip Memory*. Zugriffe der SPs auf den Device-Memory sind grundsätzlich mit hohen Latenzzeiten verbunden, solche auf den lokalen Chipspeicher werden hingegen äußerst schnell bearbeitet. Ein SP kann ausschließlich auf den On-Chip Memory desjenigen SMs zugreifen, zu dessen Verbund er gehört, während der Device-Memory in technischer Hinsicht jedem SP gleichermaßen zur Verfügung steht. Gegliedert ist der lokale Chipspeicher in

Register einer Breite von 32 Bit, die gleichverteilt jedem SP des SMs separat für Lese- und Schreiboperationen zur Verfügung stehen,

Shared Memory als parallel organisierten Speicherraum, auf den alle SPs des SMs gemeinsam lesend und schreibend zugreifen können,

Constant-Cache zur Beschleunigung der Lesezugriffe auf den Constant-Memory sowie

Texture-Cache zur Beschleunigung der Lesezugriffe auf den Texture-Memory.[1]

Verglichen mit den Größen der Caches moderner CPU-Architekturen handelt es sich beim On-Chip Memory eines SMs bzw. dem Chipspeicher der GPU insgesamt um nur kleine Speicherräume. Daß dieser Prozessor in der Hauptsache aus parallelen Recheneinheiten besteht und vorsätzlich nicht aus möglichst großen effizient angebundenen Speichern, wie sie in CPUs von hoher Bedeutung sind, dokumentiert seine Spezialisierung auf die Lösung berechnungsintensiver Aufgaben. Die Konsequenzen, die sich aus dieser Auslegung ergeben, werden später ausführlich in Kapitel 2.3.1 diskutiert.

2.1.3 Das Modell auf der Hardware

Gemäß dem Konzept CUDAs finden sich im Design der Hardware zahlreiche grundsätzliche Analogien zu den zuvor erwähnten Elementen des Programmier- und Ausführungsmodells. Eine starre Kopplung des Programmierparadigmas an eine konkrete technische Implementation wird indes konsequent vermieden: Einerseits soll Software-Entwicklern ein hinreichend universell verwendbares Werkzeug zur Verfügung stehen; andererseits soll das Modell langlebig sein, so daß es auf verschiedenen erhältlichen und zukünftigen Hardware-Lösungen anwendbar bleibt und mit deren unterschiedlicher Leistungsfähigkeit skaliert.

[1] Ein SM der CUDA-Spezifikation 1.0 verwaltet 8.192 Register sowie 16 KiB Shared Memory, 8 KiB Constant-Cache und zwischen 6 und 8 KiB Texture-Cache. In der Spezifikation 1.2 verdoppelt sich die Anzahl der Register je SM auf 16.384.

Es gilt also, die Logik des Modells auf die beschriebenen Voraussetzungen der Hardware adäquat abzubilden. Eine direkte Entsprechung findet der Thread dabei in einem SP: Als die wesentliche Berechnungseinheit im CUDA-Modell wird ein Thread von dem die allgemeinen Berechnungen durchführenden Hardware-Element, dem SP, verarbeitet. Dies geschieht parallel in Thread-Gruppen einer festen Größe, sogenannten *Warps*, unter Ausnutzung der SIMD-ähnlichen Architektur des SMs.[1] Erstellt, also aus zu bearbeitenden Threads zusammengesetzt, und koordiniert werden die Warps von einer jedem SM eigenen Thread-Verwaltungseinheit. Während jedoch in einer Anwendung nahezu beliebig viele Threads ausgeführt werden können, sind die Anzahl verfügbarer SPs und ihre konkrete Organisation in SMs notwendigerweise feste Konstanten der jeweiligen Hardware. Überdies können im CUDA-Modell das Layout des Grids sowie das der zugehörigen Blocks flexibel gewählt werden. Das Schema der Verarbeitung eines Thread-Blocks sowie der Verwaltung aller Blocks eines Grids erfordert also eine tiefergehende Erläuterung:

Gut nachvollziehbar wird die tatsächliche Bearbeitung eines Thread-Blocks bei Betrachtung der Umsetzung des Shared-Memory-Konzepts: Um von den schnellen Zugriffszeiten profitieren zu können, ist der Shared Memory physikalisch im On-Chip Memory angesiedelt. Da er als logischer Speicherraum exklusiv denjenigen Threads offensteht, die zu demselben Block gehören, und auf den On-Chip Memory nur diejenigen SPs zugreifen können, die in demselben SM zusammengefaßt sind, kann ein Thread-Block zwingend nur einem SM zur vollständigen Berechnung zugewiesen werden und nicht etwa von mehreren SMs kooperativ verwaltet werden. Das bedeutet gleichzeitig, daß alle Threads desselben Blocks allein von den SPs desjenigen SMs verarbeitet werden, dem der Thread-Block zugewiesen wird. Ein Grid wird also bearbeitet, indem die logischen Blocks den verfügbaren SMs exklusiv zugewiesen werden. Ein Block schließlich wird von dem ihn verarbeitenden SM deterministisch in Warps zerlegt, deren Threads in einem Zeitmultiplexverfahren von den SPs berechnet werden.

Eine CUDA-Anwendung nimmt keine Rücksicht darauf, wieviele SMs von der Hardware bereitstellt werden; ein Grid kann aus sehr viel mehr Thread-Blocks bestehen, als SMs physikalisch vorhanden sind. Mit einer solchen Überzahl an Blocks wird auf zwei Weisen umgegangen: Bis zu einer konstanten Höchstanzahl werden von einem SM gleichzeitig mehrere Thread-Blocks verwaltet, je nachdem für wieviele Blocks die Ressourcen

[1] Bis zur aktuellen CUDA-Spezifikation 1.3 ist die Warp-Größe unverändert auf 32 Threads festgelegt. Diese Definition wird von NVIDIA nicht begründet. Ob die Hardware – beispielsweise durch unterschiedliche Taktdomänen, Pipelining etc. – die Größe zwingend bedingt oder ob sie das Ergebnis einer Effizienzstrategie ist, um zum Beispiel gewissen Latenzzeiten beim Laden von Instruktionen durch ein besseres Verhältnis von tatsächlichen Berechnungen zu diesen Ladevorgängen zu begegnen, kann daher nicht verläßlich angegeben werden.

des SMs, namentlich die Anzahl der Register und die Größe des Shared Memorys, ausreichen. Solche einem SM zugeteilten und somit in Berechnung befindlichen Blocks werden als *aktiv* bezeichnet, und analog dazu handelt es sich bei ihrer Zerlegung um aktive Warps und entsprechend aktive Threads.[1] Sobald ein SM die Bearbeitung eines Thread-Blocks fertiggestellt hat, wird ihm aus der Reihe der noch nicht einem SM zugeteilten Blocks ein neuer zugewiesen.

Die gleichzeitige Verwaltung mehrerer Thread-Blocks durch einen SM bedarf einer genaueren Erklärung, denn hierin manifestiert sich ein Schlüsselkonzept CUDAs hinsichtlich der Effizienz: die bestmögliche Nutzung aller vorhandenen SMs. Damit die Rechenkapazitäten eines SMs nicht brachliegen, während zum Beispiel mit hohen Latenzzeiten verbundene Zugriffe auf den Device-Memory auszuführen sind, werden diese Taktzyklen idealerweise anderen Berechnungen gewidmet. Aktive Warps, deren Verarbeitung nicht fortgeführt werden kann, solange beispielsweise die Ergebnisse besagter Speicheroperationen ausstehen, können hintangestellt werden, wenn der SM währenddessen andere nicht derart blockierte Warps berechnen kann. Dies ist umso wahrscheinlicher der Fall, je mehr aktive Threads dem SM zur Berechnung zur Verfügung stehen, was wiederum von Größe und Anzahl der ihm zur Verarbeitung zugewiesenen Thread-Blocks abhängig ist. Hier kündigt sich bereits eine Möglichkeit an, wie Anwendungen optimiert werden können, um die Fähigkeiten CUDAs und der Hardware bestmöglich auszunutzen – dieser und weitere Ansätze werden später in Kapitel 2.3 vorgestellt.

Speicherraum			Zugriff[a]	
logisch	physikalisch	Zugriffsebene	Host	Device
Register	On-Chip M.	Thread		l/s
Local M.	Device-M.	Thread		l/s
Shared M.	On-Chip M.	Block		l/s
Global M.	Device-M.	Grid	l/s	l/s
Constant-M.	Device-M.	Grid	l/s	l
Texture-M.	Device-M.	Grid	l/s	l

[a] l/s: lesend/schreibend (jeweils uneingeschränkt)

Tabelle 2.1: Speicherräume

[1] Möglich sind je SM maximal 8 aktive Thread-Blocks. Weiterhin legt die CUDA-Spezifikation 1.0 die maximale Anzahl an aktiven Warps je SM auf 24 und daraus resultierend die maximale Anzahl an aktiven Threads auf 768 fest. Ab Spezifikation 1.2 sind bis zu 32 aktive Warps und somit maximal 1.024 aktive Threads je SM möglich.

Die Zusammenhänge zwischen den logischen Speicherräumen CUDAs und den physikalisch verfügbaren Speichern sind Tabelle 2.1 zu entnehmen. Darin wird auch angegeben, auf welcher Ebene der Thread-Hierarchie auf die logischen Speicherbereiche zugegriffen werden kann und welche Operationen hierbei von seiten des Hosts und des Device jeweils erlaubt sind.

Es ergibt sich aus der Tabelle, daß der Global Memory der einzige Speicherraum ist, der Host und Device gemeinsam zu Lese- und Schreiboperationen offensteht. Diese Flexibilität geht jedoch mit dem Nachteil einher, daß für den wichtigen Zugriff seitens des Device keine Beschleunigungsstrategien vorhanden sind. Die aufgrund der dort verfügbaren Cache-Mechanismen prinzipiell schnelleren Zugriffe auf Constant-Memory und Texture-Memory können nur noch lesend ausgeführt werden.

Alle grundsätzlich erlaubten Operationen auf die jeweiligen Speicherbereiche sind in keiner Weise beschränkt, d.h., sie sind an beliebigen Adressen des logischen Speicherraums erlaubt. Dies ist bemerkenswert insbesondere im Hinblick auf die für den Global Memory zulässigen Zugriffsarten: Im Gegensatz zum freien Lesen, dem sogenannten *Gathering*, das in Verbindung mit Texturen seit je her zu den fundamentalen GPGPU-Operationen zählt, ist ein uneingeschränktes Schreiben, genannt *Scattering*, mit bisherigen Graphik-APIs nur umständlich unter Zuhilfenahme der für gewöhnlich nicht zu GPGPU-Zwecken verwendeten Vertex-Shader umsetzbar.

Eine weitere wichtige Erkenntnis aus Tabelle 2.1 betrifft die Thread-eigenen Speicher in Form der Register und des Local Memorys, die einmal im schnellen On-Chip Memory, im anderen Fall im langsam angebundenen Device-Memory angesiedelt sind: Aufgrund der unterschiedlichen Implementation dieser beiden Speicherräume ergeben sich für die Nutzung des Local Memorys große Leistungseinbußen gegenüber der Verwendung von Registern. Hier existiert auch kein Cache, der die langsamen Zugriffsgeschwindigkeiten auf den Device-Memory unter gewissen Voraussetzungen beschleunigen würde. Tatsächlich handelt es sich beim Local Memory um einen Auslagerungsspeicher in der Hinsicht, daß große Strukturen, die zuviele der schnellen, aber in ihrer Anzahl limitierten Register belegen würden, in diesem ungleich größeren Speicherraum abgelegt werden; dies betrifft insbesondere Arrays. Die Entscheidung darüber, ob eine derartige Struktur in Registern des SMs oder im Local Memory vorgehalten wird, trifft der Compiler zum Zeitpunkt der Übersetzung des GPU-Programms und kann vom Entwickler nicht direkt beeinflußt werden.

2.1.4 Das SIMT-Prinzip

Alle SPs eines SMs führen zeitgleich denselben vom SM bestimmten Befehl aus, weshalb es sich bei diesem Verbund technisch in gewisser Weise um einen Prozessor nach dem SIMD-Prinzip handelt. Die gebündelte Bearbei-

tung von Threads in Form der Warps bedeutet in diesem Zusammenhang lediglich eine implizite weitere Verbreiterung der Architektur: Ein SM kann als Recheneinheit betrachtet werden, die Daten auf SIMD-Weise verarbeitet in einer Breite, die der Warp-Größe entspricht. Das Programmiermodell CUDAs unterscheidet sich jedoch in wichtigen Aspekten von jenem klassischer SIMD-Architekturen: An keiner Stelle in CUDA wird ein derartiges Schema der datenparallelen Berechnung explizit an den Entwickler herangetragen – insbesondere werden hier keinerlei Datentypen auf Basis der Warp-Größe definiert und angeboten, wie dies in aller Regel bei SIMD-Implementationen der Fall ist, und wenn auch die Berücksichtigung der konkreten Anzahl der in einem Warp zusammengefaßten Threads bei der Entwicklung möglich ist[1] und aufgrund der erzielbaren Wirkung empfohlen wird, bleibt dies eine Option und keine Notwendigkeit.

Der grundlegende Unterschied zwischen CUDA und typischen SIMD-Architekturen besteht in der skalaren Natur der SPs und dem auf dieser Tatsache konsequent aufbauenden Programmiermodell: Es ist gerade nicht der SM, der einen Thread verarbeitet, indem er in SIMD-Weise mit seinen SPs Berechnungen auf den Elementen eines mehrdimensionalen Datensatzes ausführt; vielmehr berechnet jeder SP einen separaten Thread, was je nach zugrundeliegendem Kernel denselben Effekt wie eine datenparallele Verarbeitung starr nach dem SIMD-Prinzip haben oder aber im Resultat der parallelen Verfolgung gänzlich unterschiedlicher Programmpfade entsprechen kann. Die Anweisungen eines Kernels bestimmen die Ausführung eines einzelnen möglicherweise völlig unabhängigen Threads und nicht etwa die Verarbeitung von Datensätzen einer gewissen Größe.

NVIDIA beschreibt diese Architektur als »Single Instruction, Multiple Thread« oder kurz SIMT, um den Unterschied der parallelen Ausführung auf Thread-Ebene gegenüber einer rein datenparallelen SIMD-Arbeitsweise hervorzuheben. Das Modell ist gleichsam eine Abstraktion der SIMD-Funktionalität: Tatsächlich führen alle SPs des SMs weiterhin gleichzeitig und für alle Threads eines Warps denselben Befehl aus – wie verfahren wird, wenn ein Kernel für Threads desselben Warps divergierende Ausführungspfade veranlaßt, findet später in Kapitel 2.3.2 Erläuterung. Dennoch bleibt diese spezielle Art und Weise der parallelen Thread-Verarbeitung transparent; das CUDA-Programmiermodell abstrahiert von der beschriebenen tatsächlichen Implementation des Ausführungsmodells und erlaubt so auf der Ebene der Anwendungsentwicklung eine flexible Programmierung auf der Basis von Threads, anstatt eine vergleichsweise starre SIMD-Architektur an den Entwickler heranzutragen.

[1] Die Warp-Größe kann zur Laufzeit aus einer in CUDA vordefinierten Variablen ausgelesen werden.

2.2 Die CUDA-Entwicklungsumgebung[1]

Die Entwicklung mit CUDA umfaßt verschiedene Vorgänge: die Programmierung von Kernels, die Organisation ihrer Aufrufe und der hierfür benötigten Ressourcen, die gezielte Übersetzung von Device-Code und Host-Code sowie die Verknüpfung der erstellten Objekte zu einem lauffähigen Programm. Für alle diese Prozesse und Aufgaben existieren in der CUDA-Entwicklungsumgebung Komponenten, die sich grob in drei Klassen einteilen lassen:

Werkzeuge erlauben die Übersetzung des Quelltextes einer CUDA-Anwendung sowie eine Analyse des Laufzeitverhaltens;

Erweiterungen der Programmiersprache C lassen gewisse notwendige Differenzierungen bei der GPU-Programmierung zu;

eine Laufzeitbibliothek stellt APIs[2] zur Einbindung eines Device sowie allgemeine Datentypen und Funktionen bereit.

Im folgenden wird die CUDA-Entwicklungsumgebung in einem Umfang beschrieben, der sowohl eine sinnvolle allgemeine Einsicht in ihre Elemente und Möglichkeiten erlaubt als auch für das Verständnis dieser Arbeit notwendig ist; darüber hinaus müssen viele Aspekte insbesondere der umfangreichen Laufzeitbibliothek unbehandelt bleiben.

2.2.1 Werkzeuge

Zu den Werkzeugen, die in der CUDA-Entwicklungsumgebung angeboten werden, zählen in dieser Übersicht nicht nur der Compiler-Treiber als einzige konkrete Anwendung, sondern auch gewisse Ausführungsmodi, die in der Entwicklungsphase wertvolle Unterstützung bei der Fehlerbehebung und Optimierung geben können.

Compiler-Treiber

Das elementare Werkzeug, das CUDA für die Übersetzung des Quelltextes bereitstellt, ist der Compiler-Treiber *nvcc*. Es handelt sich dabei um eine Anwendung zur flexiblen Steuerung und Ausführung aller notwendigen

[1] NVIDIA bietet neben der hier vorgestellten CUDA-Entwicklungsumgebung ein sogenanntes CUDA-Entwickler-SDK [NVI08b] an. Dabei handelt es sich um eine Sammlung von Quelltextbeispielen, jedoch explizit nicht um einen Bestandteil der Entwicklungsumgebung mit ihren für die Erstellung und Ausführung von Programmen benötigten Komponenten.

[2] Es stehen zwei APIs zur Verfügung: eine systemnahe Driver-API und eine darauf aufbauende Runtime-API, in der zahlreiche grundlegende Prozesse automatisiert werden. Eine gemeinsame Verwendung beider APIs ist nicht erlaubt.

Kompilierungs- und Verbindungsvorgänge, dessen Verwendungsschema zu großen Teilen jenem des GNU-Compilers gcc nachempfunden ist.

CUDA räumt dem Entwickler bei der Organisation des Quelltextes eines Projekts dieselbe Freiheit ein, die in der klassischen C/C++-Programmierung üblich ist: Host- und Device-Code können nach Bedarf getrennt oder auch in derselben Datei formuliert werden. Eine Kernaufgabe des Compiler-Treibers ist deshalb, den Quelltext einer Datei zu separieren, also die zur Ausführung auf einem Device oder dem Host vorgesehenen Anteile jeweils zu identifizieren und zu extrahieren. Anschließend koordiniert die Anwendung die jeweilige Kompilierung, wozu gegebenenfalls weitere Werkzeuge eingesetzt werden. Die resultierenden Binärobjekte werden im letzten Schritt zu einem lauffähigen Programm verknüpft. All diese Prozesse können von der Anwendung nvcc transparent abgewickelt werden; alternativ lassen sich auch einzelne Vorgänge separat konfigurieren und ausführen, worauf in dieser Arbeit aber nicht eingegangen wird.

Zur Übersetzung des Host-Codes wird ein auf dem Entwicklungssystem bereitgestellter C/C++-Compiler eingesetzt. Dabei bietet der nvcc die wichtige Möglichkeit, beim Aufruf übergebene Argumente an diesen Standard-Compiler zu übermitteln, wodurch sich für die Kompilierung von Host-Code, der wegen CUDA-spezifischer Inhalte zunächst vom nvcc vorverarbeitet werden muß, keinerlei Einschränkungen ergeben.

Device-Emulation

Sofern nicht die streng maschinenorientierte Driver-API zur Programmierung verwendet wird, kann der Compiler-Treiber veranlaßt werden, bei der Übersetzung Vorkehrungen zu treffen, um die Device-seitigen Berechnungen nicht auf einer existierenden GPU, sondern in einer Emulation ausschließlich auf dem Host durchzuführen. Das Verfahren läßt damit den Einsatz aller diagnostischen Mittel, die sonst nur den Host-basierten Anteil der Ausführung erfassen können, für die gesamte Programmabwicklung zu. Da die Entwicklungsumgebung für Programmteile, die auf dem Device ablaufen, keine eigenen Debugging-Mechanismen zur Verfügung stellt, kommt der Device-Emulation im Entwicklungsprozeß eine wesentliche Bedeutung zu. Sie unterstützt deshalb aktiv die Fehlersuche und -behebung: In internen Überprüfungen werden spezielle Fehlersituationen identifiziert, die bei der Ausführung auf dem Device unentdeckt bleiben und unerwünschte Effekte zur Folge haben können.

Verwirklicht wird die Device-Emulation dadurch, daß für jeden Thread eines Grids mit den Mitteln des Betriebssystems ein neuer Thread auf dem Host erstellt wird. Es ergibt sich aus dieser Vorgehensweise, daß die Emulation die Ausführung von CUDA-Programmen auch auf Systemen ermöglicht, in denen keine geeignete Hardware von NVIDIA verfügbar ist. Allerdings ist dieser Modus in keiner Hinsicht auf hohe Leistungsfähigkeit ab-

gerichtet, sondern dient explizit dem erwähnten Zweck der Programmanalyse; bereits das sehr unterschiedliche Schema, mit dem Threads in CUDA auf der GPU und in einem Betriebssystem auf dem Host erstellt, zugewiesen, koordiniert und verarbeitet werden, erlaubt gemeinsam mit den von CUDA grundsätzlich favorisierten sehr hohen Thread-Zahlen keine Entwicklung, die ein Programm zum Ziel hat, das in der Device-Emulation leistungsorientiert zum Einsatz kommt.

Profiling

Eine Kernel-Ausführung kann von einem Profiler protokolliert werden, ohne daß hierfür eine Vorbereitung in der Übersetzungsphase notwendig ist. Die Aufzeichnung wird für die Kernel-Berechnungen eines CUDA-Programms aktiviert, indem vor Programmstart eine entsprechende Umgebungsvariable gesetzt wird. Den Bedürfnissen anpassen läßt sich die Protokollierung in einer Konfigurationsdatei durch die Angabe der Ereignisse, die bei der Messung zu berücksichtigen sind.

Das Profiling wird für einen einzelnen SM der GPU durchgeführt. Das bedeutet, daß nur solche Kernel-Ausführungen brauchbare Ergebnisse liefern können, die aus einer hinreichend großen Menge an Thread-Blocks bestehen – nur so ist eine gleichmäßige Verteilung auf die verfügbaren SMs möglich, und die Messung in einem einzelnen SM kann als repräsentativ für die gesamte Kernel-Berechnung angenommen werden. Die Ergebnisse des Profilings stellen daher kein exaktes Protokoll einer Kernel-Ausführung dar; vielmehr ermöglichen sie im Entwicklungsprozeß eine Orientierung, indem relative Veränderungen Hinweise auf den Erfolg von Optimierungen geben.

2.2.2 Erweiterungen der Programmiersprache C

Die GPU-Programmierung mit CUDA erfolgt in einem Dialekt der Programmiersprache C – bis auf wenige Ausnahmen, zum Beispiel rekursive Funktionsaufrufe, die Deklaration von statischen Variablen oder die Verwendung von Funktionszeigern, sind in Device-Code sämtliche Sprachkonstrukte erlaubt, die mit C einhergehen. Zur Einbeziehung des Device in die Programmierung werden behutsam Erweiterungen definiert, mit denen notwendige Differenzierungen vorgenommen werden können:

Funktionsattribute legen fest, ob eine Funktion zur Ausführung auf dem Host oder dem Device vorgesehen ist sowie ob sie von seiten des Hosts oder des Device aufgerufen werden kann;

Variablenattribute bestimmen, für welchen Speicherraum des Device eine Variable vorgesehen ist;

vordefinierte Variablen erlauben Einsicht in die Parameter der Konfiguration einer Kernel-Ausführung;

spezielle Syntax ermöglicht die Bestimmung der Konfiguration einer Kernel-Ausführung.

Darüber hinaus werden zur allgemeinen Vereinfachung der Programmierung zusätzliche Spracherweiterungen eingeführt, auf die am Ende dieses Kapitels eingangen wird.

Funktionsattribute

Attribut	ausführbar auf	aufrufbar auf
`__host__`	Host	Host
`__device__`	Device	Device
`__global__`	Device	Host

Tabelle 2.2: Funktionsattribute

```
1  // implizit: Aufruf und Ausführung auf Host
2  int hostFunc_1(int i);
3
4  // explizit: Aufruf und Ausführung auf Host
5  __host__ int hostFunc_2(int i);
6
7  // Aufruf und Ausführung auf Device
8  __device__ int deviceFunc(int i);
9
10 // Aufruf und Ausführung auf Host und Device
11 __host__ __device__ int commonFunc(int i);
12
13 // Aufruf auf Host, Ausführung auf Device (Kernel)
14 __global__ void kernel(int i);
```

Codebeispiel 2.1: Funktionsdeklarationen

Tabelle 2.2 listet die verfügbaren Funktionsattribute und ihre Bedeutung auf. Das Attribut `__host__` an sich ist redundant, da es dieselbe Bedeutung hat wie eine Funktionsdeklaration ohne eines der von CUDA eingeführten Attribute. Es kann jedoch in derselben Deklaration kombiniert werden mit dem Attribut `__device__`, um eine Funktion für Ausführung und Aufruf sowohl auf dem Host als auch auf dem Device kompilieren zu lassen. Das Attribut `__global__` schließlich kennzeichnet eine Funktion, die zur Ausführung als Kernel vorgesehen ist. Dieser Funktionstyp darf nur mit dem Rückgabetyp `void` deklariert werden – die Ergebnisse einer Kernel-Berechnung werden bei Bedarf im Device-Memory hinterlegt, auf

den der Host über spezielle Mechanismen der API zugreifen kann. Eine Auswahl konkreter Funktionsdeklarationen zeigt Codebeispiel 2.1.

Variablenattribute

Attribut[a]	Speicherraum	Deklaration[b]
–	Register/Local M.	Device-Code
`__device__`	Global M.	dateiweit
`__constant__`	Constant-M.	dateiweit
`__shared__`	Shared M.	Device-Code

[a] –: keines der in CUDA verfügbaren Variablenattribute

[b] gibt an, wo bzw. mit welcher Sichtbarkeit die Deklaration erfolgen muß

Tabelle 2.3: Variablenattribute

Tabelle 2.3 sind die in CUDA verfügbaren Variablenattribute zu entnehmen, die auf einem Device den zu verwendenden Speicherraum bestimmen. Eine Deklaration ohne Attribut im Host-Code besitzt dahingehend keine Bedeutung – die Speicherverwaltung solcher Variablen obliegt dem Host-System und wird von CUDA nicht beeinflußt. Die in der Tabelle aufgeführte explizit nicht attributierte Deklaration gilt deshalb allein für den Device-Code.

```
1  __device__ int d;     // Global Memory
2  __constant__ int c;   // Constant-Memory
3
4  __device__ int function(void)
5  {
6     int r;       // Register oder Local Memory (unwahrscheinlich)
7     int l[256];  // Register (unwahrscheinlich) oder Local Memory
8
9     __shared__ int s;           // Shared Memory (statisch)
10    __shared__ extern int a[];  // Shared Memory (dynamisch)
11 }
```

Codebeispiel 2.2: Variablendeklarationen

Das Codebeispiel 2.2 zeigt die Anwendung der Variablenattribute. Die dort in Zeile 10 angegebene Deklaration demonstriert eine spezielle Art und Weise, auf die Speicherbereiche im Shared Memory reserviert werden können: Das Schlüsselwort `extern` in der Deklaration bewirkt, daß ein so angelegtes Array eine variable Größe besitzt; bestimmt wird sie zur Laufzeit für eine Kernel-Ausführung durch ein der Konfiguration beigefügtes Argument. Diese Option stellt eine Alternative zur sonst notwendi-

gen Festlegung von Datenfeldgrößen dar und erlaubt eine flexible Anpassung von Kernels an individuelle Einsatzszenarios. Für die anderen Speicherräume auf dem Device neben dem Shared Memory besteht hingegen keine Möglichkeit zur dynamischen Speicherverwaltung.

Vordefinierte Variablen

Die wichtigsten in CUDA vordefinierten Variablen sind jene vier, die es ermöglichen, innerhalb eines Kernels Informationen über die aktuelle Ausführung abzufragen: In `gridDim` und `blockDim` sind Größe und Layout des Grids bzw. der Blocks hinterlegt; `threadIdx` und `blockIdx` geben jeweils den Index des gerade verarbeiteten Threads im Block bzw. des ihn beinhaltenden Blocks im Grid an. Diese Variablen sind für das CUDA-Konzept von elementarer Bedeutung: In den Threads einer Kernel-Ausführung lassen sich hiermit deren individuelle Positionen im Grid und in dem jeweiligen Block ermitteln. Ein Kernel kann mit ihrer Hilfe derart programmiert werden, daß in unterschiedlichen Threads verschiedene Programmpfade verfolgt werden. Ein häufig wiederkehrendes Szenario ist zum Beispiel, daß eine als `__shared__` deklarierte Variable nur in einem einzelnen Thread eines Blocks initialisiert und anschließend in allen Threads des Blocks verwendet wird.

Syntax zur Kernel-Konfiguration

Um für die Ausführung eines Kernels die gewünschte Konfiguration anzugeben, führt CUDA eine spezielle Syntax ein, die für den Aufruf einer mit dem Attribut `__global__` deklarierten Funktion, also eines Kernels, zu verwenden ist. Danach werden die Argumente, welche die Konfiguration bestimmen, von den Symbolfolgen `<<<` und `>>>` eingefaßt zwischen Funktionsbezeichner und der Liste der sonstigen Argumente aufgeführt. Das später auf Seite 24 gegebene Codebeispiel 2.3 zeigt unter anderem die Verwendung dieser Syntax.

Die Angabe der Konfiguration beinhaltet mindestens zwei Argumente, die jeweils Größe und Layout des Grids bzw. der Thread-Blocks beschreiben. CUDA stellt hierfür einen geeigneten Strukturtyp zur Verfügung, der später in Kapitel 2.2.3 vorgestellt wird. Um neben der statischen auch die erwähnte spezielle dynamische Verwaltung des Shared Memorys zu ermöglichen, muß außerdem die Größe des hierfür vorgesehenen Speicherbereichs durch ein zusätzliches Argument der Konfiguration bestimmt werden; fehlt diese Angabe, wird diesbezüglich kein Speicher reserviert. Alle Argumente eines Kernel-Aufrufs, also seine Konfiguration ebenso wie die Funktionsargumente, werden im Shared Memory der SMs abgelegt, was schnelle Zugriffe gewährleistet.

Sonstige Spracherweiterungen

CUDA erlaubt zahlreiche Sprachkonstrukte, die nicht im C-Standard vorgesehen sind, sondern auf Fähigkeiten der Sprache C++ beruhen: Hierzu gehören unter anderem das Überladen von Funktionen und Operatoren, Template-Mechanismen, Methoden, Namensräume, Referenzen, der boolesche Typ `bool` oder auch das Schlüsselwort `class`. Die Konzepte der Objektorientierung wie zum Beispiel Vererbung finden jedoch keine Unterstützung. Mit diesen Fähigkeiten läßt sich die in CUDA verwendete Programmiersprache gleichsam als C++ ohne Objektorientierung auffassen.[1]

2.2.3 Laufzeitbibliothek

Die Laufzeitbibliothek CUDAs setzt sich aus verschiedenen Komponenten zusammen: Zur universellen Verwendung sowohl in Host- als auch in Device-Code stellt sie gewisse Datentypen sowie eine Reihe von Standardfunktionen bereit. Daneben bietet sie in Form ihrer beiden APIs eine Sammlung von Funktionen zum Aufruf auf dem Host an, die den Zugriff auf ein Device ermöglichen und damit als Bindeglied zwischen Host und Device fungieren. Ebenfalls Teil der Laufzeitbibliothek sind gewisse Funktionen, die zum ausschließlichen Einsatz in Device-Code vorgesehen sind.

In den folgenden Abschnitten dieses Kapitels werden die Komponenten der Laufzeitbibliothek im einzelnen beleuchtet und ihre jeweils wichtigsten Elemente vorgestellt.

Universelle Datentypen und Standardfunktionen

Allgemein können beliebige in C definierte Strukturtypen auch in CUDA verwendet werden – für Vektoren, die auf den Elementartypen für Ganzzahlen und Fließkommazahlen aufbauen, bietet die Laufzeitbibliothek aber bereits vordefinierte Entsprechungen an. Es handelt sich um ein- bis vierdimensionale Vektortypen, deren Bezeichnung einem einheitlichen Schema folgt: Der Name des Basistyps erhält ein Suffix mit der Anzahl der Komponenten als Ziffer; ein optionales Präfix `u` beschreibt den Typ gegebenenfalls als vorzeichenlos. Zum Beispiel entspricht der Typ `uchar2` einem zweidimensionalen Vektor mit Komponenten des Basistyps `unsigned char`, und der Typ `float4` repräsentiert einen vierdimensionalen `float`-Vektor. Auf die Komponenten solcher Vektoren wird unter Angabe der Elementbezeichner `x`, `y`, `z` und `w` zugegriffen. Zur Initialisierung von Variablen dieser Typen können in der Laufzeitbibliothek definierte Funktionen eingesetzt

[1] Im offiziellen Programmierleitfaden finden keine dieser Sprachkonstrukte Erwähnung; angegeben wird darin lediglich die vollständige Unterstützung der auf C zurückzuführenden Teilmenge der Sprache C++ in Device-Code. Im CUDA-Entwickler-SDK von NVIDIA sind jedoch – wenn auch in den meisten Fällen undokumentiert – zahlreiche konkrete Beispiele für den Einsatz der erwähnten Sprachkonstrukte zu finden.

werden: Der Aufruf `make_int2(1, 2)` beispielsweise gibt einen mit den entsprechenden Werten gefüllten zweidimensionalen `int`-Vektor zurück, und die Initialisierungsfunktionen der anderen Strukturtypen folgen diesem Schema analog.

Die Verfügbarkeit der beschriebenen Vektortypen in CUDA ist als ein unverbindliches Angebot zu verstehen – die Strukturen basieren ausdrücklich nicht auf einem von der zugrundeliegenden Technik diktierten Layout, wie es zum Beispiel bei SIMD-Typen der Fall ist, und die skalare Architektur der Hardware bevorzugt nicht eine Organisation der Daten in Vektoren, denn die Art und Weise der Berechnung wird davon nicht beeinflußt. CUDA selbst trägt nur zwei der Vektortypen direkt an den Entwickler heran; darüber hinaus erfordert keine Komponente der Laufzeitumgebung ihre Verwendung, indem beispielsweise API-Funktionen in ihren Parametern solche Typen erwarten oder Ergebnisse in dieser Form zurückgeben würden. Dennoch bietet sich ein Zurückgreifen auf diese vordefinierten Datentypen aus naheliegenden Gründen an: Sie stellen einen gewissen Standard im Rahmen CUDAs dar und bilden damit eine Basis, auf der Quelltexte auch verschiedener Projekte eine einheitliche Struktur und Verständlichkeit aufbauen können.

Die einzigen zwei Vektortypen, die tatsächlich von wichtigen CUDA-Elementen genutzt werden, sind der dem beschriebenen Schema folgende Typ `uint3` sowie die darauf basierende Variante `dim3`, die speziell der Bestimmung von Dimensionen dient. Initialisiert wird eine Variable dieses Typs durch einen Aufruf von `dim3()` mit bis zu drei `int`-Argumenten zur Angabe der jeweiligen Größe in x-, y- und z-Richtung. Fehlende Argumente entsprechen hier dem Wert 1. Verwendet wird `dim3` in der Kernel-Konfiguration zur Angabe der Grid- und Thread-Block-Parameter und innerhalb eines Kernels von den Variablen `gridDim` und `blockDim`. Die ebenfalls nur in Device-Code anwendbaren Variablen `blockIdx` und `threadIdx` basieren schließlich auf dem Vektortyp `uint3`.

```
1  int4 arg = make_int4(1, 2, 3, 4);
2
3  dim3 blockDim(16, 16);  // 256 Threads pro Block
4  dim3 gridDim(32, 32);   // 1.024 Blocks im Grid
5
6  // Größe von 256 int-Werten im Speicher
7  int sharedMemSize = blockDim.x * blockDim.y * sizeof(int);
8
9  // Kernel-Aufruf mit dynamischer Verwaltung des Shared Memorys
10 kernel <<< gridDim, blockDim, sharedMemSize >>> (arg);
```

Codebeispiel 2.3: Kernel-Aufruf mit Bestimmung der Konfiguration. Die hier definierten Variablen `blockDim` und `gridDim` sind nicht zu verwechseln mit den gleichnamigen vordefinierten Variablen, die allein in Device-Code Gültigkeit besitzen.

Das Codebeispiel 2.3 demonstriert die Verwendung der CUDA-Typen sowie der Syntax zur Konfiguration und zum Aufruf eines Kernels.

Von den weiteren Elementen, welche die CUDA-Laufzeitbibliothek zur universellen Verwendung in Host- und Device-Code anbietet, sei schließlich noch die Sammlung mathematischer Funktionen erwähnt, welche in Syntax und Semantik ihren Pendants in den Standardbibliotheken von C und C++ gleichen. Ihre Bereitstellung erlaubt den Einatz eines einheitlichen Funktionssatzes unabhängig davon, ob eine entsprechende Anweisung zur Ausführung auf dem Host oder einem Device vorgesehen ist. In Host-Code finden diese CUDA-Varianten nur dann tatsächlich Anwendung, wenn die Funktionen nicht anderweitig zur Verfügung gestellt werden.

Die Runtime-API

Von den zahlreichen Elementen der Runtime-API können in diesem Abschnitt aus Gründen des Umfangs nur solche vorgestellt werden, die von essentieller Bedeutung sind für die Implementation, welche die vorliegende Arbeit begleitet. Dabei handelt es sich in der Hauptsache um Funktionen, die der Verwaltung des Device-Memorys dienen. Ausgespart werden müssen hingegen solche Funktionalitäten, die keine Anwendung in der Implementation finden, aber auch jene, die nicht den wesentlichen Kern dieser Arbeit betreffen. Es sei also ausdrücklich darauf hingewiesen, daß CUDA in Form seiner APIs mit einer weit umfangreicheren Funktionalität aufwartet, als in diesem Abschnitt dargestellt wird.

Vorwegzunehmen ist, daß einige Funktionen der API implizit asynchron arbeiten, was bedeutet, daß der Programmablauf auf dem Host nicht unterbrochen wird, solange die durch den Aufruf ausgelösten Operationen durchgeführt werden, sondern unabhängig von deren Bearbeitung – also asynchron zu ihnen – fortgesetzt wird. Dies gilt zum Beispiel explizit auch für Kernel-Aufrufe. Von solchen API-Funktionen, die statt dessen synchron arbeiten, werden in manchen Fällen zusätzlich asynchrone Varianten angeboten, die durch das Suffix `Async` gekennzeichnet sind.

Als direkte Pendants zu den Routinen `malloc()` und `free()` der C-Standardbibliothek, mit denen Speicherbereiche reserviert und freigegeben werden können, die sich im Kontext CUDAs im Host-Memory befinden, existieren die API-Aufrufe `cudaMalloc()` und `cudaFree()`, welche die jeweilige Funktion in Bezug auf den Device-Memory erfüllen. Eine Zeigervariable, in der die Adresse des so reservierten Speicherbereichs im Device-Memory hinterlegt ist, läßt sich anschließend als Funktionsargument in einem Kernel-Aufruf übergeben; in Device-Code kann anhand eines derartigen Parameters auf den jeweiligen Speicherbereich zugegriffen werden, der in diesem Fall im Global Memory vorliegt. Das Codebeispiel 2.4 zeigt die Verwendung der Befehle.

```
1  int *deviceMemPtr = NULL;
2
3  // reserviere Speicher für 256 int-Werte im Device-Memory
4  int size = 256 * sizeof(int);
5  cudaMalloc((void **)&deviceMemPtr, size);
6
7  // Kernel-Aufruf mit Übergabe des Zeigers in den Device-Memory
8  kernel <<< gridDim, blockDim >>> (deviceMemPtr);
9
10 // gib reservierten Speicher frei
11 cudaFree(deviceMemPtr);
```

Codebeispiel 2.4: `cudaMalloc()` und `cudaFree()`

Auf Speicherbereiche, die im Device-Memory reserviert wurden, kann vom Host aus nicht in derselben Weise zugegriffen werden wie auf solche im Host-Memory; es ist beispielsweise nicht möglich, den `[]`-Operator auf einem Zeiger in den Device-Memory für einen indizierten Zugriff auf die Elemente zu verwenden. CUDA stellt zwar auch Funktionen bereit, mit denen diese Funktionalität nachgebildet werden kann, für das Kopieren ganzer Speicherbereiche stehen jedoch andere Methoden zur Verfügung: Wiederum in Analogie zu der C-Routine `memcpy()` können durch `cudaMemcpy()` Kopiervorgänge zwischen Bereichen innerhalb des Host-Memorys oder innerhalb des Device-Memorys ebenso veranlaßt werden wie Datenübertragungen aus dem Host- in den Device-Memory und umgekehrt. Die gewünschte Richtung des Transfers wird durch einen Parameter vom Typ `enum` bestimmt, wie im Codebeispiel 2.5 einzusehen ist.

```
1  // Transfer Host-Memory → Device-Memory
2  cudaMemcpy(deviceMemPtr, hostMemPtr, size, cudaMemcpyHostToDevice);
3
4  // Transfer Device-Memory → Host-Memory
5  cudaMemcpy(hostMemPtr, deviceMemPtr, size, cudaMemcpyDeviceToHost);
```

Codebeispiel 2.5: `cudaMemcpy()`

Den Transfers von Speicherbereichen zwischen Host- und Device-Memory kommt in CUDA essentielle Bedeutung zu. Umso wichtiger ist es deshalb, im Hinblick auf die Gesamtlaufzeit einer Anwendung kritische Übertragungen schnell durchführen zu können. Grundsätzlich bietet hierzu das auf beiden Seiten eingesetzte DMA[1]-Verfahren bereits gute Voraussetzungen. Da mit den Standardmethoden wie `malloc()` reservierter Speicher jedoch der Kontrolle des Paging-Mechanismus des Betriebssystems untersteht, müssen für solche Übertragungen Vorkehrungen getroffen werden, damit die Host-seitigen Speicherbereiche während des Trans-

[1] »Direct Memory Access«: Zugriffe auf den Speicher werden direkt ohne Umweg über den zuständigen kontrollierenden Prozessor gewährt und ausgeführt. Somit wird der Prozessor entlastet, und der Zugriff kann in der Regel effizienter gestaltet werden.

fers für den direkten Zugriff durch das Device zur Verfügung stehen. Um zu vermeiden, daß die hierzu notwendigen Operationen unmittelbar zu Beginn der Übertragung ausgeführt werden, erlaubt CUDA mit dem Befehl `cudaMallocHost()` eine entsprechende Vorbereitung bereits beim Reservieren des Speicherbereichs im Host-Memory. Ein auf diese Weise bereitgestellter Speicherblock ist page-locked, d.h., das Betriebssystem kann auf ihm kein Paging mehr durchführen. Damit entfallen die beschriebenen notwendigen Vorkehrungen bei anschließenden Datentransfers, was Geschwindigkeitsgewinne mit sich bringt. Zum Freigeben solcher page-locked Speicherbereiche, also auch zu ihrer Rückführung unter die Kontrolle des Paging-Mechanismus, dient der Befehl `cudaFreeHost()`. Anzumerken ist, daß derart reservierter Host-Memory gleichsam der Kontrolle durch das Betriebssystem entzogen wird, also der in Anspruch genommene physikalische Speicher dem System nicht mehr zur Verfügung steht. Das bedeutet, daß die Leistungsfähigkeit des Gesamtsystems eingschränkt wird, wenn auf diese Weise zuviele Ressourcen belegt werden. Weiterhin empfiehlt sich diese Art der Reservierung nur dann, wenn die höheren Latenzzeiten, die mit dem komplexeren Vorgang zur Bereitstellung eines solchen Speicherblocks einhergehen, durch die tatsächlich erzielten Gewinne bei den Transfers amortisiert werden. Dies ist in aller Regel erst dann der Fall, wenn der Speicherbereich wiederholt Quelle oder Ziel einer Datenübertragung ist. Der Einsatz dieser speziellen Funktion zur Speicherreservierung ist also nur unter gewissen Voraussetzungen sinnvoll, die sorgsam geprüft werden müssen.

Abschließend sei die Möglichkeit erwähnt, im Kontext CUDAs auf gewisse Ressourcen zuzugreifen, die zuvor durch die Funktionen einer Graphik-API im Speicher der Graphikkarte reserviert worden sind. Solche heißen in der hier exklusiv behandelten OpenGL-Variante generell *Buffer-Objects* und dienen dort als Datenfelder, in denen zum Beispiel Eckpunktattribute oder Pixelwerte hinterlegt werden. Um innerhalb eines Kernels Operationen auf diesen Daten ausführen zu können, muß die Ressource zunächst durch einen Aufruf von `cudaGLRegisterBufferObject()` registriert werden. Anschließend läßt sich durch `cudaGLMapBufferObject()` ein im Adreßraum CUDAs gültiger Zeiger auf diesen Speicherbereich im Device-Memory erzeugen; in einem Kernel-Aufruf als Funktionsparameter übergeben, wird hierüber der direkte Zugriff auf die Ressource möglich. Um die Kontrolle über den gemeinsam verwendeten Speicherbereich zeitweilig wieder abzugeben, dient der Befehl `cudaGLUnmapBufferObject()`, und vollständig aufgehoben wird die Berechtigung zum Zugriff auf die Ressource durch einen Aufruf von `cudaGLUnregisterBufferObject()`.

Funktionen zur Verwendung in Device-Code

Die elementare und hier als einzige vorgestellte Funktion der Device-seitigen Komponente der Laufzeitbibliothek ist `__syncthreads()`; sie stellt eine Barriere im Programmablauf dar, die in einem Thread erst überschritten wird, wenn sie in allen anderen Threads desselben Blocks erreicht wurde. Der Aufruf synchronisiert also die Threads eines Blocks und vermag dadurch sicherzustellen, daß sämtliche vorigen Operationen ausgeführt wurden und ihre Effekte somit für alle Threads des Blocks sichtbar sind. Eine solche Garantie ist für das zentrale Element der kooperativen Verarbeitung in einem Block, den Shared Memory, von großer Bedeutung, da bei Zugriffen auf diesen gemeinsamen Speicherbereich klassische Datenabhängigkeiten entstehen können: Die Reihenfolge der Ausführung von Schreib- und Leseoperationen in verschiedenen Threads kann unter Umständen zu jeweils anderen Ergebnissen führen und darf deshalb in solchen Fällen nicht beliebig erfolgen. Sieht ein Kernel zum Beispiel vor, daß in einem Thread ein bestimmter Wert in den Shared Memory geschrieben und in einem anderen Thread anschließend dieser Wert aus dem gemeinsamen Speicherraum ausgelesen wird, ist nicht in jedem Fall garantiert, daß diese Operationen in der bei der Programmierung vorgesehenen Reihenfolge ausgeführt werden. Lediglich für Threads, die in einem Warp gemeinsam berechnet werden, wird durch das SIMT-Verarbeitungsschema die sequentielle Ausführung der Kernel-Anweisungen Thread-übergreifend gewahrt. Gehören also in diesem Beispiel schreibender und lesender Thread nicht demselben Warp an, muß die Datenabhängigkeit aktiv abgesichert werden, damit die Reihenfolge der tatsächlichen Ausführungen der kritischen Operationen nicht weiterhin undefiniert bleibt. In dem beschriebenen Szenario muß deshalb zwischen den Anweisungen zum Beschreiben und Lesen des Shared Memorys eine Synchronisation der beteiligten Threads erzwungen werden.

```
1  __global__ void kernel(void)
2  {
3     __shared__ int s;
4
5     // definiere s nur in Threads mit Index (0, 0)
6     if(threadIdx.x == 0 && threadIdx.y == 0)
7        s = 13;
8
9     int i = s;        // s möglicherweise in manchen Threads undefiniert
10    __syncthreads();  // mache Definition für alle Threads sichtbar
11    int j = s;        // s garantiert in allen Threads konsistent definiert
12 }
```

Codebeispiel 2.6: Shared Memory und `__syncthreads()`

Das Codebeispiel 2.6 zeigt einen Kernel, der nach dem gegebenen Beispiel verfährt, in dem also eine sogenannte echte Datenabhängigkeit vor-

liegt. Ausgegangen wird von einer Ausführungskonfiguration mit zweidimensionalen Blocks, deren Anzahl an Threads jene eines Warps übersteigt. In dem Quelltext wird die Variable `s`, die im Shared Memory liegt, in einem einzigen Thread des Blocks mit einem Wert versehen. In der darauffolgenden Zuweisung in Zeile 9 ist `s` möglicherweise in solchen Threads undefiniert, die einem anderen Warp angehören als jener, in dem `s` initialisiert wird; der Fall tritt genau dann ein, wenn die Zuweisung in diesen Threads vor der nur einmal für den gesamten Block vorgesehenen Initialisierung durchgeführt wird, was möglich ist, weil in CUDA die Reihenfolge der Bearbeitung unterschiedlicher Warps undefiniert ist. Erst der anschließende Aufruf von `__syncthreads()` bewirkt, daß die Zuweisung an `j` in Zeile 11 garantiert in keinem Thread vor der Initialisierung von `s` erfolgt: Zunächst muß wie in jedem anderen auch in demjenigen Thread, in dem die für alle gültige Definition von `s` vorgenommen wird, die Synchronisationsbarriere in Zeile 10 erreicht werden, bevor der Programmablauf für den gesamten Block fortgesetzt und also der Wert von `s` der Variablen `j` zugewiesen wird.

2.3 Entwicklungsstrategien

In den vorangegangenen Kapiteln werden das Konzept CUDAs sowie seine Implementation auf der Hardware beschrieben und die Entwicklungsumgebung mit ihren wesentlichsten Elementen vorgestellt. Damit stehen die Mittel zur Verfügung, um CUDA zur Lösung von Berechnungsproblemen einzusetzen. Allerdings erfordern Komplexität und Flexibilität des Modells über diese grundlegenden Kenntnisse hinaus tiefere Einsicht in die spezielle Arbeitsweise CUDAs, um die Rechenkapazitäten des Graphikprozessors nicht nur miteinbeziehen, sondern in vollem Umfang gewinnbringend ausschöpfen zu können. In diesem Kapitel werden deshalb Aspekte der CUDA-Technik behandelt, deren Beachtung mitunter fundamentale Bedeutung für eine erfolgreiche Entwicklung haben kann. Sie erstrecken sich von wichtigen allgemeinen Kriterien der GPU-Programmierung über deren konkrete Berücksichtigung im Kontext CUDAs bis hin zu spezifischen Optimierungsstrategien.

2.3.1 Arithmetische Dichte und Wahl der Konfiguration

Die parallel organisierte Struktur der GPU zahlt sich in einer Problemlösung dann aus, wenn die vielen Einzelprozessoren gemeinsam und möglichst kontinuierlich zu den hierzu notwendigen Berechnungen herangezogen werden können. In ihr Gegenteil verkehren sich die Vorteile der parallelen Architektur, wenn nur wenige der verfügbaren Einheiten an der Gesamtberechnung teilnehmen können oder die Rechenwerke ihre Arbeit häufig unterbrechen müssen, weil sie beispielsweise auf Ergebnisse von

Speicherzugriffen warten müssen. Für die Bearbeitung auf der GPU – und damit auch für eine Implementierung mit CUDA – eignen sich demnach am besten solche Berechnungsprobleme, die sich in viele parallel ausführbare Kalkulationen aufteilen lassen und die eine hohe arithmetische Dichte aufweisen, worunter das Verhältnis von Rechen- zu Speicherzugriffsoperationen verstanden wird.

CUDA erlaubt durch sein flexibles Ausführungsmodell, die Organisation des Grids innerhalb gewisser Rahmenbedingungen nach eigenem Ermessen zu bestimmen. Bei der Wahl der Konfiguration gilt es, die grundsätzlichen Verarbeitungsmechanismen CUDAs zu beachten, denn eine ungünstige Verteilung der Aufgaben kann dazu führen, daß Ressourcen verschwendet werden.

Sehr naheliegend ist in diesem Zusammenhang zunächst, die Anzahl der Threads in einem Block grundsätzlich an der Warp-Größe zu orientieren, denn die implizite Zerlegung der Blocks zur endgültigen Berechnung gebietet, mit Sorgfalt darauf zu achten, daß die resultierenden Warps in ganzer Breite mit berechnungsrelevanten Threads gefüllt werden können. Da die Verarbeitungsgröße der Warps invariabel ist, entspricht ein nicht voll belegter Warp direkt einer Vergeudung von Ressourcen. Konkret wird aus Gründen der Effizienz bei Registerzugriffen zu Block-Größen geraten, die ein Vielfaches der doppelten Anzahl an Threads in einem Warp darstellen. Auf Basis dieser Einheit ist schließlich ein individueller Kompromiß zu finden: Mehr Threads pro Block werden generell von dem internen Verarbeitungsmechanismus begünstigt, bedeuten jedoch gleichzeitig, daß die konstanten Ressourcen des SMs, also Register und Shared Memory, unter Umständen für weniger aktive Blocks ausreichen. Benötigt bereits ein einzelner Thread-Block mehr Register oder Shared Memory, als ein SM bereitstellen kann, ist keine Kernel-Ausführung mehr möglich.

Die Fähigkeit CUDAs, mit arithmetischen Berechnungen Operationen zu überlagern, die hohe Latenzzeiten mit sich bringen, setzt voraus, daß ein SM soviele aktive Elemente verwaltet, daß sich die Verzögerungen in einem Warp mit der Bearbeitung anderer Thread-Gruppen überbrücken lassen. Es besteht also ein Interesse daran, die Zahl der durch einen SM aktiv verarbeiteten Warps zu maximieren. Das entsprechende Maß heißt *Belegung* und gibt an, zu wieviel Prozent das spezifikationsabhängige Maximum an aktiven Warps ausgeschöpft wird. Je geringer die Belegung für eine Kernel-Ausführung ausfällt, desto weniger effektiv lassen sich also auftretende Latenzzeiten durch Berechnungen verbergen. Allgemein profitieren bandbreitenlimitierte Kernels von einer hohen Belegung, aber auch solche, in denen Threads synchronisiert werden, können die damit verbundenen Wartezeiten dann umso besser überbrücken. Beinhaltet ein Kernel entsprechende Operationen nicht oder nur in unwesentlichem – da in Relation zur Berechnungskomplexität nur geringem – Ausmaß, spielt die Belegung eine untergeordnete Rolle. Insbesondere können in solchen Fällen

Vorkehrungen, die eine höhere Anzahl aktiver Warps auf einem SM als Ziel verfolgen, sogar eine Beeinträchtigung der Leistung zur Folge haben. Es ist also stets von der Charakteristik des Kernels abhängig, ob eine Optimierung bezüglich der Belegung sinnvoll ist.

Vorkehrungen zur Maximierung der Belegung müssen darauf abzielen, mehr aktive Threads auf einem SM zu ermöglichen. Abgesehen davon, daß ein zu lösendes Problem hierzu freilich in hinreichend viele Threads insgesamt zerlegbar sein muß, ist es für einen Kernel in diesem Zusammenhang von Bedeutung, sparsam mit den Ressourcen des SMs umzugehen, so daß sie für entsprechend viele aktive Warps ausreichen. Um beispielsweise zwei Thread-Blocks pro SM aktiv verarbeiten zu können, darf ein solcher Block nicht mehr als die Hälfte des im SM verfügbaren Registerraums und des hierfür abgestellten Shared Memorys in Anspruch nehmen, so daß sich zwei Blocks gleichzeitig diese Ressourcen teilen können. Je nach Beschaffenheit des Kernels kann es sich daher lohnen, Ergebnisse nicht für wiederholte Abfragen zu speichern, sondern bei Bedarf die jeweiligen Berechnungen erneut auszuführen – auch hier zeigt sich wieder die klare Fokussierung eines Graphikchips auf eine hohe Anzahl an Recheneinheiten zuungunsten der Größe des Chipspeichers.

2.3.2 Programmverzweigungen

CUDA erlaubt in einem Kernel beliebige dynamische Programmverzweigungen. Es lassen sich so beispielsweise abhängig von der ID eines Threads Operationen ausführen. Was für die GPU-Programmierung ein sehr mächtiges Mittel darstellt, ist intern allerdings, bedingt durch die SIMT-Verfahrensweise bei der Berechnung, ein diffiziles Problem: Ist ein Programmpfad nicht für alle Threads eines Warps derselbe, muß er dennoch für die gesamte Gruppe beschritten werden, denn die Warp-Struktur ist definiert und kann nicht aufgelöst werden. Dabei werden diejenigen Threads eines Warps, in denen der gerade begangene Ausführungspfad keine Gültigkeit besitzt, für die Dauer seiner Berechnung deaktiviert. Auf diese Weise bleibt der Warp intakt, und die Verarbeitung betrifft nur die hierfür vorgesehenen Threads. Derartige implizite Serialisierungen können die Menge der für einen Warp auszuführenden Operationen in beträchtlichem Maß vergrößern, ohne daß dies der Quelltext des Kernels anschaulich verbildlichen würde. Einen solchen Effekt haben allerdings nur jene Programmverzweigungen, welche divergente Ausführungspfade innerhalb eines Warps verursachen – die unabhängige Berechnung verschiedener Warps läßt für diese auch unterschiedliche Programmabläufe ohne die erläuterten Konsequenzen zu. Da die Verteilung der Threads in Warps deterministisch erfolgt, können in dieser Hinsicht kritische Verzweigungen durchaus identifiziert werden. Zur Optimierung eines Kernels gehört demnach auch eine sorgsame Überprüfung derartiger Konstrukte mit dem Ziel, die Anzahl

der innerhalb eines Warps beschrittenen divergenten Programmpfade insgesamt zu minimieren.

2.3.3 Speicherzugriffe

An vielen Stellen in der vorliegenden Arbeit wird erwähnt, daß Zugriffe auf den Device-Memory mit hohen Latenzzeiten verbunden sind. Für Constant- und Texture-Memory gibt es deshalb Cache-Mechanismen, die bei günstigen Zugriffsmustern die erlaubten Leseoperationen beschleunigen; für den Global Memory aber existieren keine derartigen Verfahren. Dennoch gibt es Möglichkeiten, optimiert auf diesen Speicherbereich zuzugreifen und auf diese Weise Leistungseinbußen zu minimieren.

Operationen auf dem Shared Memory sind im Vergleich weit weniger kritisch: Dieser Speicherraum erlaubt Zugriffe grundsätzlich in derselben Geschwindigkeit wie Register. Jedoch basiert auch der Shared Memory auf einer Implementation, die gewisse Zugriffsschemata begünstigt und andere nur suboptimal auf das zugrundeliegende Konzept abzubilden vermag. Für eine gewissenhafte Programmoptimierung existieren demnach auch für die Nutzung dieses Speicherraums erwähnenswerte Strategien.

Die folgenden Abschnitte gehen auf die Hintergründe der Zugriffe auf Global Memory bzw. Shared Memory ein. Darauf aufbauend wird erläutert, welche Zugriffsmuster das System vor Probleme stellen und welche eine optimale Bearbeitung zulassen.

Global Memory

CUDA ist imstande, in einer Operation entweder 32, 64 oder 128 Bit Daten aus dem Global Memory zu lesen oder dorthin zu schreiben. Das bedeutet, daß zum Beispiel zum Lesen von vier `float`-Werten aus dem Global Memory entweder vier 32-Bit-Leseoperationen veranlaßt oder – was freilich effizienter ist – in einem einzigen 128-Bit-Zugriff alle vier Werte ausgelesen werden. Weiterhin ergibt sich, daß CUDA für kleinere Datentypen wie zum Beispiel den 8 Bit breiten Typ `char` Zugriffe instruiert, die mindestens 32 Bit umfassen, auch wenn effektiv weniger Daten transferiert werden.

Strukturtypen werden in CUDA als Sequenz ihrer Elemente aufgefaßt, d.h., ein Zugriff auf solche Typen im Global Memory veranlaßt automatisch sequentielle Operationen für alle Komponenten des Verbunds. Normalerweise wird also, um das zuvor gegebene Beispiel aufzugreifen, ein vier `float`-Werte umfassender Datentyp in insgesamt vier 32-Bit-Operationen gelesen bzw. geschrieben. Noch augenscheinlicher wird das Problem, zu welchem diese Vorgehensweise führt, wenn ein Strukturtyp mit vier `char`-Elementen angenommen wird: Statt nur einer werden hier vier 32 Bit breite Speicheroperationen ausgeführt, in denen jeweils aber auch nur 8 Bit effektiv übertragen werden. Um solche Datentypen effizienter, d.h. mit weniger

Zugriffen auf den Global Memory, verarbeiten zu können, benötigt CUDA zusätzliche Informationen über ihre Gestalt.

In welcher Paketbreite Daten im Global Memory gelesen bzw. geschrieben werden können, hängt von ihrer Ausrichtung im Speicher ab: Nur wenn ihre Speicheradresse einem Vielfachen der Einheit entspricht, in welcher der Zugriff stattfinden soll, kann CUDA die effizienteren Methoden einsetzen. Damit diese Bedingung erfüllt ist und der Compiler die Zugriffe auch tatsächlich in der vorgesehenen Einheit bestimmt, ist die Definition der jeweiligen Strukturtypen mit einem Attribut zu versehen: Mittels der Angabe von `__align__()` wird CUDA veranlaßt, den Datentyp im Speicher mit der dabei als Argument übergebenen Anzahl an Bytes auszurichten. So können Ausrichtungen mit 4, 8 oder 16 Byte erzwungen werden, was Zugriffe in Paketen von 32, 64 oder 128 Bit erlaubt.[1] Einige der in CUDA vordefinierten Vektortypen sind auf diese Weise bereits für optimalen Speicherzugriff vorbereitet. Beispielsweise wird der `float4`-Typ mit dem Attribut `__align__(16)` definiert, so daß Daten dieses Formats mit einer einzigen Anweisung aus dem Global Memory gelesen bzw. dort gespeichert werden können.[2] In Codebeispiel 2.7 wird eine analoge Definition gezeigt.

```
1 struct __align__(16) Vec
2 {
3    float x, y, z, w;
4 };
```

Codebeispiel 2.7: Ausrichtung im Speicher

Strukturtypen, die Größen aufweisen, die zwischen bzw. über den von CUDA zur Speicherausrichtung angebotenen Werten liegen, lassen sich in aller Regel am effizientesten einsetzen, wenn auch für sie Ausrichtung und Zugriffseinheit so gewählt werden, daß möglichst wenige Lese- oder Schreiboperationen im Zusammenhang mit ihnen notwendig sind. Zum Beispiel kann ein Verbund dreier `float`-Werte mit 16 Byte ausgerichtet werden, um für diesen nur 12 Byte großen Typ bei Zugriffen auf den Global Memory entsprechend eine statt drei Transaktionen zu veranlassen. Als weiteres Beispiel wird für einen Typ, der aus fünf `float`-Werten besteht, empfohlen, daß er ebenfalls mit dem Attribut `__align__(16)` definiert wird, so daß ein Zugriff hierauf in zwei 128-Bit- und nicht in fünf

[1] Undokumentiert, aber in CUDA definiert ist auch eine Ausrichtung mit 2 Byte durch `__align__(2)`. Die Untersuchung der Übersetzung in Assembler führt ans Licht, daß diese Speicherausrichtung tatsächlich Anwendung findet, und in experimentellen Messungen läßt sich die erwartete Wirkung feststellen.

[2] Allgemein sind jene Vektortypen, deren Größe exakt 2, 4, 8 oder 16 Byte beträgt, mit den entsprechenden Attributen versehen; demgegenüber wird zum Beispiel der Typ `float3` nicht als im Speicher ausgerichtet definiert.

32-Bit-Operationen resultiert. Als Nachteil einer solchen nicht die tatsächliche Größe eines Datentyps widerspiegelnden Ausrichtung ist jedoch festzuhalten, daß sich der dann in Anspruch genommene Speicher ebenfalls in der Einheit der angegebenen Ausrichtung bemißt; je nach dem Umfang, in dem ein entsprechend definierter Strukturtyp zum Einsatz kommt, kann sich der Ressourcenbedarf deshalb mitunter deutlich erhöhen.

Über die beschriebene Optimierung der Zugriffe auf einzelne Daten im Global Memory hinaus existiert in CUDA die Möglichkeit, für mehrere Threads eines Warps kooperativ Operation im Global Memory auszuführen. Voraussetzung hierfür ist insbesondere ein spezielles Zugriffsmuster: Darnach wird in Threads einer Warp-Hälfte auf einen zusammenhängenden Speicherbereich im Global Memory derart zugegriffen, daß die relativen Positionen der gelesenen bzw. geschriebenen Elemente im Speicher gerade die Anordnung der beteiligten Threads in der Warp-Hälfte widerspiegeln. In einem solchen Szenario können die Speicherzugriffe zu einem einzigen Transfer zusammengefaßt werden. Dieses als *Coalescing*, d.h. Zusammenfügen oder auch Verschmelzen, bezeichnete Vorgehen befähigt CUDA, die kritischen latenzzeitbehafteten Zugriffe auf den Global Memory um den Faktor einer halben Warp-Größe zu verbreitern und hierdurch deutliche Effizienzsteigerungen zu erfahren.[1] Die weiteren Bedingungen, unter denen Coalescing möglich ist, sind der Beschreibung zur Abbildung 2.3, die ein solches Zugriffsschema illustriert, zu entnehmen.

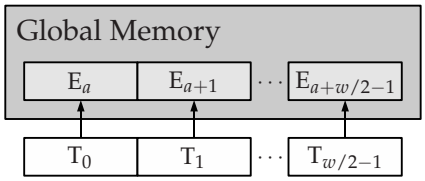


Abbildung 2.3: Coalescing. T_x bezeichnet den Thread mit Kennzahl x in einem Warp; E_a bezeichnet das Datenelement an Speicheradresse a, deren Einheit der Größe des Elements entspricht; w gibt die Größe eines Warps in Threads an. Der Speicherzugriff ist coalesced, wenn $a \mod (w/2) = 0$ und weiterhin die Datenelemente exakt 32, 64 oder 128 Bit groß und entsprechend im Speicher ausgerichtet sind.

Vor dem Hintergrund der erläuterten Fähigkeit CUDAs zu effizienteren Transferzugriffen auf den Global Memory kann es Vorteile mit sich bringen, Datenfelder, für die aufgrund des beabsichtigten Zugriffsschemas Coalescing grundsätzlich in Frage kommt, deren Typ dies aber vereitelt, neu auszulegen: Anders als in dem »Array of Structures« genannten üblichen Layout werden dann die Daten nicht in Form der Strukturtypen als einzelnes Datenfeld, sondern ihre Elemente in separaten Arrays auf Basis ihres

[1] Bis zu 16 separate 32-, 64- oder 128-Bit-Transfers können jeweils zu einem 64-, einem 128- oder zwei 128-Byte-Transfers zusammengefaßt werden.

jeweiligen Typs gespeichert; die Zusammenfassung solcher Datenfelder in einem Verbund nennt sich »Structure of Arrays«. Ein zuvor aufgrund der Größe des Strukturtyps unmögliches Coalescing kann auf diese Weise für die Arrays der einzelnen Elemente dieses Datenverbunds erreicht werden, sofern deren Typ dies zuläßt.

Shared Memory

Die besonders hohe Geschwindigkeit, mit der Zugriffe auf den Shared Memory durchgeführt werden, hat ihre Gründe zum einen in der Implementation dieses Speicherraums im SM-eigenen On-Chip Memory, der wesentlich effizienter als der Device-Memory angebunden ist, zum anderen in der auch hier konsequent verfolgten Parallelität: Eine Aufteilung des Shared Memorys in mehrere Speicherbänke erlaubt es CUDA, Lese- und Schreiboperationen nicht zentral koordinieren und hierfür serialisieren zu müssen, sondern zeitgleich auf den verschiedenen Bänken auszuführen. Nur wenn Zugriffe auf dieselbe Bank veranlaßt werden, müssen diese unter Umständen hintereinander durchgeführt werden, was dann als sogenannter Bankkonflikt eine entsprechende Verzögerung verursacht.

Die Anzahl der Speicherbänke, in welche der Shared Memory aufgeteilt ist, entspricht der halben Größe eines Warps. Das bedeutet, daß derartige Konflikte nur dann auftreten können, wenn auf identische Speicherbänke in Threads derselben Warp-Hälfte zugegriffen wird – solche aus unterschiedlichen Hälften werden infolge des Zeitmultiplexverfahrens nicht gleichzeitig bearbeitet, weshalb in ihnen veranlaßte Operationen auf dem Shared Memory nicht kollidieren können. Der Adreßraum ist auf die verfügbaren Bänke zu 32-Bit-Einheiten in fortlaufender Weise verteilt, d.h., aufeinanderfolgende 32 Bit breite Bereiche werden von jeweils benachbarten Speicherbänken verwaltet.

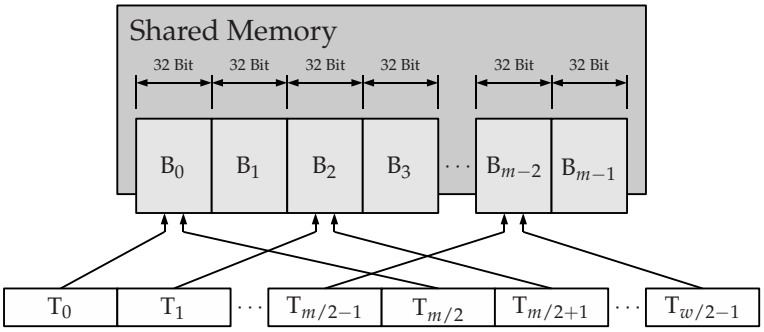


Abbildung 2.4: Bankkonflikte beim Zugriff auf den Shared Memory. In den Threads der Hälfte eines Warps der Größe w wird gemäß ihrer ID in 64 Bit weiten Abständen auf Adressen im Shared Memory zugegriffen. Die $m = \frac{w}{2}$ Speicherbänke werden dabei zum einen Teil gar nicht und zum anderen Teil doppelt angesprochen.

Möglich ist ein konfliktfreier paralleler Zugriff innerhalb einer Warp-Hälfte, wenn in den beteiligten Threads auf Adressen im Shared Memory operiert wird, die von unterschiedlichen Bänken verwaltet werden – sehr einfach kann dies erreicht werden, indem in den Threads jeweils auf benachbarte 32-Bit-Segmente dieses Speicherraums zugegriffen wird. Andererseits können auch leicht Szenarios entworfen werden, die zwangsläufig Bankkonflikte heraufbeschwören: Abbildung 2.4 stellt ein Zugriffsmuster dar, welches zum Beispiel dann zustandekommt, wenn in allen Threads einer Warp-Hälfte Daten des Typs `int2` aufeinanderfolgend in den Shared Memory geschrieben werden – die hier auftretenden 64 Bit weiten Adreßsprünge vereiteln eine gleichzeitige Bearbeitung aller Zugriffe.

Bestehen mehrere gleichzeitige und gleichgerichtete Zugriffe allein aus Leseoperationen, muß daraus jedoch nicht in jedem Fall ein Bankkonflikt entstehen: Die betroffene Bank kann zum sogenannten *Broadcast*, d.h. zum allgemeinen Senden, veranlaßt werden, der es ermöglicht, alle Leseanfragen gleichzeitig zu beantworten. Allerdings ist Broadcasting zur selben Zeit nur für eine Bank möglich, und deren Auswahl geschieht zufällig – in einem Szenario, in dem gleichzeitig auf verschiedene Bänke zugegriffen wird und mindestens eine von ihnen Ziel mehrerer Lesezugriffe ist, wird also nicht automatisch diejenige Bank für den Broadcast vorgesehen, durch deren Auswahl sich die notwendigen Serialisierungen minimieren.

Verzögerungen infolge von Bankkonflikten treten in anderer Weise in Erscheinung als solche, die mit Zugriffen auf den Global Memory einhergehen: Ob durch Coalescing in ihrer Effizienz gesteigert oder nicht, entstehen bei diesen Datenübertragungen Latenzzeiten, denen – wie im Kapitel 2.3.1 auf Seite 30 erläutert – am besten zu begegnen ist, indem sie mit Berechnungen überlagert werden. Auf Bankkonflikte kann nicht in derselben Weise reagiert werden, da Zugriffe auf den Shared Memory prinzipiell keine Latenzzeiten nach sich ziehen und deshalb direkt bearbeitet werden können – wenn auch nicht in jedem Fall gleichzeitig. Die hervorgerufenen Serialisierungen lassen sich demnach nicht durch Berechnungen überbrücken, sondern sie stellen de facto ihrerseits Berechnungsschritte in dem Sinn dar, daß durch sie ein SM mehr Rechenzyklen in die Verarbeitung des betroffenen Warps investieren muß. Wenngleich also selbst mit schwerwiegenden Bankkonflikten verbundene Zugriffe auf den Shared Memory Verzögerungen weit geringerer Ausmaße nach sich ziehen als jede mögliche Operation im Global Memory, spiegelt sich dieses Verhältnis nicht zwangsläufig in den Laufzeiten eines Kernels wider: Können die Zugriffe auf den Global Memory hinreichend gut verborgen werden, sind es möglicherweise Bankkonflikte, die den meßbar größeren Einfluß auf die Leistungsfähigkeit des Kernels haben.

2.4 Einordnung

CUDA läßt sich einerseits verstehen als NVIDIAs bedeutendster Beitrag zur Förderung einer Tradition, die unter der Bezeichnung GPGPU erst den Weg zu dieser Auslegung des Graphikprozessors ebnete, andererseits aber auch als ein neuer, geradezu unerwarteter Ansatz der Programmierung dieser spezialisierten Hardware. Dabei handelt es sich dennoch keineswegs um einander ausschließende Interpretationen, sondern eher um eine konsequente Weiterentwicklung zusammen mit einer innovativen Erweiterung, die gravierenden Einfluß auf die Art der Entwicklung haben kann, aber nicht notwendigerweise haben muß. Im folgenden wird kurz eingegangen auf diese beiden Sichtweisen, nach denen CUDA ein traditionelles GPGPU-Konzept sowohl verkörpert als auch erneuert.

2.4.1 Umsetzung des Stream-Konzepts[1]

Das Modell, in dem die GPU einen allgemeinen Coprozessor darstellt und welches in der GPGPU-Entwicklung vorherrschende Bedeutung erlangt hat, ist das der Stream-Berechnung. Darin wird auf den Elementen eines Stroms von Daten, eines *Streams*, jeweils dieselbe Funktion ausgeführt. Die Einzelergebnisse eines solchen Berechnungsdurchlaufs stellen zusammen wiederum einen oder mehrere Datenströme dar, auf denen in weiteren Funktionen nach demselben Schema operiert werden kann. Eine derartige Funktion heißt *Kernel*. Von einer herkömmlichen Funktion zur Ausführung auf einer CPU unterscheidet sich ein Kernel im wesentlichen dadurch, daß dieser nach einem Aufruf einmal für jedes Element eines Streams, insgesamt also vielfach ausgeführt wird, während jene für jede einzelne Ausführung explizit aufzurufen ist.

Das Stream-Modell stellt eine parallele Berechnung in den Mittelpunkt: Die jeweiligen Ausführungen eines Kernels für die Elemente eines Datenstroms können prinzipiell gleichzeitig erfolgen. Hierzu wird verlangt, daß sie voneinander unabhängig bleiben, d.h., daß die Berechnungen des Kernels für ein Datenelement nicht von jenen für ein anderes Datenelement desselben Streams abhängen. Im Umkehrschluß bedeutet dies, daß während eines parallelen Berechnungsvorgangs auf die Ergebnisse einer einzelnen Kernel-Ausführung nicht aus einer anderen Ausführung heraus zugegriffen werden kann. Diese Bedingungen erlauben eine einfache Umsetzung des Stream-Berechnungsmodells in Hardware: Eine entsprechende Architektur sieht den Verbund hochgradig parallel organisierter Recheneinheiten vor, denen fließend die einzelnen Elemente potentiell riesiger

[1] Das hier erläuterte Verarbeitungsparadigma ist nicht zu verwechseln mit dem gleichnamigen und in dieser Arbeit nicht behandelten Bestandteil der CUDA-Entwicklungsumgebung: Dort wird unter dieser Bezeichnung die Zusammenfassung mehrerer Operationen zu Ausführungssequenzen verstanden.

Datenströme separat zur unabhängigen Verarbeitung zugewiesen werden. Die Erkenntnis, daß die GPU, wenngleich zunächst hochspezialisiert, die Implementation gerade einer solchen Architektur darstellt und das zugehörige Programmiermodell in immer flexiblerer Weise auf diesem Prozessor umgesetzt wird, ist eine tragende Säule der GPGPU-Entwicklung.

CUDA stellt die bislang am weitesten fortgeschrittene Umsetzung des Stream-Programmier- und Ausführungsmodells auf den Graphikprozessoren von NVIDIA dar. In nahezu jeder Hinsicht, in der GPUs früherer Generationen aufgrund ihrer allzu starren Zweckgebundenheit noch wesentliche Restriktionen diesbezüglich aufweisen, ist die neue Architektur hinreichend flexibel und mächtig gestaltet, um weitestgehend uneingeschränkt Berechnungen nach dem Stream-Paradigma durchführen zu können. Wichtige Voraussetzung hierfür sind Fähigkeiten wie die Unterstützung dynamischer Programmverzweigungen, die Ganzzahlarithmetik, das Scattering oder auch eine hohe Anzahl erlaubter Instruktionen in einem Kernel[1]. Damit bietet CUDA eine Basis, auf der viele bereits im GPGPU-Kontext formulierte Ansätze aufbauen und optimiert werden können, ohne hierbei grundlegend neu konzipiert werden zu müssen.

2.4.2 Erweiterung des Stream-Konzepts

Im Stream-Modell generell nicht vorgesehen ist eine Kommunikation zwischen den Berechnungseinheiten während einer Kernel-Ausführung. Auch GPUs verfahren bei zweckgemäßem Einsatz zur Graphikbeschleunigung explizit nicht nach einem solchen Schema. Grund hierfür ist die allgemeine Komplexität, die mit der Implementation der hierbei benötigten Mechanismen einhergeht und die einer möglichst einfachen Realisierung der immer angestrebten Parallelität grundsätzlich im Weg steht. CUDA jedoch erlaubt explizit eine spezielle Kommunikation unter den Berechnungseinheiten auf Basis des Shared Memorys und stellt hierfür die notwendige Synchronisationsfunktion zur Verfügung. Mit dem Paradigma der Stream-Verarbeitung wird dabei nicht gebrochen; es eröffnet sich aber die zusätzliche Möglichkeit, Problemstellungen nach kooperativ und unabhängig behandelbaren Aspekten zu differenzieren. In dieser Hinsicht erweitert CUDA eine strenge Auslegung des Stream-Konzepts.

[1] Ein Kernel darf in CUDA aus bis zu zwei Millionen Assembler-Anweisungen bestehen.

Kapitel 3

Ray-Tracing

Bevor die vorliegende Arbeit im Detail auf den Einsatz von CUDA bei der Implementierung eines GPU-basierten Ray-Tracing-Systems eingeht, wird in diesem Kapitel ein Überblick darüber gegeben, wie bei dieser Art des Renderings verfahren wird und wie sich dies als Berechnungsproblem für die GPU formulieren läßt. Dabei wird auch die Vorgehensweise jenes anderen Verfahrens zur 3D-Graphikberechnung, zu dessen Beschleunigung dieser spezielle Prozessor ursprünglich ausschließlich konzipiert wurde, kurz erläutert, um verstehen zu können, wie und aus welchen Gründen sich seine Architektur zu ihrer heutigen Form entwickelt hat.

3.1 Seitenblick auf das Rasterungsverfahren

Als – gemessen an der Durchdringung des gesamten Hardware- und Software-Markts – weitaus erfolgreichste Vorgehensweise zur Erzeugung dreidimensionaler Computergraphik hat sich die *Rasterung* etabliert. Auf ihr basieren in diesem Zusammenhang all jene Methoden, in denen die aus mathematisch beschriebenen Primitiven zusammengesetzten Objekte der dreidimensionalen Szene durch geeignete Algorithmen auf die Bildebene der betrachtenden Kamera projiziert werden, um anschließend gerastert, d.h. in diskrete Bildelemente oder Fragments aufgeteilt, zu werden. Das sogenannte Sichtbarkeitsproblem, das die Frage aufgreift, welche Objekte im Raum zu sehen sind und welche gegebenenfalls von diesen bei der Betrachtung überdeckt werden, wird erst im Nachhinein zum Beispiel durch die Tiefenpufferung gelöst: Mit Hilfe dieser Technik wird sichergestellt, daß im erzeugten Bild das Fragment eines Primitivs nur dann eingetragen wird, wenn in dem jeweiligen Pixel bisher noch kein Fragment eines dem Betrachterstandpunkt näherliegenden Primitivs gespeichert worden ist. Die Schlüsseleigenschaft dieser Vorgehensweise ist, daß Projektion und Rasterung jedes Primitivs unabhängig von allen anderen erfolgen kann, weil sich das endgültige Bild durch das erläuterte kontrollierte Überschreiben

ergibt. Unabhängigkeit bedeutet in diesem Zusammenhang Parallelisierbarkeit, und derart beschleunigt eignet sich diese Methode hervorragend zur schnellen Berechnung von 3D-Graphik.

Während das Rasterungsverfahren an sich große Geschwindigkeitsgewinne durch die unabhängige und entsprechend parallele Verarbeitung sämtlicher Primitive erzielt, wird auf diese Weise gleichzeitig die Berechnung von Effekten vereitelt, die durch Abhängigkeiten von Objekten untereinander entstehen, also auf die globalen Zusammenhänge in einer Szene zurückzuführen sind. Hierzu gehören der Schattenwurf, Spiegelungen und Transparenzen, da hier jeweils Objekte das Erscheinungsbild anderer Objekte beeinflussen. Um solche zur Wahrung einer realistischen Wirkung unerläßlichen Phänomene dennoch in einem durch Rasterung erzeugten Bild darstellen zu können, müssen meist vergleichsweise komplizierte und mitunter stark von den tatsächlichen physikalischen Zusammenhängen abweichende Methoden eingesetzt werden. Oft entsprechen die so erzielten Ergebnisse deshalb nur bedingt ihren Vorbildern in der Realität und halten solchen Vergleichen auch nur bis zu einem bestimmten Grad stand.

3.2 Die Idee der Strahlverfolgung

Eine andere Vorgehensweise zur digitalen Bildsynthese ist die Strahlverfolgung, *Ray-Tracing* genannt. Ende der 60er Jahre beschrieb Appel dieses Verfahren als prinzipiell geeignete, aber sehr zeitaufwendige Methode zur Lösung des Sichtbarkeitsproblems nicht nur für das Kamerabild, sondern auch für Lichtquellen, so daß sich mit Hilfe dieser Technik ein korrekter Schattenwurf berechnen läßt [App68]. Was heute als klassisches Ray-Tracing bezeichnet wird, geht jedoch auf eine Veröffentlichung von Whitted aus dem Jahr 1980 zurück: Darin werden bisherige Anwendungen der Strahlverfolgung um wesentliche Aspekte der Optik erweitert und ein ganzheitliches Verfahren zur Erzeugung photorealistischer Bilder formuliert [Whi80].[1]

Der Ray-Tracing-Algorithmus basiert auf den Annahmen der Strahlenoptik, einem im Gegensatz zur Wellenoptik oder Quantenoptik auf den makroskopischen Bereich beschränkten Modell. Dabei wird die Ausbreitung von Licht in Form gerade verlaufender Strahlen angenommen. Effekte wie Lichtreflexion und -brechung ergeben sich hier direkt aus den geometrischen Zusammenhängen, unter denen die Strahlen auf spiegelnde bzw. transparente Oberflächen treffen.

Trotz dieser im Vergleich zum Rasterungsverfahren bereits näher an der physikalischen Natur des Lichts befindlichen Grundlage werden beim

[1] Die allein zur Lösung des Sichtbarkeitsproblems eingesetzten Verfahren der Strahlverfolgung werden heute häufig als *Ray-Casting* bezeichnet, um eine deutliche Abgrenzung zum in aller Regel auf Whitteds Variante bezogenen Begriff Ray-Tracing herzustellen.

Ray-Tracing die Lichtstrahlen nicht von der Quelle ihres Ursprungs aus verfolgt, da von allen diesen Strahlen nur ein Bruchteil von unmittelbarer Bedeutung für den Sinneseindruck des Sehens ist und es deshalb einen ungerechtfertigt hohen Aufwand bedeuten würde, tatsächlich das gesamte Licht in seiner allgemeinen Ausbreitung zu simulieren. Vielmehr werden bei dieser Methode allein solche Lichtstrahlen berücksichtigt, die beim Betrachter, der Kamera, aus festgelegten Richtungen einfallen. Aber auch solche Strahlen lassen sich nicht effizient bestimmen, indem sie von einer Lichtquelle ausgehend verfolgt werden, denn dazu müßte ihr Weg bereits vor dessen Berechnung bekannt sein. Kern des Ray-Tracing-Verfahrens ist daher die Erkenntnis, daß die relevanten Strahlen zur Ermittlung des Wegs, auf dem sie schließlich die Bildebene durchstoßen, von dort aus rückwärts verfolgt werden können. Ausgenutzt wird hierbei die Helmholtz-Reziprozität, die besagt, daß der Pfad, dem ein Lichtstrahl folgt, vorwärts wie rückwärts derselbe ist und daß der relative Energieverlust für beide Richtungen identisch ausfällt. Ursprung und Endpunkt eines Lichtstrahls sind für die beabsichtigte Untersuchung also vertauschbar, und da sich die postulierte Relevanz eines Strahls erst bei seiner Ankunft beim Betrachter offenbart, nehmen die Berechnungen im Ray-Tracing ihren Anfang nicht bei den Lichtquellen, sondern bei der Kamera. Anschaulicher läßt sich daher unter dieser Vorgehensweise die Verfolgung von Blick- oder Sehstrahlen verstehen statt von Lichtstrahlen, wenngleich eine solche Differenzierung aufgrund der erwähnten physikalischen Gesetzmäßigkeiten ohne Bedeutung bleibt.

3.3 Der Ray-Tracing-Algorithmus

Der Einsatz des Ray-Tracing-Verfahrens zur Lösung des Sichtbarkeitsproblems sieht vor, vom Standpunkt der Kamera ausgehend durch ihre Bildebene in organisierter Weise Strahlen zu schießen und diese mit prinzipiell allen Objekten der Szene auf Kollisionen hin zu untersuchen. Ein Objekt, das hierbei von einem Strahl zuvorderst getroffen wird, ist gerade jenes, welches im Kamerabild an der Position seines Durchstoßes sichtbar ist; existiert für den Strahl hingegen kein Schnittpunkt, ist an dieser Stelle im Bild kein Objekt, sondern ein als solcher definierter Hintergrund zu sehen. Wird die virtuelle Bildebene gemäß der gewünschten Bildauflösung als diskretes Pixelraster aufgefaßt und durch jedes dieser Pixels ein Sehstrahl in der beschriebenen Weise geschossen und verfolgt, ist das Resultat die Lösung des Sichtbarkeitsproblems für das Kamerabild.

Ihre Farbe erhalten die Pixels des Bildes im klassischen Ray-Tracing durch die Berechnung der Beleuchtungsverhältnisse, die an den für sie gefundenen Schnittpunkten herrschen. Diesem als *Shading* bezeichneten Prozeß liegt ein gewisses Beleuchtungsmodell zugrunde. Nach Whitted wer-

den darin sowohl das direkt von den Lichtquellen an dem betrachteten Ort einfallende als auch das dort indirekt infolge des Lichttransports über spiegelnde und durch transparente Oberflächen hinweg eintreffende Licht berücksichtigt.

Zunächst wird beim Shading geprüft, ob ein gefundener Schnittpunkt das Licht einer gegebenen Lichtquelle – die im folgenden stets als Punktlicht vorausgesetzt wird – empfängt oder sich in ihrem Schatten befindet. Hierzu werden sogenannte *Schattenfühler* eingesetzt: Dieser spezielle Strahltyp zeigt von dem untersuchten Oberflächenpunkt zu der jeweiligen Lichtquelle und wird in diesem Bereich wiederum auf Kollisionen mit den Objekten der Szene hin getestet. Sobald eine solche entdeckt wird, steht fest, daß sich der betrachtete Ort im Schatten dieser Lichtquelle befindet, ihr Licht seine Erscheinung im Kamerabild also nicht beeinflußt;[1] ansonsten ist zu bestimmen, wie dieser direkte Lichteinfall sich auf die Farbe auswirkt, in welcher der Schnittpunkt im Bild erscheint. Häufig wird hierzu das von Phong in [Pho75] vorgestellte Beleuchtungsmodell eingesetzt, das sich aus einem Term zur diffusen Lichtreflexion nach dem Lambertschen Gesetz und einem Term zur Imitation eines Glanzeffekts zusammensetzt:

$$I_d = I_a + \underbrace{k_d \sum_{i=1}^{\#l} (\cos \theta_i)}_{\text{Diffusterm}} + \underbrace{k_s \sum_{i=1}^{\#l} (\cos^n \phi_i)}_{\text{Glanzterm}} \tag{3.1}$$

mit

I_d als von der Oberfläche reflektiertem Anteil des direkt einfallenden Lichts,

I_a als konstantem ambienten Licht,

k_d als vom Material abhängiger Konstante für die diffuse Reflexion mit $0 \leq k_d \leq 1$,

$\#l$ als Anzahl der beleuchtenden Lichtquellen,

θ_i als Winkel zwischen dem Vektor der Normalen an dem betrachteten Oberflächenpunkt und dem Vektor in Richtung der i-ten beleuchtenden Lichtquelle,

k_s als vom Material abhängigem Reflexionsgrad mit $0 \leq k_s < 1$,

ϕ_i als Winkel zwischen dem Vektor in Gegenrichtung des geschnittenen Strahls und dem Vektor in Richtung der Spiegelung des Lichts der i-ten beleuchtenden Lichtquelle an der Oberfläche sowie

[1] Nach diesem Schema werden Objekte mit transparenten Materialeigenschaften wie lichtundurchlässige Objekte behandelt. Als Alternative bietet sich an, eine vom jeweiligen Transmissionsgrad abhängige Abschwächung des Lichteinfalls zu berechnen.

n als vom Material abhängigem Steuerparameter für den Glanzeffekt.

Das über Indirektionen an dem betrachteten Oberflächenpunkt eintreffende Licht wird im klassischen Ray-Tracing-Beleuchtungsmodell durch zwei zusätzliche Terme repräsentiert, die dem nach Formel (3.1) berechneten direkten Licht aufaddiert werden:

$$I = I_d + k_s S + k_t T \tag{3.2}$$

mit

I als von der Oberfläche reflektiertem Anteil des direkt und indirekt einfallenden Lichts,

S als Betrag des Lichts aus Richtung des an der Oberfläche gespiegelten Sehstrahls,

k_t als vom Material abhängigem Transmissionsgrad mit $0 \leq k_t < 1$ sowie

T als Betrag des Lichts aus Richtung des an der Oberfläche gebrochenen Sehstrahls.[1]

Das in Formel (3.2) mit S und T bezeichnete Licht ergibt sich durch Anwendung derselben Technik, mit der das Sichtbarkeitsproblem für das Kamerabild gelöst wird: Neue Strahlen werden erzeugt, die ihren Ursprung in dem betrachteten Oberflächenpunkt haben und deren Richtungen nach den Regeln der Strahlenoptik berechnet werden; ihre Verfolgung gestaltet sich analog zu jener der initialen Sehstrahlen, und die gefundenen Schnittpunkte werden wiederum dem Prozeß des Shadings unterzogen. Beide Terme lassen sich also durch rekursive Aufrufe derselben Ray-Tracing-Prozedur berechnen. Damit beherrscht dieses Verfahren der digitalen Bildsynthese gleichsam inhärent die – im Modell der Strahlenoptik physikalisch korrekte – Simulation des Schattenwurfs, der Reflexion und der Brechung.

Als sinnvoll erweist sich für die weitere Diskussion eine Differenzierung der im Ray-Tracing verfolgten Strahlen in sogenannte *Primärstrahlen*, welche den initialen Sehstrahlen entsprechen, und *Sekundärstrahlen*, als welche Schattenfühler und insbesondere sämtliche durch Reflexion oder Brechung entstandene Strahlen bezeichnet werden.

[1] Oft wird für die Bestimmung der Intensität des Glanzeffekts in dieser Formel nicht mehr der Reflexionsgrad herangezogen, sondern eine separate Materialkonstante definiert, so daß Oberflächen auch ohne darüber hinausgehende spiegelnde Eigenschaften den Eindruck variabler Glätte erwecken können.

3.4 Beschleunigungsstrategien

Die zentrale Operation im Ray-Tracing ist der Schnittpunkttest: Strahlen werden erzeugt, um mit den Primitiven der Szene auf Schnittpunkte hin untersucht zu werden, und es sind die Ergebnisse dieser Suche, die das Shading und die Erstellung weiterer Strahlen bestimmen. Whitted ermittelt in einer Analyse der Programmausführung in [Whi80], daß der Anteil der Gesamtlaufzeit, der zur Berechnung der Schnittpunkte aufgewandt wird, für einfache Szenen auf 75% zu beziffern ist und mit zunehmender Komplexität auf über 95% ansteigt. Tatsächlich besteht ein linearer Zusammenhang zwischen der Anzahl an Primitiven in einer Szene und der Laufzeit der Schnittpunktsuche, wenn für einen Strahl wie beschrieben sämtliche Elemente der Szene auf Kollisionen hin überprüft werden. Damit ist der Fokus aller Strategien, die eine Beschleunigung des Ray-Tracing-Verfahrens zum Ziel haben, klar vorgegeben: Die Berechnung von Schnittpunkten ist so effizient wie möglich zu gestalten und die Anzahl der Kollisionstests ist dem notwendigen Minimum anzunähern.

3.4.1 Bounding Volumes

Ein schon früh formulierter Ansatz zur Vermeidung unnötiger Berechnungen stammt unter anderem von Clark [Cla76]: Die komplexen Primitive einer Szene – zum Beispiel allgemeine Polygone oder Freiformflächen – und sogar ganze Gruppen von Elementen können in Strukturen gekapselt werden, die auf einfacheren mathematischen Beschreibungen beruhen. Beim Ray-Tracing genügt dann ein vergleichsweise trivialer Kollisionstest mit dem umfassenden Volumen, um festzustellen, ob die weitere Schnittpunktsuche unter den beinhalteten Primitiven überhaupt notwendig ist oder von vornherein als fruchtlos feststeht. Gemäß ihrem Verwendungszweck heißen solche kapselnden Elemente allgemein *Bounding Volumes* (BV) oder im speziellen Fall von Quadern oder solchen ähnelnden Formen auch *Bounding Boxes* (BB). Neben Kugeln kommen als BVs besonders häufig Quader zum Einsatz, deren Kanten entlang den Achsen des Weltkoordinatensystems orientiert sind und die auch als *Axis-aligned Bounding Boxes* (AABB) bezeichnet werden.

3.4.2 Beschleunigungsdatenstruktur

Das beschriebene Prinzip der BVs läßt sich bereits als einfache Hierarchie interpretieren: Auf höherer Ebene wird mit vereinfachten Methoden festgestellt, ob die komplexe Suche nach gültigen Schnittpunkten auf der niedrigeren Ebene überhaupt erfolgreich sein kann. Dieser Ansatz kann konsequent fortgeführt werden, um den insgesamt bei der Suche zu betreibenden Aufwand weiter zu reduzieren:

Indem nicht mehr nur unmittelbar die Primitive einer Szene in BVs zusammengefaßt, sondern jene ebenfalls wieder zu Paaren in übergeordneten BVs gekapselt werden und nach diesem Schema verfahren wird, bis schließlich alle auf diese Weise erzeugten Volumina von einem einzigen BV umgeben werden, läßt sich die gesamte Szene hierarchisch strukturieren. Das Ergebnis ist eine *Bounding-Volume-Hierarchie* (BVH), der die Datenstruktur eines binären Suchbaums zugrundeliegt: In der Wurzel wird das die Szene umspannende BV gespeichert, innere Knoten repräsentieren weitere Volumina beinhaltende BVs und solche, welche direkt konkrete Primitive umschließen, fungieren als Blätter.

Zur Ermittlung von Schnittpunkten wird eine derartige Hierarchie nach dem Verfahren der Tiefensuche traversiert: Begonnen wird mit der Untersuchung bei der Wurzel des Baums, und nur wenn ein Strahl das BV eines Knotens trifft, werden die BVs seiner Kinder bzw. die enthaltenen Primitive zu weiteren Tests herangezogen. Damit ermöglicht diese Beschreibung der räumlichen Organisation einer Szene die Suche nach einem gültigen Schnittpunkt in einer mittleren Laufzeit, die nur noch in einem logarithmischen Verhältnis zur Anzahl der in der Szene vorhandenen Primitive steht.

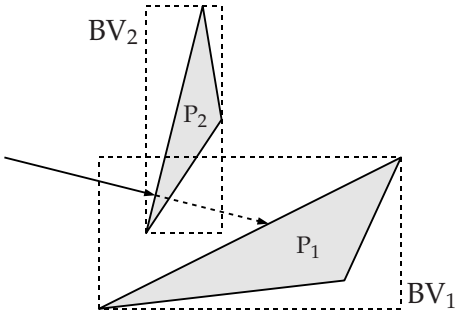
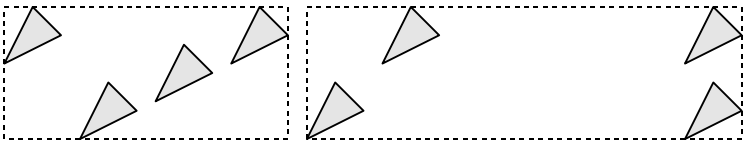


Abbildung 3.1: Szenario der Schnittpunktsuche in einer BVH. Der Strahl trifft zunächst auf BV_1. Wird der damit assoziierte Knoten deshalb zuerst traversiert, ergibt sich ein Schnittpunkt mit dem Primitiv P_1. Die Traversierung darf hier jedoch nicht abgebrochen werden, denn der vorderste Schnittpunkt kommt mit dem Primitiv P_2 zustande, das von dem erst später getroffenen BV_2 umschlossen wird.

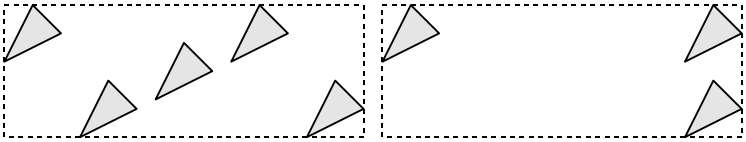
Wichtig ist die Berücksichtigung der Tatsache, daß sich in einer BVH die BVs zweier Kindknoten überschneiden können; wie in Abbildung 3.1 demonstriert, kann deshalb, obwohl im zuerst getroffenen Volumen auch tatsächlich Primitive von dem Strahl durchstoßen werden, der gesuchte vorderste Schnittpunkt unter Umständen von einem Primitiv herrühren, das von dem erst später getroffenen BV umgeben und referenziert wird. Es sind also immer alle Kindknoten, deren BVs geschnitten werden, auch zu traversieren – die Reihenfolge der Kollisionen bestimmt dabei idealerweise, in welchem Knoten die Traversierung zunächst fortgesetzt wird.

Der Einsatz einer BVH im klassischen Ray-Tracing geht auf Rubin und Whitted zurück [RW80]. In ihrer Veröffentlichung wird die Hierarchie manuell erzeugt, was freilich nur für aus heutiger Sicht sehr einfache Szenen eine Option darstellt. Zur automatischen Konstruktion einer BVH wird in vielen Fällen top-down-gerichtet nach einem von Kay und Kajiya in [KK86] beschriebenen rekursiven Algorithmus vorgegangen: Zunächst werden alle Elemente der Szene in einem BV zusammengefaßt; dieses stellt die Wurzel des binären Suchbaums dar. Die eingeschlossene Menge wird anschließend nach räumlichen Gesichtspunkten in zwei Hälften geteilt, für die jeweils separate BVs, die Kinder des Wurzelknotens, erzeugt werden. Rekursiv wird diese Teilung für die Kindknoten fortgesetzt, bis die erzeugten BVs nur noch eine festgesetzte Höchstzahl an Elementen beinhalten; solche werden als Blätter des Baums registriert.

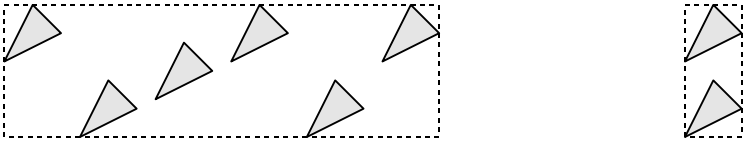
Surface-Area-Heuristik



(a) Teilung im Median der Objektmenge



(b) Teilung in der Mitte des eingenommenen Raums



(c) Optimale Teilung

Abbildung 3.2: Strategien zur Aufteilung einer Objektmenge. Die in (c) gezeigte Teilung ist optimal hinsichtlich der Kriterien, daß der leere Raum in den erzeugten BVs sowie ihre Oberfläche minimiert wird.

Wie bei der Erstellung einer BVH die Menge der Elemente in jedem Schritt aufgeteilt wird, kann die Qualität der resultierenden Hierarchie, also ihre beschleunigende Wirkung im Ray-Tracing, maßgeblich beeinflussen; dies ist direkt aus Abbildung 3.2 ersichtlich. Damit eine BVH ihren Zweck der Beschleunigung möglichst gut erfüllt, ist es demnach erforderlich, ihrer Konstruktion eine adäquate Strategie zugrundezulegen. Ein etablierter Ansatz hierzu ist die *Surface-Area-Heuristik* (SAH) von Goldsmith

und Salmon [GS87]: Darin wird die Oberfläche eines erstellten BVs zum zentralen Kriterium erhoben, auf Basis dessen sich gewisse Kosten modellieren lassen; sie drücken aus, wie hoch der Aufwand einzuschätzen ist, den eine konkrete Strukturierung der Szene bei der Schnittpunktsuche im Ray-Tracing verursacht. Je nach zugrundegelegtem Modell können anhand solcher Kosten bei der Erzeugung einer Hierarchie Lösungen bewertet und miteinander verglichen werden. In dem top-down-gerichteten Konstruktionsprozeß einer BVH lassen sich damit die Fragen beantworten, wo die Teilung einer Menge von Elementen ein optimales Ergebnis liefert und ob durch diese weitere Unterteilung im späteren Ray-Tracing-Prozeß überhaupt noch eine Beschleunigung gegenüber einer direkten Zusammenfassung all dieser Elemente in einem BV zu erwarten ist. Auf diese Weise kann für Szenarios wie jenes, das in Abbildung 3.2 skizziert ist, automatisch die beste – oder eine dem Optimum angenäherte – Lösung gefunden werden.

Konstruktionsverfahren, in denen eine SAH zur Anwendung kommt, sind aus naheliegenden Gründen aufwendiger als solche, in denen nach statischen Kriterien Entscheidungen getroffen werden. Damit müssen aber nicht mehr zwangsläufig abschreckend hohe Laufzeiten einhergehen, was nur zum Teil der obligatorischen Zunahme allgemeiner Rechenleistung geschuldet ist: Insbesondere aus jüngerer Vergangenheit stammen zahlreiche Veröffentlichungen, in denen durch effiziente Techniken und geschickte Approximationen die zur Erstellung SAH-basierter Hierarchien erforderliche Zeit beträchtlich reduziert wird [HMS06, GPSS07, Wal07]. Häufig amortisiert sich der durch die Auswertung des Kostenmodells einhergehende Mehraufwand bereits nach wenigen Ray-Tracing-Durchläufen angesichts der Beschleunigung, welche das Bildsyntheseverfahren durch die Optimierung der Hierarchie erfährt.

3.5 GPU-basiertes Ray-Tracing

Die 3D-Graphikberechnung nach dem Rasterungsverfahren läßt sich als gewisse Folge von Abläufen formulieren, die sich von der unabhängigen Transformation der Primitive über die Ermittlung der endgültigen Pixelfarbwerte bis hin zu ihrer Darstellung erstreckt. Diese Verarbeitungskette wird allgemein Graphik-Pipeline genannt. Alle darin zusammengefaßten Prozesse folgen einem relativ trivialen Schema und eignen sich deswegen gut dazu, auf einer hierauf spezialisierten Hardware ausgeführt zu werden. Da sich die Primitive im Rasterungsverfahren voneinander unabhängig verarbeiten lassen, liegt es nahe, die Hardware in diesem Sinn parallel auszulegen, so daß mehrere Primitive gleichzeitig die Graphik-Pipeline durchlaufen und nach der Rasterung die Farbwerte ganzer Pixelblöcke auf einmal bestimmt werden können. Auch die Architekturen moderner GPUs basieren im Kern noch immer auf diesen Überlegungen und Erkenntnissen.

Ray-Tracing läßt sich nicht auf dieselbe Art wie das Rasterungsverfahren parallelisieren, denn hier können nicht die Objekte der Szene isoliert betrachtet werden, um für sie dann in den jeweils gültigen Pixels einmalig Eintragungen vorzunehmen. Vielmehr beruht das Ray-Tracing auf einer Verarbeitungskette, die für jedes Pixel separat durchlaufen wird, und es sind die damit korrespondierenden Strahlen, die bei dieser Methode unabhängig voneinander betrachtet werden können und deren Verfolgung deshalb parallel erfolgen kann. Das Schema der Graphik-Pipeline, nach dem bei der Rasterung vorgegangen wird und auf dem die Architektur des Graphikprozessors beruht, ist im Ray-Tracing daher nicht anwendbar. Dennoch kann die GPU auch für dieses Verfahren zur Bildsynthese eingesetzt werden, indem sie als allgemeiner Stream-Prozessor aufgefaßt und das Ray-Tracing dementsprechend als Stream-Programm formuliert wird. Im Jahr 2002 stellten Purcell et al. eine solche Lösung in einer vielbeachteten Veröffentlichung vor [PBMH02]. Dabei erforderte das zu jener Zeit verhältnismäßig stark eingeschränkte Programmier- und Ausführungsmodell der GPU spezielle Maßnahmen, um zum Beispiel Limitierungen bei der Anzahl an Instruktionen umgehen oder auf dynamische Programmverzweigungen verzichten zu können – wie im Kapitel 2.4.1 auf Seite 38 erläutert, gelten diese und viele weitere Einschränkungen heute nicht mehr oder nur noch in stark abgeschwächter Form. Auf die seit je her vorhandenen prinzipbedingten Unterschiede zwischen GPU und CPU muß jedoch weiterhin Rücksicht genommen werden, damit ein Ray-Tracing-System von den besonderen Fähigkeiten der GPU bzw. eines Stream-Prozessors profitieren kann. Im folgenden wird daher trotz ihrer nachhaltigen Bedeutung nicht die spezielle Implementation von Purcell et al. behandelt; vielmehr wird auf die Änderungen und Anpassungen eingegangen, die im Zuge der Formulierung des Ray-Tracing-Algorithmus als Stream-Programm notwendig sind.

3.5.1 Ray-Tracing als Stream-Programm

Um den in Kapitel 3.3 auf Seite 41 vorgestellten klassischen Ray-Tracing-Algorithmus als Berechnungsproblem zu formulieren, das nach dem Schema der Stream-Verarbeitung gelöst werden kann, bedarf es zunächst einer Identifikation der wesentlichen Datenströme und anschließend der Überlegung, welche Operationen in einem Kernel auszuführen sind:

Ziel des Ray-Tracing-Vorgangs im hier geschilderten Zusammenhang ist es, ein Bild zu erstellen. Bei distanzierter Betrachtung sind die einzigen gesuchten Daten also die Farbwerte dieses Bildes. Um sie zu erhalten, werden für jedes Pixel dieselben Berechnungen zur Strahlverfolgung und zum Shading durchgeführt, jedoch jeweils unter Verwendung anderer Daten. Tatsächlich ist deren Unterschiedlichkeit aber allein darauf zurückzuführen, daß die Pixels, durch welche die Primärstrahlen geschossen werden,

in Form ihrer Entsprechungen auf der Bildebene der virtuellen Kamera jeweils andere Positionen im Raum einnehmen. Diese Koordinaten des jeweiligen Pixels sind die einzige Information, welche jede Ausführung des Ray-Tracing-Algorithmus individuell benötigt; die in der Folge durchgeführten Berechnungen beziehen zwar viele weitere Daten wie Kameraparameter oder die Primitive der Szene ein, diese sind aber für alle Pixels bzw. Strahlen stets dieselben. Daraus folgt, daß in einem als Stream-Programm formulierten Ray-Tracer die eingehende Menge der Pixelkoordinatenpaare und die ausgehenden endgültigen Farbwerte – das erzeugte Bild – als die elementaren Datenströme zu betrachten sind.

Die Berechnungen, die im Ray-Tracing für ein Pixelkoordinatenpaar durchgeführt werden, lassen sich direkt als die Operationen eines Kernels verstehen: Die wesentlichen Stationen sind die Erzeugung der Primärstrahlen, die Schnittpunktsuche, bei der eine Beschleunigungsdatenstruktur traversiert wird, und das Shading, in dem weitere Strahlen erzeugt werden können, für deren Verarbeitung wieder diese Schritte durchlaufen werden. Es ist jedoch nicht notwendig, all diese Operationen in einem einzigen Kernel zu vereinen; die beschriebenen Prozesse können auch als separate Funktionen formuliert werden. Zwischenergebnisse der einzelnen Verarbeitungsschritte – zum Beispiel die gefundenen Schnittpunkte, für die anschließend das Shading durchzuführen ist – lassen sich dann dem jeweils folgenden Kernel in der Gestalt weiterer Streams übergeben.

Wie die Implementierung eines Stream-Ray-Tracers konkret erfolgt, ob also ein einzelner Kernel alle Operationen umfaßt oder die Berechnungen auf separate Funktionen verteilt werden, hängt von den Umständen ab: Die Lösung Purcells et al. basiert auf mehreren Kernels, weil sich auf diese Weise das Ray-Tracing in Einzelschritten geringerer Komplexität durchführen läßt – die eingeschränkten Fähigkeiten der dort verwendeten GPU-Architektur lassen aus verschiedenen Gründen keine vollständige Bildsynthese in einem einzigen Kernel zu. Neuere Implementationen GPU-basierter Ray-Tracer zeigen, befreit von zwingenden Limitierungen früherer Architekturen, eine Tendenz hin zu dem auf einem einzelnen Kernel basierenden Design [PGSS07, GPSS07]; begründet werden kann die Entscheidung hierfür damit, daß ein solches Ray-Tracing-System in aller Regel bandbreitenschonender zu verwirklichen ist, weil hier auf die zusätzlichen Streams, die zur Kommunikation zwischen den Verarbeitungsschritten erforderlich sind, verzichtet werden kann. Die grundsätzlich erhöhte Komplexität eines Kernels, der alle Operationen des Ray-Tracings in sich vereint, wird demgegenüber in Kauf genommen.

3.5.2 Ray-Tracing als iterativer Prozeß

Ein noch nicht behandeltes zentrales Problem bei der Überführung des klassischen Ray-Tracing-Algorithmus in ein auf einer GPU lauffähiges Pro-

gramm ist der Umgang mit den darin verankerten Rekursionen: Die Strahlverfolgung mündet in einem Aufruf der Shading-Routine, die erst terminieren kann, wenn sie die Ergebnisse weiterer Verfolgungs- und Shading-Berechnungen für Strahlen erhalten hat, welche wiederum dieselben Abhängigkeiten aufweisen. Ein solches Schema benötigt einen Stapelspeicher für Funktionsaufrufe, auf den in GPU-Architekturen bislang jedoch verzichtet wird, um die Komplexität der Ausführungseinheiten dadurch nicht zu erhöhen – so sind auch in CUDA keine allgemeinen Rekursionen erlaubt, wie in Kapitel 2.2.2 auf Seite 19 zur Erwähnung kommt. Es ist deshalb notwendig, dieses Aufrufschema für die Ausführung auf der GPU in einen iterativen Prozeß umzuformen. Purcell et al. erreichen dies durch eine Gewichtung der Strahlen: Sie gibt den Anteil an, zu dem das Resultat der Verfolgung eines Strahls in das Gesamtergebnis, also die endgültige Pixelfarbe, eingeht. Wird diese Information beim Shading eines Schnittpunkts einem hierbei neu erzeugten Sekundärstrahl beigefügt, kann auf rekursive Aufrufe verzichtet werden, wie im folgenden gezeigt wird:

Die rekursive Berechnung des gesamten von einem Schnittpunkt reflektierten Lichts geschieht nach Formel (3.2) auf Seite 43. Zur Eliminierung der Rekursion wird zunächst wieder zu dem in Formel (3.1) auf Seite 42 gegebenen Modell zurückgekehrt, in dem ausschließlich der direkte Lichteinfall auf den betrachteten Schnittpunkt zur Berücksichtigung kommt. Dieses wird nun folgendermaßen ergänzt:

$$I_R = I_d \cdot w_R \tag{3.3}$$

mit

I_R als in Gegenrichtung des Strahls R reflektiertem Licht,

I_d als von der Oberfläche reflektiertem Anteil des direkt einfallenden Lichts nach Formel (3.1) auf Seite 42 sowie

w_R als Gewichtung des Strahls R.

Primärstrahlen werden grundsätzlich mit dem Gewicht $w = 1$ initialisiert. Im Shading der für sie gefundenen gültigen Schnittpunkte wird nach der jetzt eingesetzten Formel (3.3) nur noch das direkt eintreffende Licht berücksichtigt. Die fehlenden Anteile, die aus Richtung einer Reflexion oder Brechung des Strahls stammen, also die Terme $k_s S$ und $k_t T$ in Formel (3.2), werden nicht mehr durch rekursive Aufrufe ermittelt, sondern erst nachträglich in weiteren Iterationen dem bisher berechneten Licht aufaddiert. Hierzu erhalten die korrespondierenden Sekundärstrahlen eine Gewichtung, die dem Maß entspricht, in dem das Resultat ihrer Verfolgung in die endgültige Farbe eingeht: $w \cdot k_s$ bzw. $w \cdot k_t$. Die sichtbaren Effekte von Reflexionen und Brechungen ergeben sich auf diese Weise unmittelbar im Shading der Schnittpunkte der hierzu erzeugten Sekundärstrahlen.

Damit ist der rekursive Ray-Tracing-Algorithmus in einen iterativen Prozeß überführt worden: Ein Berechnungsdurchlauf schließt mit dem Shading ab, in dem Farbwerte produziert werden, die den bisherigen Pixelwerten aufaddiert werden, und mit der Erzeugung von Sekundärstrahlen für die gefundenen Schnittpunkte beginnt der Gang durch die Stationen der Verarbeitungskette von neuem. Werden nun im Shading eines Schnittpunkts Sekundärstrahlen zur Ermittlung des Lichteinfalls sowohl aus Richtung der Reflexion als auch aus jener einer Brechung erzeugt, d.h. für ursprünglich einen zwei neue Strahlen erstellt, kann sich die Anzahl der für ein Pixel verfolgten Strahlen mit jeder Indirektion verdoppeln. In einem rekursiven Algorithmus sind für diesen Umstand aufgrund des impliziten Stapelspeichers keine besonderen Vorkehrungen zu treffen; für einen auf Iterationen beruhenden Stream-Ray-Tracer allerdings bedeutet dieser prinzipiell exponentielle Zuwachs an zu verarbeitenden Strahlen ein diffiziles Problem: Je nach Organisation ist dann innerhalb eines Kernels ein Stapelspeicher für die noch zu verfolgenden Sekundärstrahlen einzurichten oder die Speicherung solcher Strahlen in einem entsprechend vergrößerten Stream oder auch mehreren Datenströmen vorzusehen. Alternativ ist auch ein Vorgehen nach dem Prinzip des Path-Tracings möglich, das Geimer in [Gei06] vorschlägt, um dem in diesem Szenario stark ansteigenden Speicherbedarf begegnen zu können.

Kapitel 4

Implementation[1,2]

Bei der die vorliegende Arbeit begleitenden Implementation handelt es sich um ein Ray-Tracing-System, das wesentliche Berechnungsvorgänge der Bilderzeugung unter Verwendung der NVIDIA-CUDA-Technik auf der GPU ausführt. Der Ray-Tracer ist grundsätzlich als Stream-Programm nach den Vorschlägen Purcells et al. ausgelegt, begeht aber gleichzeitig auch neue Wege, die erst durch das spezielle Programmier- und Ausführungsmodell CUDAs eröffnet werden.

Die detaillierte Beschreibung des Systems in diesem Kapitel erfolgt mit direkter Bezugnahme auf die Arbeitsweise CUDAs. Veröffentlichte Quelltextausschnitte basieren auf der tatsächlichen Implementation, enthalten jedoch teilweise Umformulierungen und Bereinigungen, um den Zwecken der Präsentation und der Übersichtlichkeit gerecht zu werden.

4.1 Entwicklungsziele und Fähigkeiten des Systems

Zielsetzung des Implementierungsvorgangs ist die Erstellung eines GPU-basierten Ray-Tracing-Systems mit klassischen Fähigkeiten, d.h. der Möglichkeit zur Darstellung von Schatten, Reflexionen und Brechungen im Stil Whitteds. Vornehmlich angestrebt wird eine hohe Ausführungsgeschwindigkeit, d.h. je nach Szenen- und Darstellungskomplexität interaktive bis echtzeitkonforme Bildwiederholraten.[3] Deshalb wird höchstes Augenmerk

[1] Sämtliche Angaben in diesem und den folgenden Kapiteln beziehen sich auf den Einsatz der CUDA-Entwicklungsumgebung in dem Betriebssystem Linux.

[2] In den Quelltextbeispielen dieses Kapitels kommt zur Repräsentation von Vektoren im dreidimensionalen Raum der in CUDA vordefinierte Typ `float3` zum Einsatz. Die Verwendung eines alternativen Datentyps, der im Gegensatz zu jenem mit 128 Bit im Speicher ausgerichtet ist, wird später in Kapitel 6 diskutiert.

[3] Diese subjektiven Maße werden in der vorliegenden Arbeit gemäß gängigen Standards wie folgt definiert: Als interaktiv gelten Bildwiederholraten von mindestens einem Bild pro Sekunde; für Echtzeitkonformität werden 25 oder mehr Bilder pro Sekunde gefordert.

auf die effiziente Traversierung einer Beschleunigungsdatenstruktur gelegt, was den Darlegungen in Kapitel 3.4 auf Seite 44 zufolge von höchstem Belang für die Laufzeit des Ray-Tracing-Algorithmus ist. Weiterhin vorgesehen ist eine hinreichende Flexibilität, was verstanden wird als die Fähigkeit, praxisrelevante Szenen darstellen zu können. Dabei kommen von vornherein ausschließlich statische Szenen in Betracht.

Um eine umfassende Untersuchung der Fähigkeiten CUDAs und anschließend eine aussagekräftige Bewertung der Architektur vornehmen zu können, steht im Zentrum der Entwicklung die Umsetzung dreier unterschiedlicher Varianten einer Hierarchietraversierung mit jeweils verschiedenen Anforderungsprofilen. Weiterhin wird neben dem Ray-Tracing nach Whitted ein Ray-Casting-Modus verwirklicht, der durch ein stark vereinfachtes Shading ohne den Einsatz von Sekundärstrahlen eine vergleichsweise neutrale Analyse der absoluten Leistungsfähigkeit des Systems hinsichtlich der Strahlverfolgung zuläßt.

Diese Formulierung von Zielen beeinflußt und diktiert verschiedene grundsätzliche Entscheidungsprozesse, von denen die wichtigsten im folgenden vorweggenommen werden:

Als sichtbare Elemente einer Szene werden von dem implementierten System ausschließlich Dreieckpolygone verarbeitet. Die Festlegung auf diese einzige Primitivform erlaubt eine entsprechend vereinfachte Szenenbeschreibung und eine geradlinige Suche nach Schnittpunkten. Überdies hat sich das Dreieckpolygon in vielen Disziplinen der dreidimensionalen Computergraphik gleichsam als Standard etabliert, weil es sich in kompakter Weise repräsentieren läßt und komplexere Formen wie zum Beispiel Freiformflächen jederzeit in – dann freilich verlustbehaftete – Darstellungen von Dreiecksnetzen überführt werden können.

Als Beschleunigungsdatenstruktur wird eine BVH auf Basis von AABBs eingesetzt. Während sich unter anderem in [Hav00] für statische Szenen im CPU-basierten Ray-Tracing der *k*-d-Baum als die Datenstruktur mit dem höchsten Beschleunigungspotential erwiesen hat, fallen solche Vergleiche für das Ray-Tracing auf der GPU traditionell weniger eindeutig aus, da hier weiterhin die effiziente Traversierung der Hierarchie von größerem Einfluß auf die erzielbare Geschwindigkeit ist als die ihr zugrundeliegende Strategie: Insbesondere die bereits auf CUDA basierenden Lösungen mit einem *k*-d-Baum in [PGSS07] und einer BVH in [GPSS07] zeichnen ein ausgeglichenes Bild hinsichtlich der Beschleunigungswirkung beider Datenstrukturen. Bedeutende Vorteile der BVH gegenüber dem *k*-d-Baum sind hingegen ihre einfachere Erstellung – Elemente müssen hierbei nicht geteilt oder vervielfacht werden – und insbesondere ihr geringerer Speicherbedarf, da bei dieser Form der hierarchischen Strukturierung einer Szene aus verschiedenen Gründen weniger Knoten erzeugt werden müssen [GPSS07]. Solange keine dynamischen Vorgänge eine wiederholte Neustrukturierung erforderlich machen und die Erstellung der Beschleunigungsdatenstruktur

als Vorverarbeitungsschritt auf der CPU ausgeführt wird, kann der Vorteil der einfacheren Erzeugung als vernachlässigbar betrachtet werden. Der geringere Speicherbedarf ist jedoch für das GPU-basierte Ray-Tracing insofern von großer Bedeutung, als die Hierarchie im Graphikspeicher hinterlegt werden muß, der im Vergleich zum Arbeitsspeicher eines Hosts nicht erweiterbar und in aller Regel wesentlich kleiner bemessen ist. Guenther et al. beziffern in [GPSS07] die Größe einer BVH auf etwa 25% bis 33% derjenigen eines ähnlich erstellten und vergleichbar leistungsfähigen *k*-d-Baums, was in direkter Folge die Darstellung komplexerer Szenen erlaubt.

Um die in Kapitel 3.5.2 auf Seite 51 erwähnten Umstände zu vermeiden, die eintreten können, wenn eine Materialbeschreibung zugleich reflektierende und lichtbrechende Charakteristika aufweist, schließen sich diese Eigenschaften in dem implementierten Ray-Tracing-System gegenseitig aus. Es wird also – abgesehen von Schattenfühlern – für jeden gefundenen Schnittpunkt nur höchstens ein Sekundärstrahl erzeugt und verfolgt.

4.2 Grundlegender Aufbau

Im folgenden werden das Rahmenprogramm, das auf dem Host ausgeführt wird, und anschließend die wichtigsten Vorgänge bei der Bilderzeugung erläutert sowie einige der zentralen Datentypen vorgestellt. Ausgelassen werden hier zunächst die Aspekte der Erstellung und Traversierung der eingesetzten BVH – eine detaillierte Sicht auf diesen Themenkomplex gibt im Anschluß ein separates Kapitel.

4.2.1 Rahmen

Die elementaren Datenströme des auf dem Stream-Konzept basierenden Ray-Tracing-Algorithmus werden in Kapitel 3.5.1 auf Seite 48 identifiziert: Für eine eingehende Menge von Koordinaten sind die jeweiligen Pixelwerte zu berechnen. In CUDA kann dieses Gerüst einfach und elegant umgesetzt werden: Wie in Kapitel 2.1.1 auf Seite 8 beschrieben, wird die Ausführung eines Kernels durch eine flexibel gestaltbare Konfiguration bestimmt. Sie kann nun analog zur Abbildung 2.1 auf Seite 8 gerade so gewählt werden, daß durch sie der eingehende Stream von Pixelkoordinaten implizit verkörpert wird: Bildet das Grid exakt das zweidimensionale Layout des zu erzeugenden Bildes nach, so wird für jedes Pixel ein Thread erstellt, dessen Index gerade die assoziierten Pixelkoordinaten widerspiegelt.

Größe und Gestalt des Grids werden in CUDA jedoch nicht in den Einheiten von Threads, sondern von Blocks angegeben; deren Layout wiederum findet in der Beschreibung eines allgemeinen Stream-Ray-Tracers keine Entsprechung, sondern wird idealerweise nach den in Kapitel 2.3.1 auf Seite 30 erläuterten Gesichtspunkten gewählt. Mit Festsetzung der Gestalt der

Thread-Blocks und der Bildmaße ergibt sich das Grid als implizites Pixelraster demnach durch Division der Bildbreite bzw. -höhe durch die x- bzw. y-Dimension der Blocks; die resultierende Konfiguration, als Datenstrom von Pixelkoordinaten verstanden, stellt den Ausgangspunkt der Bilderzeugung dar.

Der Stream der Farbwerte wird hingegen explizit als Datenfeld erstellt, damit das Ergebnis der Ray-Tracing-Berechnungen als Bildinformation im Device-Memory vorliegt und in diesem Sinn wahlweise direkt mit Hilfe einer Graphik-API dargestellt oder auch in den Host-Memory zur Weiterverarbeitung transferiert werden kann. Die Größe des Arrays entspricht der Anzahl der Pixels im Bild und damit der Anzahl der Threads im Grid. In der Zuordnung der Farbwerte setzt sich die intuitive Sicht auf das Grid als Pixelraster fort: Dem Index des mit einem Thread assoziierten Farbwerts im eindimensionalen Array entspricht gerade dessen individuelle ID im Grid; ihre Berechnung wird im Codebeispiel 4.1 gezeigt.

```
1  __device__ unsigned int calcThreadId(void)
2  {
3     // Koordinaten des Threads im Grid
4     uint2 globalIdx =
5        make_uint2(
6              blockIdx.x * blockDim.x + threadIdx.x,
7              blockIdx.y * blockDim.y + threadIdx.y
8        );
9
10    // gib individuelle Thread-ID im Grid zurück
11    return globalIdx.x + globalIdx.y * blockDim.x * gridDim.x;
12 }
```

Codebeispiel 4.1: Funktion zur Berechnung der Thread-ID im Grid

Ist eine direkte Darstellung der Bilddaten mit den Funktionen einer Graphik-API vorgesehen, wird die hierfür bereitgestellte Ressource gemäß den Darlegungen in Kapitel 2.2.3 auf Seite 27 eingebunden. Um das erzeugte Bild dagegen in den Host-Memory zu übertragen, ist dort ebenso der hierfür benötigte Speicher zu reservieren wie im Device-Memory. In Anbetracht der Tatsache, daß die Implementation zur Berechnung von Bildern in schneller Wiederholung eingesetzt wird, lohnt es sich dann, den Speicherbereich im Host-Memory als page-locked einzurichten, um – wie in Kapitel 2.2.3 auf Seite 26 erläutert – die Schreibzugriffe darauf so effizient wie möglich zu gestalten.

Das Codebeispiel 4.2 zeigt die auf die wesentlichsten Vorgänge reduzierte Anzeigeschleife, die im implementierten Ray-Tracing-System für seine einfachste Variante, das Ray-Casting, zum Einsatz kommt. In dem darin aufgerufenen Kernel `traceAndShade` werden sämtliche Berechnungen bis hin zur Ermittlung der endgültigen Pixelwerte durchgeführt. Ferner sind in diesem Beispiel die Anweisungen zur Einbindung eines OpenGL-Pixel-

```
cudaGLRegisterBufferObject(pbo);

while(true) {
   // beziehe Zeiger auf das Pixel-Buffer-Object
   cudaGLMapBufferObject((void **)&pixels, pbo);

   // erzeuge Bild durch Ray-Casting
   traceAndShade <<< gridDim, blockDim >>>
      (cam, scene, pixels);

   // gib Kontrolle über das Pixel-Buffer-Object ab
   cudaGLUnmapBufferObject(pbo);

   display(pbo, width, height);
}

cudaGLUnregisterBufferObject(pbo);
```

Codebeispiel 4.2: Anzeigeschleife im Ray-Casting-Modus. Gezeigt wird die Variante mit direkter Darstellung der im Device-Memory vorliegenden Bilddaten.

```
while(true) {
   // Primärstrahlerzeugung und -verfolgung
   tracePrim <<< gridDim, blockDim >>>
      (cam, scene, hits, floatPixels);

   // Shading
   for(unsigned int l = 0; l < numLights; l++)
      shade <<< gridDim, blockDim >>>
         (scene, lights[l], hits, floatPixels);

   for(unsigned int i = 0; i < numIndirections; i++) {
      // Sekundärstrahlerzeugung und -verfolgung
      traceSec <<< gridDim, blockDim >>>
         (scene, hits, floatPixels);

      // Shading
      for(unsigned int l = 0; l < numLights; l++)
         shade <<< gridDim, blockDim >>>
            (scene, lights[l], hits, floatPixels);
   }

   // Formatumwandlung der Pixelwerte
   convertPixels <<< gridDim2, blockDim2 >>>
      (floatPixels, pixels);

   // Bildtransfer: Device-Memory → Host-Memory
   cudaMemcpy(pixels_h, pixels, size, cudaMemcpyDeviceToHost);

   display(pixels_h, width, height);
}
```

Codebeispiel 4.3: Anzeigeschleife im Ray-Tracing-Modus. Gezeigt wird die Variante mit Transfer der Bilddaten in den Host-Memory und anschließender Darstellung.

Buffer-Objects aufgeführt: In Zeile 5 wird die Zeigervariable `pixels` auf denjenigen Speicherbereich im Device-Memory ausgerichtet, dessen Inhalte mit Funktionen der Graphik-API direkt dargestellt werden können. Die Resultate der Ray-Tracing-Berechnungen werden hier eingetragen und anschließend zur Anzeige gebracht.

Wird die Bilderzeugung hingegen im Stil des klassischen Ray-Tracings vollzogen, verteilen sich die Device-seitigen Berechnungensanweisungen, wie aus Codebeispiel 4.3 hervorgeht, auf verschiedene Kernels – allgemeine Argumente für und wider ein solches Design werden bereits in Kapitel 3.5.1 auf Seite 49 skizziert. Das hier implementierte System präsentiert sich in dieser Form nicht nur schonender im Umgang mit Ressourcen, sondern auch durchweg leistungsfähiger, was sich in einem unternommenen Vergleich mit einer Lösung bestätigt, in welcher die gesamte Ray-Tracing-Verarbeitungskette in einem einzigen Kernel ausgeführt wird. Auf Konsequenzen und Möglichkeiten, die sich aus dieser Gestaltung ergeben, sowie auf weitere Details der hier gegebenen Quelltextbeispiele wird im Verlauf der nächsten Kapitel eingegangen.

4.2.2 Szenenlayout

Die Beschreibung einer Szene enthält in erster Linie geometrische Information. In der vorliegenden Implementation ist dies das Array von Dreieckpolygonen, aus denen die Objekte der Szene zusammengesetzt sind. Zu ihrer Repräsentation werden ein Stützpunkt und zwei Kanten verwendet. Die Normalen jedes Eckpunkts eines Dreiecks werden in einem eigenen Strukturtyp zusammengefaßt und nicht gemeinsam mit dem jeweiligen Polygon gespeichert, da sie nur für das Shading bzw. die Erstellung von Sekundärstrahlen und insbesondere nicht für Schnittpunktberechnungen von Bedeutung sind. Dasselbe gilt für die Materialeigenschaften, die ebenfalls Teil der Szenenbeschreibung sind und von jedem Polygon durch einen Index referenziert werden. Die Relevanz dieses Layouts rührt daher, daß Dreieck- und Normalendaten sowie die Indices der zugewiesenen Materialeigenschaften während der Bildberechnung als Arrays im Global Memory vorliegen, auf dem die notwendigen Zugriffe nur mit großen Verzögerungen durchgeführt werden können.[1] Durch Trennung dieser Daten kann sichergestellt werden, daß nur die für den jeweiligen Bearbeitungsschritt tatsächlich notwendigen Informationen geladen werden. Eine darüber hinausgehende lokalisierte Speicherung der einzelnen Komponenten dieser Strukturtypen, d.h. ein »Structure-of-Arrays«-Layout, wie es in Kapitel 2.3.3 auf Seite 34 angesprochen wird, bringt in der Theorie keine Vor-

[1] Die durch Cache-Mechanismen beschleunigt ansprechbaren Speicherräume eignen sich zur Speicherung dieser Daten nicht oder nur bei anderer Repräsentation: Der Constant-Memory ist für allgemeine Szenen zu klein bemessen, der Texture-Memory erlaubt nicht die Speicherung beliebiger Datentypen.

teile, weil das Zugriffsschema beim Laden dieser Szenendaten grundsätzlich kein Coalescing begünstigt: Wenn in den Threads unabhängig voneinander Strahlen verfolgt werden, erfolgen die Lesezugriffe auf die Geometriedaten ebenfalls in untereinander nicht weiter organisierter Weise.

Mit den Funktionen der CUDA-Laufzeitbibliothek werden die Arrays von Dreieckpolygonen, Normalen und Materialindices in den Global Memory kopiert. Die Zeiger auf die jeweiligen Speicherbereiche werden in einem Strukturtyp zusammengefaßt, der außerdem einen Verweis auf die BVH der Szene enthält. In den Codebeispielen 4.2 und 4.3 repräsentiert die Variable `scene` dieses Bündel, das in den Kernel-Aufrufen als Argument übergeben wird, so daß auf die verwiesenen Datenfelder innerhalb der Kernels zugegriffen werden kann. Nicht in diese Szenenbeschreibung aufgenommen werden die Parameter der virtuellen Kamera und der Lichtquellen. Der Grund hierfür liegt darin, daß in der Implementation eine klare Trennung statischer und dynamischer Elemente vollzogen wird: Während die Inhalte der allgemeinen Szenenbeschreibung, d.h. Geometriedaten und Materialeigenschaften, zwischen zwei Bildberechnungen prinzipbedingt als nicht veränderlich vorgesehen sind, ist eine interaktive Beeinflussung der Kamera- und der Lichtquellenparameter hingegen beabsichtigt. Deshalb – und aus weiteren später erläuterten Gründen – werden solche dynamischen Daten als separate Argumente in den Kernel-Aufrufen übergeben, wie ebenfalls aus den Codebeispielen 4.2 und 4.3 hervorgeht. Die konkreten Materialbeschreibungen werden schließlich im Constant-Memory hinterlegt; für diese Daten weist der Speicherbereich in aller Regel eine hinreichende Größe auf, und beim Shading kann von dem hier verfügbaren Cache-Mechanismus profitiert werden.

4.2.3 Kamerabeschreibung und Primärstrahlerzeugung

```
1 struct Camera
2 {
3    float3 pos, topLeft, hStep, vStep;
4 };
```

Codebeispiel 4.4: Strukturtyp zur Repräsentation der Kamera. In `pos` wird der Augpunkt, in `topLeft` der von dort ausgehende Richtungsvektor zur oberen linken Ecke der Bildebene im Raum gespeichert; `hStep` und `vStep` entsprechen den Schritten, die im Raum von einem Pixel zum horizontal daneben bzw. vertikal darunterliegenden Pixel führen.

Mit Hilfe der Beschreibung einer virtuellen Kamera wird die Bildebene definiert. Der für die Berechnungen auf dem Device eingesetzte Strukturtyp `Camera` kapselt, wie im Codebeispiel 4.4 einzusehen, solche Informationen, die eine unmittelbare Bestimmung der Position zulassen, welche das

mit einem Thread assoziierte Pixel auf der Bildebene einnimmt. Sie können in Vorverarbeitungsschritten auf dem Host aus einem beliebig flexiblen allgemeinen Kameramodell gewonnen werden.

Der Vorteil einer im Vorfeld derart auf die wesentlichsten Parameter reduzierten Kamerabeschreibung zeigt sich bei der Erzeugung der Primärstrahlen: Die Richtung vom Augpunkt zu dem von dem Strahl zu durchstoßenden Ort auf der Bildebene kann für jeden Thread individuell in einer einfachen Funktion ermittelt werden, indem der gegebene Vektor zur oberen linken Ecke der Bildebene um die entsprechenden Pixelschritte im Raum versetzt ausgerichtet wird. Das Codebeispiel 4.5 zeigt diese Funktion zur Erzeugung der Primärstrahlen.

```
1  __device__ void initPrimRay(Ray &ray, const Camera &cam)
2  {
3     // Festsetzung des Strahlursprungs auf den Augpunkt
4     ray.origin = cam.pos;
5
6     // Ausrichtung des Strahls auf die Position des Pixels in der Bildebene
7     ray.dir = normalize(
8        cam.topLeft +
9        cam.hStep * (blockIdx.x * blockDim.x + threadIdx.x) +
10       cam.vStep * (blockIdx.y * blockDim.y + threadIdx.y)
11    );
12
13    // Initialisierung des Strahlabschnittsparameters auf ∞
14    ray.t = INF;
15 }
```

Codebeispiel 4.5: Funktion zur Erzeugung der Primärstrahlen. Die vordefinierten Variablen `blockIdx`, `blockDim` und `threadIdx` werden wie in Codebeispiel 4.1 zur Bestimmung des Thread-Index im Grid herangezogen, der gleichzeitig die Koordinaten des assoziierten Pixels repräsentiert.

4.2.4 Schnittpunktberechnung

Der in der hier vorgestellten Implementation eingesetzte Schnittpunkttest zwischen Strahl und Dreieckpolygon basiert auf dem bekannten Algorithmus von Möller und Trumbore [MT97]. Er erlaubt eine kompakte Beschreibung der Geometrie, da er nicht auf vorausberechnete Werte zurückgreift, und kommt damit dem generellen Bedürfnis entgegen, zusätzliche Berechnungen auf der GPU dem Laden von Daten aus dem Global Memory vorzuziehen. Ein unternommener Vergleich mit einem alternativen Algorithmus von Wald [Wal04] weist die Lösung von Möller und Trumbore überdies bei der Übersetzung durch den CUDA-Compiler-Treiber als bemerkenswert deutlich sparsamer in der Beanspruchung von Registern aus.

Für den Schnittpunkttest zwischen Strahl und AABB wird auf den von Williams et al. in [WBMS05] vorgestellten Algorithmus zurückgegriffen.

Um die Ergebnisse darin sich wiederholender Berechnungen vorwegzunehmen, die allein von gewissen Parametern des untersuchten Strahls abhängig sind, wird in jener Veröffentlichung vorgeschlagen, die Resultate in dessen Strukturtyp aufzunehmen. Eine solche Maßnahme muß im GPU-basierten Ray-Tracing sorgfältig auf ihre Auswirkungen hin untersucht werden: Da sich hierdurch die Anzahl der von einem Thread beanspruchten Register grundsätzlich erhöht, kann die Folge eine Beeinträchtigung der parallelen Ausführung der Kernel-Berechnungen sein, wie in Kapitel 2.3.1 auf Seite 31 erläutert wird. Umgekehrt kann sich die Investition zusätzlicher Register auszahlen, wenn hierdurch Kalkulationen in großer Anzahl oder von hoher Komplexität umgangen werden. Diesbezügliche Untersuchung zeigen für die hier vorgestellte Implementation auf, daß die Vorausberechnung einer komponentenweisen Inversen der Strahlrichtung insgesamt Geschwindigkeitsvorteile, die in [WBMS05] ebenfalls empfohlene Vorwegnahme von Vorzeichentests hingegen -verluste mit sich bringt.

4.2.5 Shading

Ist für einen Primärstrahl der vorderste Schnittpunkt mit einem Primitiv ausgemacht worden, wird zur Berechnung des von dort reflektierten Lichts das Shading angestoßen. Da im Ray-Tracing-Modus für diesen Schritt ein separater Kernel vorgesehen ist, müssen hier die von ihm benötigten Informationen zunächst zwischengespeichert und anschließend weitergereicht werden. Zu diesem Zweck werden sie in einen hierfür vorbereiteten Stream geschrieben, dessen Elemente den im Codebeispiel 4.6 angegebenen Aufbau aufweisen. Die Größe dieses Datenstroms entspricht wieder der Anzahl der Pixels des zu erzeugenden Bildes, denn für jeden Primärstrahl existiert entweder ein oder gar kein vorderster Schnittpunkt.

```
1 struct __align__(16) Hit
2 {
3    float3 pos, dir, n;
4    unsigned int matId;
5 };
```

Codebeispiel 4.6: Strukturtyp zur Speicherung von Schnittpunktinformationen. In `pos`, `dir` und `n` werden der Ort des Schnittpunkts, die Richtung des geschnittenen Strahls sowie die Normale an dem getroffenen Oberflächenpunkt eingetragen; `matId` gibt den Index der im Shading zu verwendenden Materialeigenschaften an.

Tatsächlich vereitelt der zur Speicherung der Schnittpunktinformationen herangezogene Strukturtyp ein Coalescing der Operationen auf dem Stream, obwohl das Zugriffsmuster dies grundsätzlich zuließe. Die Entscheidung hierfür und gegen ein »Structure-of-Arrays«-Layout, das in diesem Fall mit den geeigeneten Datentypen eine Bündelung der Speicherzu-

griffe ermöglichen würde, wird später in den Ausführungen des Kapitels 6 begründet.

Einfaches Shading

Das im Ray-Casting-Modus durchgeführte Shading fällt sehr einfach aus: Es wird implizit eine einzelne weißes Licht aussendende Punktlichtquelle im Zentrum der Kamera angenommen, für die das Lambertsche Beleuchtungsmodell ausgewertet wird; der Einsatz von Schattenfühlern erübrigt sich aufgrund der Positionierung der einzigen Lichtquelle, und auf Glanzberechnungen nach Phong wird hier ebenso verzichtet wie auf die Erzeugung von reflektierten oder gebrochenen Sekundärstrahlen. Die in diesem Schritt berechneten Farbwerte können deshalb direkt in ein darstellbares Format umgewandelt und in den entsprechenden Stream der Pixelwerte eingetragen werden.

Vollständiges Shading

Das Shading, das in der Verarbeitungskette des vollständigen Ray-Tracers zum Einatz kommt, bezieht, anders als beim Ray-Casting, prinzipiell beliebig viele Lichtquellen in die Berechnung ein, die frei konfigurierbar hinsichtlich ihrer Position im Raum und der Lichtfarbe sind. Für jede einzelne wird ein Schattenfühler erzeugt und in vereinfachter Weise verfolgt: Nur im Bereich zwischen dem betrachteten Oberflächenpunkt und der Lichtquelle wird nach einer Kollision mit einem Objekt der Szene gesucht, und die für Sehstrahlen zwingende Ermittlung des vordersten Schnittpunkts bleibt ebenso aus wie die Berechnung der zugehörigen baryzentrischen Koordinaten.

Für vom Schnittpunkt aus sichtbare Lichtquellen wird das Phongsche Beleuchtungsmodell ausgewertet. Die jeweiligen Ergebnisse werden nicht direkt in ein darstellbares Format konvertiert, sondern in den im Codebeispiel 4.3 durch das Argument `floatPixels` referenzierten Stream der Farbwerte als Gleitkommazahlen eingetragen: Dem bisherigen Wert an der dem Pixel entsprechenden Position im Datenstrom werden sie gewichtet aufaddiert. Im Kernel `tracePrim` wird der Stream deshalb entsprechend vorbereitet, indem der Farbwert jedes Pixels auf Schwarz und die in der vierten Komponente des verwendeten Datentyps hinterlegte Gewichtung auf 100% initialisiert werden. Tatsächlich wird dieser Faktor hier also nicht mit einem Strahl, sondern mit dem jeweiligen Pixel assoziiert – in diesem Sinn läßt er sich auch als derjenige Anteil interpretieren, zu dem die Farbe des Pixels noch nicht endgültig feststeht.

Naheliegend für den beschriebenen Shading-Vorgang ist, die Daten aller Lichtquellen in einem Vorbereitungsschritt im Device-Memory zu speichern, um dann innerhalb des eingesetzten Kernels in einer Schleife über

jede Quelle zu iterieren und die jeweiligen Berechnungen durchzuführen. In aller Regel warten selbst sehr komplex ausgeleuchtete Szenen mit einer Anzahl an Lichtquellen auf, für deren Speicherung der Constant-Memory hinreichend groß bemessen ist, so daß sich jener Speicherraum aufgrund der für ihn verfügbaren Cache-Mechanismen besonders für diesen Einsatz empfiehlt; durchgeführte Untersuchungen bestätigen, daß die Verwendung des Constant-Memory hier spürbare Geschwindigkeitsvorteile im Vergleich zur Nutzung des Global Memory mit sich bringt.

Tatsächlich aber erweist sich in Zeitnahmen ein anderer Ansatz, der bereits aus Codebeispiel 4.3 hervorgeht, als der effizienteste: Hier wird das Shading für jede Lichtquelle separat durchgeführt, indem derselbe Kernel im Host-Code wiederholt aufgerufen wird und dabei die jeweiligen Informationen in Form eines Arguments übergeben werden. Auf diese Weise werden die Daten der gerade behandelten Lichtquelle im Shared Memory zur Verfügung gestellt, was sehr effiziente Zugriffsmöglichkeiten während des Shadings eröffnet. Generell als Nachteil dieser Variante ist zu werten, daß sich hier der Aufwand eines Kernel-Aufrufs mit der Anzahl der Lichtquellen multipliziert; Messungen dokumentieren jedoch, daß auch bei Einsatz sehr vieler Lichtquellen keine Zeitverluste gegenüber einer Variante, in der alle Daten im Constant-Memory vorgehalten werden, zu verzeichnen sind. Grund hierfür ist, daß die Auslagerung der Schleife über die Lichtquellen aus dem Device- in den Host-Code eine beträchtliche Verringerung der Komplexität des hierzu eingesetzten Kernels zur Folge hat; dies äußert sich auch in einer geringeren Anzahl der von den Threads in Anspruch genommenen Register, woraus sich direkt mehr Freiheiten bei der Gestaltung der Shading-Funktion ergeben. Als weitere positive Eigenschaft dieser Lösung stellt sich heraus, daß sich hier die Effekte einer interaktiven Manipulation der Lichtquellendaten unmittelbar auf die Kernel-Berechnungen auswirken, während in der anderen Variante erst ein expliziter Transfer in den Device-Memory solche Veränderungen im Kernel sichtbar macht.

4.2.6 Erzeugung und Verfolgung der Sekundärstrahlen

Auf Basis der in dem Typ `Hit` eingetragenen Schnittpunktinformationen kann die durch Reflexion oder Brechung verursachte Ablenkung eines Sehstrahls ermittelt werden. Deshalb wird derselbe Stream, der bereits den Kernel der Primärstrahlerzeugung und -verfolgung mit dem Shading-Kernel verbindet, in der Folge auch an jenen zur Sekundärstrahlbehandlung weitergeleitet. Darin wird abhängig von den geometrischen Zusammenhängen eines Schnitts und den assoziierten Materialeigenschaften die neue Richtung des von dort ausgehenden abgelenkten Strahls berechnet.

Ebenfalls als Argument übergeben wird auch diesem Kernel der Zeiger `floatPixels`, der den Stream der Farbwerte im Gleitkommazahlformat repräsentiert. Eingetragen wird hier als Gewichtung, also erneut in der

vierten Komponente des assoziierten Pixelwerts, jener Anteil, zu dem das Resultat des Shadings eines für diesen Sekundärstrahl gefundenen Schnittpunkts in den endgültigen Farbwert eingeht. Er berechnet sich durch Multiplikation der Gewichtung des zuvor unter diesen Pixelkoordinaten verfolgten Strahls, also dem bisher hier gespeicherten Wert, mit dem Reflexions- bzw. Transmissionssgrad, der aus den Materialeigenschaften herrührt.

Die Verfolgung des neu erstellten Strahls erfolgt analog zu jener eines Primärstrahls. Zur Speicherung von Schnittpunktinformationen wird derselbe Stream verwendet: Die bisher darin eingetragenen Daten können überschrieben werden, weil sie bereits im vorigen Shading-Prozeß und der Erzeugung des nun verfolgten Sekundärstrahls ihren Zweck erfüllt haben. Anschließend wird dieser Datenstrom wieder an den Shading-Kernel zur Einfärbung der gefundenen Schnittpunkte weitergereicht.

Die Verarbeitungskette bestehend aus Sekundärstrahlerzeugung, -verfolgung und anschließendem Shading wird wiederholt durchlaufen, sooft dies die beliebig gewählte Anzahl an Indirektionen erfordert. Im Codebeispiel 4.3 werden diese Vorgänge deshalb in einer Schleife ausgeführt, und der diesbezügliche Ablauf des Ray-Tracings kann wie schon der Schritt des Shadings frei und unmittelbar von seiten des Hosts gesteuert werden, ohne daß diese Flexibilität eine erhöhte Komplexität der eingesetzten Kernels nach sich zieht.

Nachdem die für einen Ray-Tracing-Durchlauf bestimmte Anzahl an Indirektionen berechnet worden ist, erfolgt die endgültige Umwandlung der Pixelwerte in ein darstellbares Format. Im Aufruf des hierzu eingesetzten Kernels `convertPixels` wird eine Konfiguration bestimmt, die unabhängig von jener der vorigen Kernel-Aufrufe ist; Grund hierfür ist, daß dieser Umrechnungsvorgang ein sehr einfacher Prozeß ohne inhaltlichen Bezug zu der komplexen Bilderzeugung ist und deshalb Präferenzen für ein anderes Grid- und Block-Layout zeigt.

4.3 Beschleunigungsdatenstruktur

Zur Beschleunigung der Schnittpunktsuche wird eine auf Basis einer SAH erzeugte BVH verwendet. Im wesentlichen verfährt der implementierte Konstruktionsprozeß in der von Wald in [Wal07] vorgeschlagenen Weise. Allerdings finden hier zahlreiche Maßnahmen, durch welche die Zeit zur Erstellung der Hierarchie verkürzt werden kann, aus Gründen der Vereinfachung keine Anwendung – die Entscheidung hierfür ist das Ergebnis des Kompromisses, eine hochwertige BVH in einem modernen und potentiell weiter optimierbaren Verfahren zu konstruieren, jedoch vor dem Hintergrund, daß im Fokus der Entwicklung die Verarbeitung vorerst ausschließlich statischer Szenen steht, keine Priorität bei der Beschleunigung des Prozesses zu setzen.

Vorweggenommen wird an dieser Stelle die grobe Unterscheidung der drei implementierten Traversierungsmethoden, die im Verlauf dieses Kapitels detailliert beschrieben werden: Zwei greifen auf einen Stapelspeicher zurück, der jeweils in unterschiedlicher Weise verwaltet wird; die dritte Variante verzichtet auf eine solche Datenstruktur und ist deshalb auf ein spezielles Layout der BVH angewiesen.

4.3.1 Repräsentation im Device-Memory

Die auf dem Host erstellte BVH besteht aus Knoten, in denen mit Zeigern auf ihre Kinder bzw. die von ihnen umschlossenen Primitive verwiesen wird. Damit die Hierarchie in einem Kernel auf der GPU traversiert werden kann, muß sie wie die übrige Szenenbeschreibung in den Device-Memory übertragen werden. Die von den Zeigern referenzierten Speicherbereiche liegen jedoch freilich im Host-Memory, was bedeutet, daß nach einer einfachen Kopie der BVH in den Device-Memory alle Zeiger auf innerhalb dieses Speicherraums ungültige Adressen verweisen. Eine Spiegelung der Datenstruktur in den Device-Memory ist demzufolge mit dem Aufwand verbunden, Knoten und Primitive einzeln zu kopieren und ihre in dem neuen Speicherraum eingenommenen Adressen in allen Zeigern einzutragen, durch die sie referenziert werden. Ein solches Verfahren ist in CUDA prinzipiell möglich, aber nur vergleichsweise kompliziert umzusetzen. In dieser Implementation wird deshalb anders vorgegangen:

Die im Host-Memory vorliegende Datenstruktur wird in ein einfaches Array überführt, so daß sämtliche Elemente anhand von Indices referenziert werden können – diese geben statt der absoluten Adressen relative Positionen im Speicher an und sind deshalb in Host- und Device-Memory gleichermaßen gültig. Der Vorgang besteht aus mehreren Schritten: Zunächst wird die bestehende Hierarchie auf dem Host in geeigneter Weise traversiert und dabei jeder passierte Knoten mit einem fortlaufenden Index versehen, der seine Position in dem zu erzeugenden Datenfeld widerspiegelt. Anschließend werden in einer erneuten Traversierung in jedem Knoten für die darin referenzierten Elemente deren Indices hinterlegt. Mit Hilfe der jetzt verfügbaren Informationen kann die angestrebte neue Repräsentation der BVH verwirklicht werden: In einem eindimensionalen Array eines hierfür neu eingeführten Strukturtyps werden die relevanten Daten der Knoten an den Positionen ihrer Indices eingetragen.

Bereits bei der Erstellung der Hierarchie auf dem Host wird die Liste der Primitive dergestalt sortiert, daß alle einem Blatt zugeordneten Elemente aufeinanderfolgend im Speicher vorliegen – da in einer BVH ein Primitiv immer nur von einem einzigen Blatt referenziert wird, ist dies konfliktfrei möglich. Zur Angabe, auf welche Elemente verwiesen wird, genügen dann zwei Ganzzahlwerte: der Index desjenigen beinhalteten Primitivs, das im Array zuvorderst gespeichert ist, und die Anzahl der von

dem Blatt insgesamt referenzierten Elemente; die übrigen Indices lassen sich aus diesen Informationen ableiten. Im Codebeispiel 4.7 wird der Strukturtyp vorgestellt, der auf dem Device zur Repräsentation eines Knotens zum Einsatz kommt.

```
1 struct __align__(16) Node
2 {
3    float3 bounds[2];
4    unsigned int numPrim;
5    unsigned int ptr;
6 };
```

Codebeispiel 4.7: Strukturtyp zur Repräsentation eines Knotens. In `bounds` werden die minimalen und maximalen Koordinaten der zugehörigen AABB gespeichert; `numPrim` gibt die Anzahl assoziierter Primitive an und dient damit auch der Differenzierung des Knotentyps; in `ptr` wird für ein Blatt der Index des ersten beinhalteten Primitivs, für einen inneren Knoten der Index des linken Kindknotens bzw. der des als nächstes zu traversierenden Knotens hinterlegt.

Nach welchem Schema die Knoten der Hierarchie für die spätere Organisation in einem Array indiziert werden und wie genau mit der Information in dem Element `ptr` umgegangen wird, ist jeweils abhängig von dem Verfahren, nach dem die BVH schließlich auf der GPU traversiert wird. Den beiden implementierten Traversierungsmethoden, in denen ein Stapelspeicher zum Einsatz kommt, geht bei der Konstruktion der BVH eine Indizierung voraus, die den Kindern eines inneren Knotens jeweils aufeinanderfolgende Kennzahlen zuweist. Auf diese Weise wird erreicht, daß zur Referenzierung beider Kinder ein einzelner Index genügt, nämlich jener des linken Kindknotens – die Position des rechten Kindknotens im Array, durch das die BVH im Device-Memory repräsentiert wird, läßt sich dann einfach ableiten, ohne explizit gespeichert werden zu müssen. Dies dient der kompakten Darstellung eines Knotens, ist darüber hinaus für die Verfahrensweisen dieser Methoden jedoch keine Voraussetzung. Anders verhält es sich bei der dritten implementierten Traversierungsmethode, in der auf einen Stapelspeicher verzichtet wird: Hier ist die spezielle Reihenfolge, in der die Knoten in dem Datenfeld gespeichert werden, essentieller Bestandteil des Verfahrens, weshalb die hierzu notwendigen Vorverarbeitungsschritte detailliert zusammen mit der korrespondierenden Traversierungsstrategie in einem späteren Abschnitt beleuchtet werden.

4.3.2 Individuelle Traversierung mit Stapelspeicher

Direkt dem traditionellen Vorgehensschema entlehnt ist die für jeden Strahl individuelle Traversierung der Hierarchie mit Hilfe eines Stapelspeichers:

Werden die AABBs beider Kinder eines inneren Knotens von einem Strahl durchstoßen, wird ein Verweis auf einen der Kindknoten auf dem Stapelspeicher zur späteren Berücksichtigung abgelegt und mit der Traversierung des anderen Kindknotens fortgefahren.

Die Möglichkeiten zur Umsetzung der Datenstruktur eines Stacks auf einer GPU sind zunächst eingeschränkt: Die den einzelnen Ausführungseinheiten separat zur Verfügung stehenden Speicherbereiche sind in aller Regel zu klein bemessen und lassen darüber hinaus keine dynamische Verwaltung zu. Durch CUDA werden diese Beschränkungen zum Teil aufgehoben: Wie in Kapitel 2.1.3 auf Seite 15 beschrieben wird, existiert in Form des Local Memorys ein für jeden Thread separat eingerichteter Auslagerungsbereich im Device-Memory, dessen Größe zum Zeitpunkt der Programmübersetzung festgelegt wird. Dies entspricht freilich keiner dynamischen Reservierung, erlaubt jedoch die statische Bereitstellung eines hinreichend großen Speicherbereichs, um einen für den Zweck der Hierarchietraversierung genügend Elemente umfassenden Stack implementieren zu können. Seine Größe muß also im voraus fest bestimmt werden und sich dabei an der maximalen Tiefe der erstellten BVH orientieren – da jene aufgrund des SAH-gelenkten Konstruktionsprozesses nicht als balancierter Binärbaum vorliegt und weitere Faktoren wie die unterschiedliche Anzahl an Primitiven in den Blättern die Gestalt der Hierarchie nicht exakt vorhersagbar machen, ist bei der Festlegung grundsätzlich konservativ vorzugehen, um einem Überlauf des Stapelspeichers zuvorzukommen.

Implementiert wird der Stack nach diesen Überlegungen als herkömmliches Array in Kombination mit einem als Zeiger fungierenden Index, der die Position des zuoberst liegenden Elements angibt. Die Push-Operation entspricht der Erhöhung des Zeigers um eins und dem Einfügen des Elements unter dem neuen Index, während die Pop-Operation umgekehrt verfährt und nach dem Zurückgeben des obersten Elements den Zeiger um eins reduziert. Als leer wird der Stapelspeicher erkannt, wenn der Zeiger auf den ungültigen Bereich verweist, der um eins unter dem niedrigsten Index liegt; auf diesen Wert wird der Zeiger auch initialisiert.

Der Rumpf der Funktion zur Traversierung der BVH mit einem im Local Memory eingerichteten Stapelspeicher wird im Codebeispiel 4.8 gezeigt. Angemerkt sei hierzu vorweg, daß innerhalb eines Threads wiederholte Dereferenzierungen von Zeigern in den Device-Memory nicht jedesmal in Transferoperationen auf diesem Speicherbereich resultieren; beim Übersetzungsvorgang durch den CUDA-Compiler-Treiber werden derartige Anweisungen registriert und Ergebnisse gegebenenfalls in Registern vorgehalten, um die kostspieligen Operationen auf ein Minimum zu reduzieren.[1] Allein deshalb ist es der Leistungsfähigkeit der hier gezeigten

[1] Eine Inspektion der Assembler-Übersetzung bestätigt die beschriebene Vorgehensweise des CUDA-Compiler-Treibers.

```
1  // BV der gesamten Szene getroffen?
2  if(!intersect(scene.bvh[0], ray)) return -1;
3
4  Node *stack[STACK_SIZE];
5  int stackTop = -1;
6  Node *node = &scene.bvh[0];
7  int hitPrimId = -1;
8
9  while(true) {
10    // handelt es sich um ein Blatt?
11    if(node->numPrim > 0) {
12       // schneide mit allen Primitiven des Blattes
13       for(unsigned int i = 0; i < node->numPrim; i++)
14          if(intersect(scene.primitives[node->ptr + i], ray))
15             hitPrimId = node->ptr + i;
16
17       if(stackTop < 0) break;          // Stack leer: Abbruch
18       else node = stack[stackTop--];  // lade Zeiger auf nächsten Knoten
19    }
20    // kein Blatt: schneide mit BVs der Kinder
21    else {
22       Node *leftChild = &scene.bvh[node->ptr];
23       float leftMinT, rightMinT;
24       bool left = intersect(*leftChild, ray, leftMinT);
25       bool right = intersect(*(leftChild + 1), ray, rightMinT);
26
27       // BVs beider Kinder getroffen?
28       if(left && right) {
29          // welchen Kindknoten zuerst traversieren?
30          if(rightMinT < leftMinT) {
31             // merke Zeiger auf linken Knoten
32             stack[++stackTop] = leftChild;
33             node = leftChild + 1;
34          } else {
35             // merke Zeiger auf rechten Knoten
36             stack[++stackTop] = leftChild + 1;
37             node = leftChild;
38          }
39       }
40       else if(left)   // BV des linken Kindes getroffen?
41          node = leftChild;
42       else if(right)  // BV des rechten Kindes getroffen?
43          node = leftChild + 1;
44       else            // kein BV getroffen: prüfe Stack
45          if(stackTop < 0) break;
46          else node = stack[stackTop--];
47    }
48 }
49
50 return hitPrimId;
```

Codebeispiel 4.8: Individuelle Traversierung mit Stapelspeicher

Funktion nicht abträglich, wenn für den aktuell traversierten Knoten, wie aus Zeile 6 ersichtlich ist, ein Zeiger auf seinen Speicherort im Device-Memory eingerichtet und im folgenden an verschiedenen Stellen dereferenziert wird. Ein solches Vorgehen bietet meßbare Geschwindigkeitsvorteile, weil sich hierdurch die Komplexität des Kernels reduziert.

In dem Fall, daß sich die BVs beider Kinder eines Knotens mit dem untersuchten Strahl schneiden, wird in demjenigen Kindknoten mit der Traversierung fortgefahren, dessen AABB von dem Strahl zuerst getroffen wird, während der andere zur späteren Überprüfung vorgemerkt wird, indem auf dem Stapelspeicher ein Zeiger auf ihn hinterlegt wird. Durch diese allgemeine Strategie wird versucht, den relevanten vordersten Schnittpunkt so früh wie möglich auszumachen und so die Suche durch geschicktes Lenken zu verkürzen. Im Codebeispiel wird diese Entscheidung in dem Block von Zeile 28 bis 39 getroffen.

Wird nur eine der AABBs der Kindknoten von dem Strahl durchstoßen, muß der Stapelspeicher nicht in Anspruch genommen werden; das betroffene Kind wird direkt als nächster zu traversierender Knoten registriert. Diese Fälle werden im Codebeispiel in den Zeilen 40 bis 43 behandelt.

Bleibt die Traversierung des untersuchten Knotens erfolglos, d.h., der Strahl trifft zwar das BV des Knotens, nicht aber jene seiner Kinder, ist der Stapelspeicher zu prüfen: Befinden sich dort keine Zeiger mehr auf weitere zu berücksichtigende Elemente, kann die Schnittpunktsuche abgebrochen werden. Ansonsten ist mit der Traversierung desjenigen Knotens fortzufahren, der von dem auf dem Stack zuoberst liegenden Zeiger referenziert wird, und der Verweis entsprechend von dem Stapelspeicher zu entfernen. Im Codebeispiel geschieht dies in den Zeilen 44 bis 46. In derselben Weise wird vorgegangen, nachdem die mit einem Blatt assoziierten Primitive auf Schnittpunkte hin untersucht wurden, um den nächsten zu traversierenden Knoten bzw. das Ende der Schnittpunktsuche zu ermitteln. Dies geht im Codebeispiel aus den Zeilen 17 und 18 hervor.

Die hier präsentierte Traversierung unter Einsatz eines Stapelspeichers macht sich eine exklusive Fähigkeit CUDAs zunutze: Der Local Memory eröffnet die Möglichkeit, für jeden Thread einen eigenen Stack zu verwalten, ohne dabei durch die Anzahl der verfügbaren Register limitiert zu sein oder jene wertvolle Ressource auch nur über Gebühr in Anspruch zu nehmen. Der Compiler-Treiber veranlaßt automatisch die Auslagerung des Stacks in diesen langsam angebundenen, aber entsprechend großen Speicherraum. Konzeptionell bleibt die vorgestellte Lösung jedoch vollständig in dem Rahmen des allgemeinen Stream-Berechnungsmodells, das bereits seit langem GPGPU-Entwicklungen zugrundeliegt: Die Traversierung der Hierarchie wird für jeden Strahl in einem eigenen Thread völlig unabhängig von allen anderen durchgeführt. Damit ist dieses Verfahren ein Beispiel für die in Kapitel 2.4.1 auf Seite 38 erwähnte Umsetzung des Stream-Konzepts durch CUDA.

Zur besseren Lesbarkeit wird auf diese Variante der Traversierung im folgenden unter dem Begriff Local-Memory-Stack-Traversierung (LST) Bezug genommen.

4.3.3 Kooperative Traversierung mit Stapelspeicher

Die zweite implementierte Methode der Traversierung mit einem Stapelspeicher setzt in voller Konsequenz auf dem neuen Programmier- und Ausführungsmodell auf, das durch CUDA auf GPUs eingeführt wird. Es sieht die Speicherung und Verwaltung des verwendeten Stacks nicht wie die zuvor beschriebene LST-Variante für jeden Strahl bzw. Thread separat vor; vielmehr wird ein Stapelspeicher im Shared Memory eingerichtet, der von allen Threads eines Blocks gemeinsam genutzt wird – analog wird für diese Vorgehensweise der Begriff Shared-Memory-Stack-Traversierung (SST) eingeführt. Ein solches Verfahren skizzieren Günther et al. in [GPSS07]; die Veröffentlichung geht jedoch nur sehr eingeschränkt auf die Details der Implementation ein, weshalb sich in den folgenden Beschreibungen auch begründete deutliche Abweichungen von den Vorschlägen wiederfinden, die unter Umständen der unterschiedlichen Umsetzung des Verfahrens geschuldet sind.

Die wesentliche Strategie bei der Verwendung eines gemeinsamen Stapelspeichers für alle in einem Block organisierten Threads ist die Traversierung der BVH für ganze Strahlenpakete. Dabei wird der Pfad der Traversierung nicht für jeden Strahl separat gewählt, sondern für alle Strahlen eines Verbunds gemeinsam begangen. Das bedeutet, daß für diejenigen Strahlen, welche das BV eines gewissen Kindknotens nicht schneiden, dieser dennoch traversiert werden muß, sobald mindestens ein Strahl desselben Pakets mit jener BV kollidiert. Dem vermeintlichen Nachteil der höheren Anzahl an redundanten Traversierungsschritten steht die Speichereffizienz des Verfahrens gegenüber: Nicht nur reduziert sich der für alle Stacks zusammen benötigte Speicherplatz; die Operationen auf dieser Datenstruktur können außerdem sehr viel schneller ausgeführt werden, weil sie im Shared Memory statt wie bei der LST im langsam angebundenen Local Memory eingerichtet wird. Da hier alle Threads eines Blocks in ihrer Repräsentation als Strahlen eines Pakets immer auf denselben Daten, d.h. denselben Knoten oder auch denselben Primitiven, operieren, genügt es weiterhin, diese Informationen nur einmal für einen gesamten Block aus dem Device-Memory zu laden; werden sie dann ebenfalls im Shared Memory hinterlegt, können alle Threads des Blocks darauf zugreifen, und redundante langwierige Ladevorgänge aus dem Global Memory werden vermieden.

Anders als bei der LST werden in dieser Variante auf dem Stapelspeicher nicht die Zeiger auf die im Global Memory gespeicherten Knoten abgelegt, sondern die Knoten selbst: Würde in jedem Thread der Zeiger auf einen Knoten im Global Memory dereferenziert, bedeutete dies erneut red-

undante kostspielige Ladezugriffe auf diesen Speicherraum, und die Geschwindigkeitsvorteile, die sich aus der Verwendung des Shared Memorys ergeben könnten, blieben ungenutzt; werden statt dessen die vollständigen Knotendaten in organisierter Weise nach einem nur einmal für den gesamten Block angewiesenen Ladevorgang aus dem Global Memory auf dem Stapelspeicher abgelegt, richten sich die Stack-Zugriffe aller Threads desselben Blocks in der Folge nur noch auf den schnell angebundenen Shared Memory. Aus demselben Grund wird für die bei der Traversierung eines Knotens untersuchten Kinder ebenfalls ein gemeinsamer Speicherbereich vorbereitet, so daß auch diese Daten nur einmal für einen Block aus dem Global Memory geladen und anschließend von allen seinen Threads effizient gelesen werden können.

Ebenfalls im Unterschied zur LST, bei der für den Stapelspeicher im Local Memory vorab eine feste Anzahl an Elementen veranschlagt werden muß, läßt sich hier seine Größe dynamisch zur Laufzeit bestimmen; das Array des Stacks im Shared Memory wird zu diesem Zweck mit dem Schlüsselwort `extern` deklariert. Entsprechend ergänzt bei Einsatz dieser Traversierungsmethode ein zusätzliches Argument die Angabe der Konfiguration in den Kernel-Aufrufen, die im Codebeispiel 4.3 auf Seite 56 gezeigt sind. Dadurch wird die Kapazität des Stapelspeichers festlegt.

```
1  // individuelle Thread-ID im Block
2  unsigned int tId = threadIdx.x + threadIdx.y * blockDim.x;
3
4  __shared__ extern Node stack[];  // gemeinsamer Stack
5  __shared__ int stackTop;         // gemeinsamer Stack-Zeiger
6  __shared__ Node children[2];     // gemeinsamer Speicher für Kindknoten
7
8  // Initialisierung durch ersten Thread im Block
9  if(tId == 0) stackTop = -1;
10
11 // lade Wurzelknoten der BVH: Global Memory → Shared Memory
12 if(tId * sizeof(float) < sizeof(Node)) {
13    float *src = (float *)scene.bvh;
14    float *dest = (float *)children;
15    dest[tId] = src[tId];
16 }
17
18 // mache Initialisierung und Transfer für alle Threads sichtbar
19 __syncthreads();
20
21 Node node = children[0];  // lade Wurzelknoten der BVH aus Shared Memory
22
23 // BV der gesamten Szene getroffen?
24 if(!intersect(node, ray)) ray.active = false;
```

Codebeispiel 4.9: Vorbereitung der kooperativen Traversierung

Auf Basis dieser Überlegungen können die im Codebeispiel 4.9 einzusehenden Vorkehrungen getroffen werden. Aus den Zeilen 12 bis 16 des Quelltextes geht hervor, daß der Wurzelknoten der BVH, dessen BV die ge-

samte Szene umspannt, nicht in Form seines Strukturtyps in einem einzelnen Thread, sondern in 32 Bit breiten Datenpaketen unter Beteiligung mehrerer Threads aus dem Global Memory geladen und im Shared Memory gespeichert wird. Dieser zunächst unnötig kompliziert erscheinenden Vorgehensweise liegt die Absicht zugrunde, die Schreibzugriffe auf den Shared Memory möglichst effizient zu gestalten: Werden hier in den Threads einer Warp-Hälfte in einer Operation gerade 32 Bit in diesen Speicherraum geschrieben, ist unabhängig von der tatsächlichen Größe des einen Knoten repräsentierenden Datentyps sichergestellt, daß bei diesem Vorgang keine Bankkonflikte auftreten. Weiterhin läßt sich dadurch der Transfer in kleineren Einheiten parallel durchführen, anstatt die Daten des Strukturtyps in nur einem Thread sequentiell zu übertragen. Ausdrücklich keine Priorität hat bei dieser Vorgehensweise das dadurch ermöglichte Coalescing beim Lesen der Daten aus dem Global Memory – hierzu könnten auch 64 oder 128 Bit breite Datenpakete veranschlagt werden, die prinzipiell noch effizientere Zugriffe zulassen würden. Warum sich hier die im Quelltext dargestellte Lösung als die leistungsfähigste erweist, findet Erklärung in der späteren Auswertung des Systems in Kapitel 6.

Der initiale Schnittpunkttest mit der AABB der gesamten Szene in Zeile 24 offenbart ein wichtiges Detail des Ansatzes, ganze Pakete von Strahlen gemeinsam zu verarbeiten: Die Traversierung darf für einen einzelnen Strahl an dieser Stelle nicht ohne weiteres abgebrochen werden, selbst wenn er das BV der Szene verfehlt; als Teil eines Strahlenpakets muß auch er dem Traversierungspfad aller anderen Strahlen dieses Verbunds folgen. Eine Überprüfung an dieser Stelle, ob tatsächlich alle Strahlen das BV der gesamten Szene verfehlen und aus diesem Grund die Schnittpunktsuche vorzeitig für das gesamte Strahlenpaket eingestellt werden kann, erweist sich durchgeführten Zeitnahmen zufolge als nicht vorteilhaft, weswegen zur allgemeinen Verringerung der Komplexität darauf verzichtet wird. Wie an dieser Stelle im Quelltext zu sehen, wird jedoch zur späteren Auswertung in einem dafür vorgesehenen Feld im Strukturtyp zur Repräsentation eines Strahls vermerkt, ob die folgenden Berechnungen für diesen Strahl von Bedeutung sind oder er nur passiv dem Pfad der Traversierung folgt.

Handelt es sich bei dem gerade für ein Strahlenpaket traversierten Knoten um ein Blatt, ist es naheliegend, die referenzierten Primitive auf dieselbe Weise kooperativ aus dem Global Memory in den Shared Memory zu transferieren, wie dies zuvor für den Wurzelknoten der BVH veranlaßt wird. Anschließend können alle Schnittpunkttests für das Strahlenpaket auf den Daten im Shared Memory ausgeführt werden. Der Quelltext zu diesem Vorgang ist im Codebeispiel 4.10 angegeben. Bewußt nicht berücksichtigt wird an dieser Stelle ein Coalescing der Lesezugriffe auf den Global Memory – dieses ist hier zwar ebenfalls möglich, würde jedoch einer weiteren Vorkehrung bedürfen, nämlich der Berechnung derjenigen Thread-ID, welche der relativen Position der zu lesenden Einheit im Speicher ent-

```
1  // gemeinsamer Speicher für Primitive des Blattes
2  __shared__ Prim primitives[MAX_PRIMITIVES_PER_BOX];
3
4  // lade Primitive des Blattes: Global Memory → Shared Memory
5  float *src = (float *)&scene.primitives[node.ptr];
6  float *dest = (float *)primitives;
7  for(
8     unsigned int i = 0;
9     (tId + i * BLOCK_SIZE) * sizeof(float) < node.numPrim * sizeof(Prim);
10    i++
11 )
12    dest[tId + i * BLOCK_SIZE] = src[tId + i * BLOCK_SIZE];
13
14 __syncthreads();  // mache Transfer für alle Threads sichtbar
15
16 if(ray.active)
17    for(unsigned int i = 0; i < node.numPrim; i++)
18       if(intersect(primitives[i], ray))
19          hitPrimId = node.ptr + i;
```

Codebeispiel 4.10: Kooperative Schnittpunktsuche in einem Blatt

spricht. Ermittelt werden kann diese Kennzahl durch eine vergleichsweise kostspielige Modulooperation. Weshalb hier auf das die Lesezugriffe potentiell stark beschleunigende Coalescing verzichtet, gleichzeitig aber angestrebt wird, Bankkonflikte beim Zugriff auf den Shared Memory zu vermeiden, wird ebenfalls später in Kapitel 6 thematisiert.

Nachdem sämtliche Primitive eines Blattes mit den nicht als passiv markierten Strahlen eines Pakets auf Kollisionen hin untersucht worden sind, ist zu prüfen, ob auf dem Stapelspeicher weitere Knoten zur Traversierung hinterlegt sind. Ist dies nicht der Fall, kann die Schnittpunktsuche für das gesamte Strahlenpaket abgebrochen werden. Anderenfalls muß das zuoberst auf dem Stack liegende Element in den Threads des Blocks als nächster zu verarbeitender Knoten geladen und anschließend koordiniert von dem Stapelspeicher entfernt werden. Das Codebeispiel 4.11 dokumentiert diese Schritte.

```
1  if(stackTop < 0) break;  // Stack leer: Abbruch
2  else {
3     node = stack[stackTop];  // lade nächsten Knoten
4     ray.active = true;
5
6     __syncthreads();  // warte, bis Knoten in allen Threads geladen wurde
7     if(tId == 0) stackTop--;  // entferne Knoten vom Stack
8     __syncthreads();  // mache Entfernung für alle Threads sichtbar
9  }
```

Codebeispiel 4.11: Pop-Operation auf dem gemeinsamen Stapelspeicher

Die Umsetzung der Pop-Operation für den gemeinsamen Stack gebietet besondere Umsicht bei der Aktualisierung des Zeigers auf das zuoberst

liegende Element: Da dieser Verweis als Teil der Datenstruktur des Stapelspeichers in allen Threads eines Blocks gemeinsam verwendet wird, darf er erst dann neu ausgerichtet werden, wenn in jedem Thread der unter diesem Index vorliegende Knoten erfolgreich von dem Stack geladen worden ist. Deshalb ist in Zeile 6 des Beispielcodes eine Synchronisation der Threads vorgesehen, die genau diese Reihenfolge der Abläufe sicherstellt. Auf dieselbe Weise muß schließlich die Änderung an dem Zeiger für alle Threads sichtbar gemacht werden, bevor in diesen erneut auf den Stapelspeicher zugegriffen wird.

Die Traversierung eines inneren Knotens der BVH beginnt in dieser Variante erneut mit einem Transfer der hierfür benötigten Daten aus dem Global Memory in den Shared Memory. Es handelt sich um die Kinder des aktuell untersuchten Knotens, die in dem bereits in Zeile 6 des Codebeispiels 4.9 reservierten gemeinsamen Speicherbereich für die Kindknoten hinterlegt werden. Hier wird dasselbe Verfahren zur Vermeidung von Bankkonflikten und unnötigen Serialisierungen eingesetzt, das bereits bei der Übertragung des Wurzelknotens sowie der Primitive eines Blattes zur Anwendung kommt – das bedeutet, daß auch bei diesem Vorgang keine Vorkehrungen für ein Coalescing der Lesezugriffe auf den Global Memory getroffen werden. Einzusehen sind die konkreten Anweisungen hierzu in den Zeilen 2 bis 6 des Codebeispiels 4.12.

```
1  // lade Kindknoten: Global Memory → Shared Memory
2  if(tId * sizeof(float) < 2 * sizeof(Node)) {
3     float *src = (float *)&scene.bvh[node.ptr];
4     float *dest = (float *)children;
5     dest[tId] = src[tId];
6  }
7
8  float leftMinT = INF, rightMinT = INF;
9  bool left = false, right = false;
10
11 __shared__ unsigned int hitCode[4];  // gemeinsamer Kollisionscode
12 if(tId < 4) hitCode[tId] = 0;
13
14 // mache Transfer und Initialisierung für alle Threads sichtbar
15 __syncthreads();
16
17 if(ray.active) {
18    left = intersect(children[0], ray, leftMinT);
19    right = intersect(children[1], ray, rightMinT);
20    hitCode[2 * left + right] = 1;  // codiere Kollisionen
21    if(!left && !right) ray.active = false;
22 }
23 __syncthreads();  // mache Kollisionscode für alle Threads sichtbar
```

Codebeispiel 4.12: Kooperative Traversierung mit Stapelspeicher, Teil 1

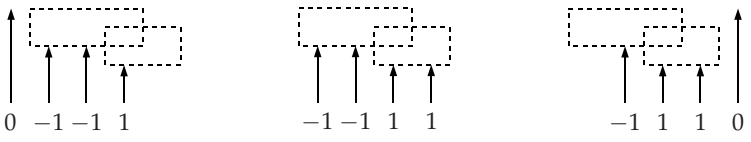
Bei der Traversierung eines Strahlenpakets muß die Information, welche der BVs beider Kinder eines inneren Knotens getroffen werden, kooperativ ausgewertet werden: Auch wenn eine AABB von nur einem Strahl ei-

nes Verbunds getroffen wird, muß der damit assoziierte Knoten für das gesamte Strahlenpaket traversiert werden, und umgekehrt kann ein Knoten nur dann bei der weiteren Traversierung ignoriert werden, wenn sein BV nachweislich von allen Strahlen eines Verbunds verfehlt wird. Aus diesem Grund wird ein Kollisionscode berechnet: In jedem Thread wird gemäß der für den jeweils verarbeiteten Strahl gewonnenen Information eine Position in einem hierfür bereitgestellten gemeinsamen Speicherbereich ermittelt; an ihr wird eine Markierung gesetzt. Im späteren Verlauf der Traversierung werden die Eintragungen in diesem Speicherbereich zentral ausgewertet, um das weitere Vorgehen für das Strahlenpaket zu bestimmen.

Konkret dient zur Codierung der Kollisionen ein Array aus vier Elementen, wie es in Zeile 11 des Codebeispiels 4.12 deklariert wird. Je nachdem, ob ein Strahl mit keinem, einem – und dann mit welchem – oder beiden untersuchten BVs kollidiert, wird in dem Array an einer der vier Position durch jeden beteiligten Thread ein Wert gesetzt. Im Quelltextbeispiel wird dieser Vorgang in Zeile 20 veranlaßt. Ist er für alle Threads abgeschlossen, geht aus den Positionen, unter denen jetzt entsprechende Eintragungen vorzufinden sind, hervor, in welcher Konstellation das Strahlenpaket mit den AABBs der Kindknoten zusammentrifft.

Für den Fall, daß die Auswertung des Kollisionscodes Schnittpunkte mit beiden BVs der Kindknoten ergibt, muß eine Strategie vorliegen, nach der entschieden wird, in welcher Weise die Traversierung fortzusetzen ist. Eine einfache Möglichkeit ist, für solche Fälle eine feste Traversierungsabfolge vorzusehen, also zum Beispiel immer den linken Kindknoten direkt zu traversieren und den rechten auf dem Stapelspeicher für die spätere Berücksichtigung vorzumerken. Ein derart statisches Verfahren wird den unterschiedlichen Konstellationen, in denen es zu Kollisionen mit beiden untersuchten AABBs kommen kann, jedoch freilich nicht gerecht. Vielmehr ist es auch bei der Behandlung ganzer Pakete von Strahlen sinnvoll, die Umstände des Zusammentreffens mit den BVs eingehender zu analysieren.

Günther et al. schlagen in [GPSS07] vor, die Traversierung in demjenigen Kindknoten fortzusetzen, dessen BV von der größeren Anzahl an Strahlen in einem Verbund zuerst getroffen wird. Hierzu wird eine Kollisionsmaske erstellt, die für jeden Strahl eines Pakets einen Ganzzahlwert als Eintrag erhält, aus dem hervorgeht, ob überhaupt eine AABB getroffen wird bzw. mit welcher zuerst eine Kollision zustandekommt: Ein beide BVs verfehlender Strahl wird, wie in Zeile 21 des Codebeispiels 4.12 angewiesen, deaktiviert und in der Maske durch den Wert 0 repräsentiert; für ausschließlich oder zuerst mit der AABB des rechten Kindknotens kollidierende Strahlen wird der Wert 1, für solche, die nur oder zuvorderst auf das mit dem linken Kind assoziierte BV treffen, -1 eingetragen. Aus der Summe aller Werte in der Kollisionsmaske kann anschließend hergeleitet werden, welche AABB von den meisten Strahlen des Pakets zuerst getroffen wird: Ein negatives Ergebnis verweist auf die des linken, ein positives



(a) $\sum = -1$ **(b)** $\sum = 0$ **(c)** $\sum = 1$

Abbildung 4.1: Szenarios der Kollisionen eines Strahlenpakets mit zwei BVs. Für die in (a) und (b) gezeigten Situationen wird im Anschluß jeweils der Knoten des linken BVs zuerst traversiert, für das in (c) gezeigte Szenario entsprechend der Knoten des rechten BVs.

auf die des rechten Kindknotens, während eine in 0 resultierende Summe angibt, daß beide BVs von derselben Anzahl an Strahlen zuvorderst getroffen werden. Anhand dieser Information wird entschieden, welcher Knoten zur notwendigen späteren Untersuchung auf dem Stapelspeicher abgelegt und in welchem mit der Traversierung zunächst fortgefahren wird. In Abbildung 4.1 wird dieser Vorgang schematisch anhand möglicher Konstellationen mit den jeweiligen Einträgen in die Kollisionsmaske und der resultierenden Summe dargestellt.

```
1  // BVs beider Kinder getroffen?
2  if(hitCode[1] && hitCode[2] || hitCode[3]) {
3     __shared__ int hitMask[BLOCK_SIZE];  // gemeinsame Kollisionsmaske
4
5     // trage Kollisionsinformationen jedes Strahls in Maske ein
6     hitMask[tId] = (2 * right * (rightMinT < leftMinT) - 1) * ray.active;
7     __syncthreads();  // mache Eintragungen für alle Threads sichtbar
8
9     int sum = sumUp(hitMask);  // berechne Summe über alle Einträge
10
11    __shared__ int nearIdx;  // Index des zuerst zu traversierenden Knotens
12    if(tId == 0) {
13       nearIdx = 0;
14       if(sum > 0) nearIdx = 1;
15       stack[++stackTop] = children[!nearIdx];  // merke Knoten
16    }
17    __syncthreads();  // mache Index für alle Threads sichtbar
18    node = children[nearIdx];
19 } else if(hitCode[1])  // BV des linken Kindes getroffen?
20    node = children[1];
21 else if(hitCode[2])    // BV des rechten Kindes getroffen?
22    node = children[2];
23 else                   // keine BV getroffen
24    // [...] prüfe Stack
```

Codebeispiel 4.13: Kooperative Traversierung mit Stapelspeicher, Teil 2

Das beschriebene Vorgehen zur Auswertung des Kollisionscodes wird durch das Codebeispiel 4.13 dokumentiert. Darin wird in Zeile 3 für die Kollisionsmaske ein Datenfeld im Shared Memory eingerichtet, das für jeden Thread eines Blocks einen Ganzzahleintrag vorsieht. Dessen Berech-

nung wird in Zeile 6 veranlaßt. Die anschließend in Zeile 9 angestoßene Kalkulation der Summe über alle Werte in der Maske wird parallel ausgeführt. Als Basis hierfür dient eine hochgradig optimierte Lösung, die als Quelltextbeispiel im CUDA-Entwickler-SDK vorliegt.[1] Die Auswertung des Resultats dieser Berechnung geschieht in einem einzelnen Thread des Blocks: Dabei wird der Knoten ausgewählt, der auf dem Stapelspeicher abzulegen ist, sowie ein gemeinsamer Index definiert, der schließlich allen Threads den nächsten zu traversierenden Knoten anzeigt. Diese Vorgänge sind den Zeilen 11 bis 18 zu entnehmen.

Verfehlen alle Strahlen eines Pakets die BVs beider Kindknoten, wird die bereits diskutierte Pop-Operation auf dem gemeinsamen Stapelspeicher ausgeführt, um entweder den nächsten zu traversierenden Knoten von dort zu laden oder die Schnittpunktsuche in allen Threads des Blocks zu beenden. Wird die Traversierung fortgesetzt, sind zunächst alle Strahlen des Blocks zu reaktivieren, wie dies in Zeile 4 des Codebeispiels 4.11 veranlaßt wird. Der Grund hierfür liegt darin, daß ein Strahl immer dann deaktiviert wird, wenn er die AABBs beider Kinder des gerade untersuchten Knotens nicht durchstößt;[2] solange bei der Traversierung der Hierarchie abgestiegen wird, kann solch ein Strahl in den folgenden Schnittpunkttests ignoriert werden, da er das übergeordnete BV bereits verfehlt. Wird jedoch ein Knoten von dem Stapelspeicher geladen, kehrt hierdurch die Traversierung auf eine höhere Ebene in der Hierarchie zurück, und da für einen Strahl nicht gespeichert wird, welcher ergebnislose Schnittpunkttest zu seiner eventuellen Deaktivierung geführt hat, muß ihre Rechtmäßigkeit für jeden Traversierungspfad separat überprüft werden.

Mit seiner Strategie, ganze Strahlenpakete unter Einsatz gemeinsam genutzter Ressourcen zu verfolgen, überschreitet die in diesem Kapitel vorgestellte Traversierungsmethode die Grenzen eines Stream-Modells, in dem die Unabhängigkeit zwischen den parallelen Recheneinheiten streng gewahrt wird; die im Bereich des GPU-Computings exklusiv in CUDA gebotene Möglichkeit, im Rahmen von Thread-Blocks kooperativ vorzugehen, kommt hier umfassend zur Anwendung. In vielerlei Hinsicht repräsentiert das Verfahren also gerade die in Kapitel 2.4.2 auf Seite 38 angesprochene Erweiterung, die das allgemeine Stream-Konzept durch CUDA erfährt.

[1] `http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/reduction/doc/reduction.pdf`

[2] Diese Strategie verursacht redundante Schnittpunktberechnungen: Ein Strahl könnte bereits deaktiviert werden, wenn der Traversierungspfad auf einen Knoten festgelegt wird, dessen AABB von dem Strahl nicht durchstoßen wird, anstatt dies erst nach dem dann zwingend erfolglosen Kollisionstest mit den BVs seiner Kindknoten zu veranlassen. Zeitnahmen belegen aber, daß die in der hier dargestellten Lösung reduzierte Komplexität der Traversierung größere Vorteile bietet als eine möglichst frühe Deaktivierung, für die freilich zusätzliche Anweisungen auszuführen wären.

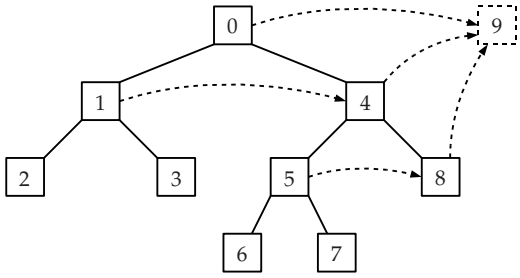
4.3.4 Traversierung ohne Stapelspeicher

In Form seiner dritten Traversierungsmethode wartet die Implementation mit einer Variante auf, die einen großen Teil der notwendigen Berechnungen in einem Vorverarbeitungsschritt durchführt und bei der tatsächlichen Traversierung deshalb ohne einen Stapelspeicher auskommt. Das Verfahren geht auf eine Veröffentlichung von Smits [Smi98] zurück und fand aufgrund der allgemeinen Vorteile, die nicht nur mit dem Verzicht auf einen Stack einhergehen, bereits sowohl bei der Traversierung einer BVH auf der GPU [TS05] als auch im geschwindigkeitsorientierten CPU-basierten Ray-Tracing [Gei06] Anwendung.

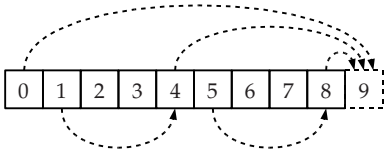
Die Methode baut auf der Idee auf, die Knoten der BVH derart in einem Array zu organisieren, daß ihre Reihenfolge gerade jener Abfolge entspricht, in der die Knoten bei einer regulären Tiefensuche traversiert werden. Nach diesem Schema konstruiert, entspricht ein linearer Zugriff auf alle Elemente des Datenfelds direkt wieder einer vollständigen Traversierung nach dem Prinzip der Tiefensuche. Um dabei gewisse Teilbäume zu ignorieren, genügt es, den entsprechenden Speicherbereich zu überspringen und bei dem Index im Array fortzufahren, welcher den ersten Knoten anzeigt, der dem Teilbaum nicht mehr angehört. Angegeben wird diese Position durch einen sogenannten *Abbruchzeiger*, welcher in jedem inneren Knoten vermerkt wird. Dieser kann bereits bei der rekursiven Konstruktion der BVH ausgerichtet werden: Der Wurzelknoten der BVH erhält als Abbruchzeiger einen ungültigen Verweis; bei der Erzeugung zweier Kinder wird dem linken Kindknoten die Speicheradresse des rechten als derartiger Zeiger zugewiesen, während der rechte Kindknoten den Abbruchzeiger des Vaterknotens erbt.

Die Umwandlung der BVH in ein Datenfeld mit der beschriebenen Organisation wird erreicht, indem die Knoten zunächst in einer vollständigen Tiefensuche mit fortlaufenden Indices versehen werden. Diese geben ihre Position im Array an. In einer zweiten Traversierung wird nun in jedem Knoten der Abbruchzeiger auch als Index desjenigen Knotens hinterlegt, auf den der Zeiger verweist – der ungültige Verweis, der stets das Ende der Traversierung signalisiert, wird dabei durch die Gesamtzahl der Elemente in dem Array repräsentiert, die gleichbedeutend mit dem ersten Index nach dem letzten gültigen Eintrag ist. In Abbildung 4.2 wird ein Beispiel für den Baum einer BVH und das resultierende Datenfeld gezeigt.

Zur Repräsentation eines Knotens wird auch bei dieser speziellen Darstellungsform der Hierarchie weiterhin der im Codebeispiel 4.7 auf Seite 65 definierte Strukturtyp eingesetzt – tatsächlich genügt aufgrund der eindeutigen Abfolge, in der die Knoten im Array vorliegen, erneut ein einziger als Zeiger fungierender Index, um alle notwendigen Information zu speichern: In einem Blatt verweist dieser nach wie vor auf das erste aller assoziierten Primitive, und in einem inneren Knoten repräsentiert er den Abbruch-



(a) Baum einer BVH



(b) Repräsentation als Datenfeld

Abbildung 4.2: Repräsentation der BVH bei der impliziten Traversierung. Als Pfeile eingetragen sind die Abbruchzeiger. Knoten 9 und Index 9 sind jeweils ungültig und signalisieren das Ende einer Traversierung.

zeiger. Alle anderen Verweise sind implizit: Nach der Traversierung eines Blattes wird gemäß der Tiefensuche immer in dem Knoten fortgefahren, der im Array direkt unter dem nächsten Index gespeichert ist, und dieselbe Strategie wird angewandt, um zu dem linken Kind eines Knotens zu gelangen, dessen BV von einem Strahl durchstoßen wird. Hier kündigt sich die besondere Konsequenz dieser Darstellung einer BVH für ihre Traversierung an: Sie wird strikt in derselben Abfolge vollzogen, in der die Knoten im Array gespeichert sind. Das bedeutet, daß die Kinder eines Knotens immer in der von der speziellen Speicherung diktierten Reihenfolge traversiert werden müssen und nicht – wie in den zuvor präsentierten Verfahren LST und SST – der Pfad gewählt werden kann, auf dem die Traversierung aller Wahrscheinlichkeit nach früher einen gültigen Schnittpunkt produziert.

Im Codebeispiel 4.14 wird die Implementierung des beschriebenen Verfahrens gezeigt, das im folgenden auch als implizite Traversierung bezeichnet wird. Daraus geht unmittelbar hervor, daß die Methode im Vergleich zu den auf Stapelspeichern basierenden Varianten eine stark vereinfachte Strategie verfolgt: Da hier wichtige Entscheidungen über den Verlauf der Schnittpunktsuche bereits bei der Erstellung der BVH getroffen werden, kommt der Algorithmus zur Traversierung mit nur wenigen Anweisungen aus. Insbesondere ist das Verfahren nicht auf spezielle Fähigkeiten CUDAs angewiesen, denn mit dem Verzicht auf einen Stack bedarf es auch keines besonderen Speicherraums für eine solche Datenstruktur. Somit dient die

```
1  unsigned int nodeId = 0;
2  int hitPrimId = -1;
3
4  // Abbruchindex = Anzahl der Knoten
5  unsigned int lastEscId = scene.bvh[0].ptr;
6
7  while(nodeId < lastEscId)
8     Node node = scene.bvh[nodeId];
9
10    // BV nicht getroffen?
11    if(!intersect(node, ray))
12       if(node.numPrim > 0) nodeId++;  // Blatt: nächster Knoten implizit
13       else nodeId = node.ptr;         // kein Blatt: folge Abbruchzeiger
14    // BV getroffen
15    else {
16       if(node.numPrim > 0)  // Blatt: schneide mit allen Primitiven
17          for(unsigned int i = 0; i < node.numPrim; i++)
18             if(intersect(scene.primitives[node.ptr + i], ray))
19                hitPrimId = node.ptr + i;
20
21       nodeId++;  // nächster Knoten implizit
22    }
23 }
24
25 return hitPrimId;
```

Codebeispiel 4.14: Traversierung ohne Stapelspeicher

Implementation dieser Methode einer Hierarchietraversierung im Rahmen der vorliegenden Arbeit der Untersuchung, inwiefern ein Algorithmus, der sich bereits früher in der GPU-Programmierung unter entsprechend ungünstigeren Voraussetzungen als gangbarer Weg erwiesen hat [TS05], Vorteile oder Nachteile mit sich bringt gegenüber komplexeren Lösungen, die auf erst jüngst in Graphikprozessoren eingeführte Fähigkeiten und eine erweiterte Programmierbarkeit angewiesen sind.

Kapitel 5

Integration

Der hier vorgestellte Ray-Tracer ist als ein abgeschlossenes System konzipiert, das eine zentrale Schnittstelle in Gestalt einer Klasse nach außen führt, anhand deren Methoden alle obligatorischen und spezifischen Einrichtungs- und Ausführungsschritte veranlaßt werden können. Die konsequente Kapselung der internen Vorgänge zur Bilderzeugung vereinfacht den beabsichtigten universellen Einsatz der Implementation: Während die Entwicklung hauptsächlich im Rahmen eines eigenständig lauffähigen Programms durchgeführt worden ist, erfolgte sie bereits unter der allgemeinen Zielsetzung, das System auch als Programmbibliothek für den Einsatz in anderen Konstellationen bereitstellen zu können. In dieser Form konnte der Ray-Tracer erfolgreich als Rendering-Modul in die Echtzeit-Ray-Tracing-Umgebung Augenblick eingebunden werden; die hierzu unternommenen Schritte werden in diesem Kapitel kurz dokumentiert.

Augenblick ist ein an der Universität Koblenz entwickeltes plattformunabhängiges Ray-Tracing-System, das in einem hochgradig optimierten CPU-basierten Verfahren interaktive bis echtzeitkonforme Geschwindigkeiten bei der Bilderzeugung erreicht. Es zeichnet sich dabei vor allem durch seine Vielseitigkeit aus, die sich zum einen bereits in den grundsätzlichen Fähigkeiten – beispielsweise darin, NURBS-Flächen direkt darstellen zu können – widerspiegelt, zum anderen aus der zum Prinzip erhobenen Erweiterbarkeit ergibt: Plug-In-Mechanismen ermöglichen es unter anderem, die Ray-Tracing-Verarbeitungskette neu zu gestalten, indem einzelne Glieder wie zum Beispiel das Shading neu formuliert oder ergänzt werden oder der gesamte Vorgang ersetzt wird durch ein alternatives Verfahren. Genau bei dieser Möglichkeit setzt das Unternehmen an, den hier implementierten Ray-Tracer in Augenblick als Rendering-Modul einzubinden.

Die Bildsynthese wird in Augenblick in einer sequentiellen Abfolge von Ausführungseinheiten vollzogen. Solche *Units* dieser sogenannten *Execution Chain* umfassen jeweils einen oder mehrere *States*; das sind die Zustände, die das System bei der Berechnung des Bildes einnimmt. Beispiele für

States sind die Strahlerzeugung, die Strahlverfolgung, das Shading oder auch die Umwandlung der Pixelwerte in ein darstellbares Format.

Für Execution States bietet Augenblick eine abstrakte Plug-In-Klasse an; Objekte einer hiervon abgeleiteten Klasse können als neue Zustände einer Ausführungseinheit registriert werden. Welche Operationen auszuführen sind, wenn der durch das Plug-In repräsentierte Zustand in einer Verarbeitungskette eingenommen wird, ergibt sich aus dem Überladen der virtuellen Funktion `execute()`. Außerdem können dem Plug-In-Objekt bei seiner Erzeugung Argumente zur Initialisierung von Parametern übergeben werden.

Zur Integration des hier implementierten GPU-basierten Ray-Tracers in Form eines Plug-Ins wird der gesamte Vorgang der Bildsynthese als ein Zustand formuliert; der Aufruf zur Ausführung aller Berechnungen erfolgt in der überladenen `execute()`-Methode. Das Plug-In wird als einziger State einer Unit und diese wiederum als einzige Einheit der neu zusammengesetzten Execution Chain bestimmt, durch welche die herkömmliche Verarbeitungskette ersetzt wird. Von hier an bewirkt ein Aufruf in Augenblick zur Erzeugung eines neuen Bildes, daß hierzu die in dem Plug-In formulierten Anweisungen ausgeführt werden – die Bildsynthese wird, wie ausführlich in dieser Arbeit beschrieben, vollständig auf der GPU vollzogen.

Dem Umstand, daß die einmalige Einrichtung eines Plug-Ins in einem anderen Thread des Betriebssystems vollzogen wird als die Aufrufe seiner Ausführungsmethode, ist geschuldet, daß für den erfolgreichen Einsatz des GPU-basierten Ray-Tracers in Augenblick seine Initialisierung ebenfalls in der Methode `execute()` implementiert werden muß; dieses Vorgehen ist notwendig, weil die bei der Entwicklung eingesetzte Runtime-API den Kontext, in dem CUDA alle relevanten Ressourcen verwaltet, implizit und unabänderlich an den Host-Thread bindet, in dem die jeweiligen Funktionen hierzu aufgerufen werden. Erst die systemnahe Driver-API erlaubt eine explizite Verwaltung des Kontexts und würde somit auch eine Initialisierung des Systems in jener Weise ermöglichen, die für Plug-Ins in Augenblick im allgemeinen vorgesehen ist.

Kapitel 6

Analyse und Bewertung

Eine ausführliche Auswertung des implementierten Ray-Tracing-Systems steht im Zentrum dieses Kapitels. Dabei werden in umfangreichen Tests die jeweiligen Stärken und Schwächen der drei Traversierungsvarianten ergründet und im Vergleich untereinander diskutiert, die Skalierungsfähigkeiten des Systems analysiert und eine allgemeine Einordnung auf der Basis einer Gegenüberstellung mit dem CPU-basierten Ray-Tracer Augenblick vorgenommen. Die während des Implementierungsvorgangs gesammelten Erfahrungen finden hier ebenfalls Erwähnung und werden gemeinsam mit den in der Leistungsauswertung ermittelten Ergebnissen für eine abschließende Beurteilung der CUDA-Entwicklungsumgebung herangezogen.

6.1 Allgemeine Erkenntnisse

Die im Vorfeld der systematischen Leistungsauswertung sowie im gesamten Verlauf der Entwicklung gewonnenen allgemeinen Erkenntnisse werden im folgenden zusammengetragen. Sie geben Aufschluß über die Eigenschaften der einzelnen Traversierungsmethoden und bilden eine Basis für die spätere Diskussion der Resultate der Zeitmessungen.

6.1.1 Optimale Konfigurationen

Für alle Modi, in denen das implementierte System bei der Bildsynthese verfahren kann, erweist sich eine Konfiguration mit quadratischen Thread-Blocks als ideal. Diese Erkenntnis geht konform mit den Erwartungen: Die Blocks repräsentieren dann ebenfalls quadratische Bildbereiche, wodurch sich unter den assoziierten Primärstrahlen und in entsprechend abnehmendem Maß auch unter den folgenden Sekundärstrahlen die größte Kohärenz erreichen läßt – d.h., die Wahrscheinlichkeit, daß bei der Schnittpunktsuche in den Threads eines Blocks ähnliche Traversierungspfade begangen

werden, ist für solche Konfigurationen am höchsten. Unmittelbar leuchtet die Bedeutung der Kohärenz vor dem Hintergrund der Vorgehensweise bei der SST ein, da hier explizit Strahlenpakete gemeinsam verfolgt werden. Aber auch für die LST und die implizite Traversierung erklärt sich der meßbare Vorteil einer möglichst kohärenten Strahlverfolgung: Die in Kapitel 2.3.2 auf Seite 31 beschriebene Serialisierung der Berechnung divergierender Ausführungspfade in den Threads eines Warps ist dann seltener bzw. in geringerem Umfang notwendig.

LST und implizite Traversierung erreichen ihre jeweils besten Leistungen mit einer Block-Größe, die aufgrund des Registerbedarfs der Threads bei der Ausführung der relevanten Kernels auch gleichzeitig das mögliche Maximum darstellt; sie beträgt für beide Verfahren 16×16 Threads. Leicht erklärt sich dieses Phänomen vor dem Hintergrund, daß beide Traversierungsmethoden keinerlei Kooperation zwischen den Threads eines Blocks vorsehen – für die Berechnungen ist es also prinzipiell unerheblich, ob die Warps, die ein SM verarbeitet, aus wenigen großen oder mehreren kleineren Blocks stammen, weil sie ohnehin unabhängig voneinander verarbeitet werden. Deshalb kommt hier schließlich die generelle Präferenz CUDAs für größere Blocks zum Tragen. Weiterhin erlauben die in diesem Layout jeweils 16 nebeneinander angeordneten Threads Coalescing beim Zugriff auf die Streams der Pixelwerte, da jene Anzahl gerade der Hälfte eines Warps entspricht; daß dieser Mechanismus tatsächlich zur Anwendung kommt, belegt eine Ausführungsanalyse durch den Profiler. Mit insgesamt 256 beinhalteten Threads entsprechen diese Block-Maße auch gerade dem Achtfachen der Größe eines Warps, womit der in Kapitel 2.3.1 auf Seite 30 erwähnten Empfehlung nachgekommen wird, die als Block-Größe ein Vielfaches der doppelten Anzahl an Threads in einem Warp nahelegt.

Frei von Beschränkungen durch den Ressourcenbedarf stellen sich für die SST 8×8 Threads umfassende Blocks als ideale Wahl heraus. Ein solches Ergebnis ist bemerkenswert insofern, als durch diese Maße zwar exakt die besagte doppelte Warp-Größe nachgebildet wird, die angesprochenen Möglichkeiten zum Coalescing jedoch ausdrücklich beschnitten werden: Die hierfür notwendige systematische Adressierung findet dann nicht mehr in der Breite einer Warp-Hälfte, sondern nur noch in der eines -Viertels statt; die Analyse der Ausführung durch den Profiler bestätigt die erwartete Verringerung der gebündelten Speicherzugriffe gegenüber Konfigurationen mit Thread-Blocks der doppelten Breite. Daß sich dennoch gerade dieses Layout als das leistungsfähigste erweist, hat neben der eingangs erwähnten Kohärenz verschiedene weitere Gründe: Zum einen bedeuten mehr Strahlen in einem Paket eine potentiell langwierigere Suche nach allen Schnittpunkten, weil ein größeres Strahlenbündel freilich die Wahrscheinlichkeit erhöht, daß mehr Pfade bei der gemeinsamen Traversierung der BVH begangen werden müssen. Zum anderen hat eine Ausdehnung der Blocks größere Verzögerungen bei Synchronisationen zur Fol-

ge: In den Threads muß dann länger auf die Fortsetzung ihrer Berechnungen gewartet werden, weil zunächst mehr andere Threads desselben Blocks die Synchronisationsbarriere erreichen müssen. In der Hauptsache ist es jedoch der besonderen Beschaffenheit der Kernels, in denen die SST zum Einsatz kommt, zuzuschreiben, daß hier ein Block-Layout, durch welches Coalescing effektiv vereitelt wird, die besten Voraussetzungen bietet:

6.1.2 Beschaffenheit der implementierten Kernels

Die arithmetische Dichte der Kernels, in denen das SST-Verfahren implementiert ist, erweist sich als derart hoch, daß selbst die nicht durch Coalescing beschleunigten Zugriffe auf den Global Memory vollständig mit Berechnungen überlagert werden können und sie somit die Laufzeit der gesamten Ausführung nicht negativ beeinflussen. Untermauert wird diese Feststellung durch die Ergebnisse einer Untersuchung der Leistungsfähigkeit der SST-Variante, wenn, wie in dem Kapitel 4.3.3 auf den Seiten 71 und 73 angedeutet, für das kooperative Laden der Primitive bzw. der Knoten Coalescing erzwungen und ebenfalls die in Kapitel 4.2.5 auf Seite 60 angesprochene Umwandlung des Datentyps `Hit` in ein »Structure-of-Arrays«-Layout vorgenommen wird, um auch hier die Speicherzugriffe zu bündeln: Obwohl das Profiling den Erfolg dieser Maßnahmen bescheinigt und die Anzahl der nicht durch Coalescing beschleunigten Zugriffe auf ein Minimum sinkt, wird hierdurch auch im Test mit anderen Konfigurationen keine Verringerung der Gesamtlaufzeit der Kernel-Ausführungen erreicht; im Gegenteil steigt diese gemeinsam mit einem nun zusätzlich erhöhten Registerbedarf der Threads an. Die dem Coalescing zuträgliche Vergrößerung der Transfereinheiten auf 64 oder 128 Bit verursacht noch gravierendere Geschwindigkeitseinbußen: Ein erneuter Blick in die Laufzeitanalyse führt ans Licht, daß jetzt die Zugriffe auf den Shared Memory infolge von Bankkonflikten serialisiert werden müssen. Außerdem werden nun weniger Threads in die Übertragung der Daten aus dem Global Memory in den Shared Memory einbezogen, wodurch sich der zuvor parallele Vorgang zunehmend in eine sequentielle Ausführung verkehrt. Sehr deutlich zeigt sich in diesem Versuch, daß die Kernels der SST nicht bandbreitenlimitiert sind, sondern ihre Laufzeit allein von der Berechnungskomplexität abhängt. Vorkehrungen für besonders effiziente Zugriffe auf den Global Memory fruchten bei dieser Traversierungsvariante deshalb nicht, sondern verursachen nur unnötige zusätzliche Berechnungsschritte und Serialisierungen, welche die Leistungsfähigkeit nachweislich negativ beeinflussen.

Eine andere Beschaffenheit weisen die Kernels der LST und der impliziten Traversierung auf: Ihre Ausführungsgeschwindigkeit verdoppelt sich nahezu, wenn zur Repräsentation von Vektoren im dreidimensionalen Raum nicht der in CUDA bereitgestellte Typ `float3`, sondern ein analoger, jedoch mit 128 Bit im Speicher ausgerichteter Strukturtyp verwendet

wird oder – auf diese Variante wird schließlich in den veröffentlichten Zeitnahmen zurückgegriffen – direkt der ebenfalls mit 128 Bit ausgerichtete Vektortyp `float4` zum Einsatz kommt. Hier wirkt die damit einhergehende Verringerung der Anzahl der Zugriffe auf den Global Memory in dem Maß, wie es für bandbreitenlimitierte Kernels zu erwarten ist. Als prinzipieller Nachteil dieser Traversierungsverfahren gegenüber der SST steht somit vorab fest, daß hier entsprechend mehr Speicherplatz beansprucht wird – in Grenzfällen läßt sich also eine Szenenbeschreibung nur noch dann vollständig in den Device-Memory kopieren, wenn entweder die SST zur Anwendung kommt oder die LST bzw. die implizite Traversierung unter für sie ungünstigen Bedingungen eingesetzt werden.

Freilich wird durch die Maßnahme, in der LST und der impliziten Traversierung auf im Speicher günstig ausgerichtete Datentypen zurückzugreifen, kein Coalescing für das Laden der Szenendaten möglich; die völlige Unabhängigkeit der Threads eines Blocks untereinander schließt das weiterhin aus. Anders verhält es sich bei den Zugriffen auf den Stream der Schnittpunktinformationen, wenn der ursprünglich hierbei eingesetzte Strukturtyp `Hit` in ein »Structure-of-Arrays«-Layout überführt und so die Daten in mehreren separaten Streams vorgehalten werden: Wieder kann mit Hilfe des Profilers das so erzwungene Coalescing nachgewiesen werden. Die erzielte Wirkung ist jedoch zwiespältig: Nur sehr geringfügig, d.h. im niedrigen einstelligen Prozentbereich, ergibt sich eine Beschleunigung; dem steht eine kritische Zunahme des Registerbedarfs der Threads gegenüber, die für einige Kernels der Verarbeitungsketten beider Traversierungsmethoden eine Verringerung der Block-Größe notwendig macht. Aus den Resultaten dieser Untersuchungen kann gefolgert werden, daß bei der LST und der impliziten Traversierung die Lesezugriffe auf die Szenendaten den weitaus größten Anteil der Gesamtlaufzeit der Kernels ausmachen – sie sind demnach bandbreitenlimitiert. Daß hier das erzwungene Coalescing – im Gegensatz zu dem Effekt dieser Maßnahme bei der SST – Beschleunigungen bewirkt, untermauert diese Feststellung. Gleichzeitig fallen die durch diese spezielle Optimierung erzielbaren Gewinne äußerst gering aus, weil sie nur einen periphären Vorgang betrifft; der zentrale Prozeß, der die größten Verzögerungen verursacht, nämlich das Laden der Szenendaten, wird davon nicht beeinflußt. Angesichts dieser Ergebnisse wird in der folgenden Leistungsauswertung auch bei der LST und der impliziten Traversierung auf den Einsatz des »Structure-of-Arrays«-Layouts zur Speicherung der Schnittpunktinformationen verzichtet; die zwar meßbaren, aber kaum wahrnehmbaren Geschwindigkeitsvorteile fallen weniger stark aus als die Einschränkungen, die mit dem gesteigerten Ressourcenbedarf einhergehen. Dieses Urteil wird vor dem Hintergrund der Zielsetzung gefällt, ein trotz aller Bemühungen um hohe Geschwindigkeit weiterhin praxistaugliches Ray-Tracing-System zu konzipieren; die Möglichkeiten aufrecht zu erhalten, Erweiterungen wie zum Beispiel ein komple-

xeres Shading einzuführen, wird als wichtiger eingestuft als die hier zu erzielende Beschleunigung.

6.1.3 Sonstige Ergebnisse

Die Tatsache, daß sich für die SST auf der einen und die Verfahren der LST und der impliziten Traversierung auf der anderen Seite unterschiedliche Kostenfaktoren bei der SAH-gelenkten Erzeugung der BVH als optimal herausstellen, ist leicht zu erklären: Erwartungsgemäß profitiert das auf der Verfolgung von Strahlenpaketen basierende Verfahren davon, wenn die für einen Traversierungsschritt veranschlagten Kosten im Vergleich höher angesetzt werden, da hier jeder zusätzlich traversierte Knoten den Berechnungsaufwand für alle Threads eines Blocks erhöht. In der Leistungsauswertung werden diese unterschiedlichen Präferenzen berücksichtigt.

Schließlich sei in dieser Zusammenfassung auf den Unterschied eingegangen, der sich zwischen einer direkten Anzeige der im Device-Memory vorliegenden Bilddaten und ihrem Transfer in den Host-Memory mit anschließender alternativer Darstellung offenbart: Zwar ist auf dem direkten Weg eine schnellere Bildanzeige möglich, der Geschwindigkeitszuwachs fällt mit etwa 30% absolut, d.h. allein für den Darstellungsprozeß gemessen, aber geringer aus als erwartet. Ohne Einsicht in die internen Vorgänge, die das Einbinden des OpenGL-Pixel-Buffer-Objects mit Hilfe der Funktionen der Runtime-API begleiten, kann dieses Ergebnis jedoch nicht tiefergehend untersucht werden. In der Praxis ist der gemessene Unterschied zwar nur von geringem Belang, denn freilich nimmt in dem implementierten Ray-Tracing-System der Prozeß der Bildanzeige nur einen minimalen Anteil an der Gesamtlaufzeit ein; dennoch kommt in der folgenden Leistungsauswertung ausschließlich die etwas schnellere Variante der unmittelbaren Darstellung der Bilddaten zum Einatz, da hier dem leichten Geschwindigkeitsvorteil keinerlei Nachteile gegenüberstehen.

6.2 Leistungsauswertung

Die Leistungsfähigkeit des implementierten Ray-Tracing-Systems wird in umfassenden Tests ermittelt. In den folgenden Kapiteln werden zunächst die Rahmenbedingungen für diese Erhebung erläutert und anschließend die Resultate der Zeitmessungen präsentiert sowie eingehend diskutiert.

6.2.1 Testrahmen

Für die Zeitnahmen kommen die in Tabelle 6.1 aufgeführten fünf Szenenmodelle zum Einsatz. Ihre unterschiedliche Beschaffenheit ermöglicht eine differenzierte Analyse der Stärken und Schwächen jeder einzelnen Traversierungsmethode des GPU-basierten Ray-Tracing-Systems.

Name	Dreiecke	Quelle
Bunny	69.451	Stanford Computer Graphics Laboratory[a]
Dragon	871.414	ebenda
Buddha	1.087.716	ebenda
Fairy-Forest[b]	174.117	The Utah 3D Animation Repository[c]
Porsche	591.123	bereitgestellt von der Universität Koblenz

[a] `http://graphics.stanford.edu/data/3Dscanrep/`

[b] Schlüsselbild 16 der Serie

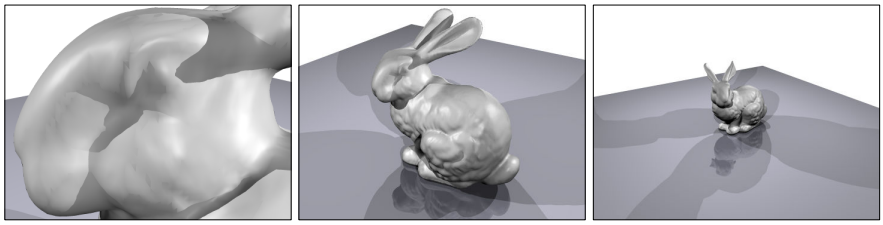
[c] `http://www.sci.utah.edu/~wald/animrep/`

Tabelle 6.1: Szenenmodelle

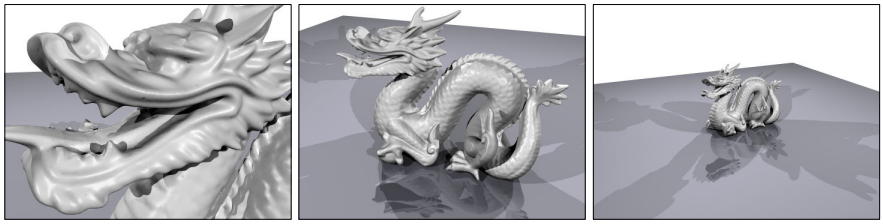
Die drei an der Universität Stanford durch die Rekonstruktion von Laserabtastungsdaten gewonnenen Modelle Bunny, Dragon und Buddha stellen einen gewissen Standard in Veröffentlichungen über Ray-Tracing-Verfahren dar. Ihr Erstellungsprozeß bedingt, daß die resultierenden Polygonnetze homogen aufgelöst sind und sich die Mengen der Primitive gleichmäßig über die Oberflächen der Modelle verteilen. Ein Effekt dieser Beschaffenheit und des räumlichen Zusammenhangs der Geometriedaten ist, daß sich auch in den hierfür erstellten BVHs eine gewisse Gleichmäßigkeit einstellt. Insgesamt können anhand der drei Modelle gut die Skalierungsfähigkeiten des Ray-Tracing-Systems hinsichtlich unterschiedlicher Polygonzahlen und -dichten untersucht werden.

Eine bewußt andere Charakteristik weist das Szenenmodell Fairy-Forest auf: Hier herrscht eine sehr ungleichmäßige Verteilung der Polygone vor, da detaillierte und filigrane Objekte abseits des Zentrums einer vergleichsweise grob modellierten Kulisse positioniert sind. Die für dieses Modell erzeugte BVH greift diese Beschaffenheit auf, indem hier mehrere dichte Anhäufungen von BVs innerhalb weiter leerer Räume plaziert werden. Mit diesen Eigenschaften eignet sich das Szenenmodell Fairy-Forest zur Analyse der Fähigkeit des Ray-Tracing-Systems, besonders inhomogene Konstellationen zur Darstellung zu bringen.

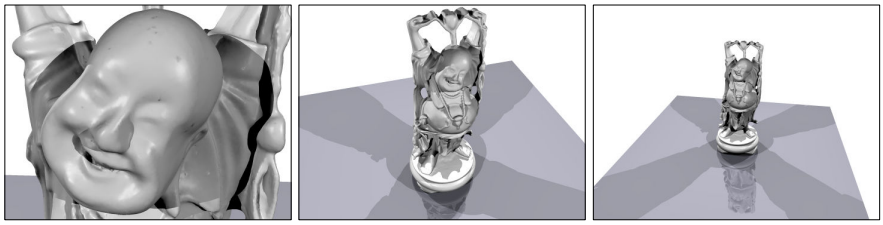
Das Szenenmodell Porsche positioniert sich hinsichtlich der Anzahl seiner Polygone in der Mitte der für die Leistungsauswertung herangezogenen Modelle, und auch in seinen strukturellen Eigenschaften ordnet es sich zwischen den Extrema ein: Hier wird eine allgemeine Geschlossenheit der Gesamtszene, wie sie die Stanford-Modelle auszeichnet, vereint mit einer ungleichmäßigeren Verteilung der Polygone. So setzt sich die Karosserie zu weiten Teilen aus vergleichsweise wenigen Elementen zusammen, während die offene Fahrgastzelle und darin befindliche feine Strukturen wie die Lüftungsschlitze, aber auch Details wie die tiefen Reifenprofile Herausforderungen für die BVH-Erzeugung und -Traversierung darstellen.



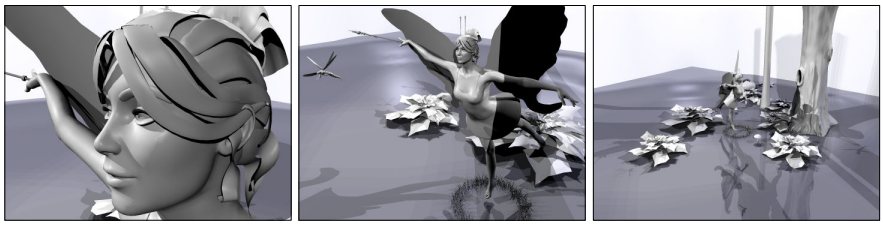
(a) Bunny



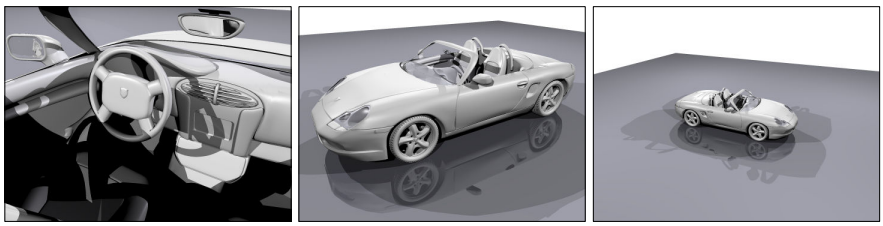
(b) Dragon



(c) Buddha



(d) Fairy-Forest



(e) Porsche

Abbildung 6.1: Szenenmodelle und Kameraeinstellungen. Von links nach rechts sind jeweils Großaufnahme, Halbtotale und Totale abgebildet. Die Darstellungen zeigen Ray-Tracing-Berechnungen mit vier Lichtquellen.

Um im Ray-Tracing-Modus hinreichend viele Sekundärstrahlen zu erzeugen, werden alle Szenenmodelle bis auf Fairy-Forest durch eine aus zwei Dreiecken zusammengesetzte Bodenfläche mit schwach reflektierenden Materialeigenschaften ergänzt – in Fairy-Forest wird hierfür die bereits im Modell vorhandene Grundfläche herangezogen. Weiterhin erhalten darin die vier die Szene einfassenden ineinander geschachtelten Ringe eine transparente Charakteristik. Im Modell Porsche werden Windschutzscheibe und Scheinwerfergläser transparent und alle Rückspiegel nahezu vollständig reflektierend definiert. Die Anzahl der beim Ray-Tracing verfolgten Indirektionen wird individuell festgelegt: Für die Stanford-Modelle genügt die einmalige Verfolgung von Sekundärstrahlen, da hier solche nur von der planen reflektierenden Bodenfläche ausgehen können. Für Fairy-Forest sind aufgrund der zahlreichen Transparenzen fünf Indirektionen vorgesehen, während für die Szene Porsche zwei als ausreichend erachtet werden.

In allen Szenen kommen jeweils drei unterschiedliche Kameraeinstellungen zur Anwendung: eine Großaufnahme, eine Halbtotale und eine Totale. Abbildung 6.1 zeigt, wie sich die Modelle aus diesen Betrachterpositionen jeweils darstellen.

Komponente	Spezifikation
CPU	Intel Core2 Duo E6600
	2,4 GHz Prozessortakt
	1,066 GHz Bustakt
Arbeitsspeicher	2 GiB DDR2 SDRAM DIMM
	800 MHz Speichertakt
Graphikkarte	NVIDIA GeForce 8800 GTX
	768 MiB
	575 MHz Prozessortakt
	900 MHz Speichertakt

Tabelle 6.2: Systemspezifikationen, Teil 1

Die wesentlichen Spezifikationen des Computersystems, auf dem die im folgenden veröffentlichten Ergebnisse aller Varianten des GPU-basierten Ray-Tracings ermittelt werden, sind Tabelle 6.2 zu entnehmen. Betrieben wird dieses System mit Linux. Die in den Vergleichen zusätzlich aufgeführten Leistungswerte der CPU-basierten Ray-Tracing-Umgebung Augenblick werden auf einem mit Mac OS X Leopard in der Version 10.5.4 betriebenen Computersystem der in Tabelle 6.3 angegebenen Spezifikationen ermittelt. Hierzu ist anzumerken, daß sich die von beiden Ray-Tracern erzeugten Bilder zwar in den meisten Fällen nur unmerklich, aber dennoch stets geringfügig unterscheiden. Dies hat systembedingte Gründe; zum Beispiel lassen die jeweils eingesetzten Shading-Routinen keine ein-

Komponente	Spezifikation
CPU	2 × Quad-Core Intel Xeon E5462 2,8 GHz Prozessortakt 1,6 GHz Bustakt
Arbeitsspeicher	2 GiB DDR2 SDRAM FBDIMM 800 MHz Speichertakt
Graphikkarte	NVIDIA GeForce 8800 GT 512 MiB

Tabelle 6.3: Systemspezifikationen, Teil 2

heitliche Berechnung von Glanzeffekten zu, und ebenfalls werden Transparenzen unterschiedlich gehandhabt. Weiterhin müssen für eine fehlerfreie Darstellung jeweils verschiedene Skalierungsfaktoren gewählt werden, um den mit Gleitkommazahlen einhergehenden Rechenungenauigkeiten zu begegnen. Es werden also für beide eingesetzten Systeme bewußt individuelle Vorkehrungen getroffen, damit sich die berechneten Bilder weitestgehend ähneln und dabei keine einseitigen Benachteiligungen entstehen. Der im folgenden durchgeführte Vergleich auf Grundlage der jeweils erzielten Resultate gibt demnach weniger absolute als vielmehr tendentielle Aufschlüsse, bleibt aber angesichts der erwähnten Bestrebungen gerechtfertigt und aussagekräftig.

6.2.2 Zeitnahmen und Diskussion

Alle in diesem Kapitel veröffentlichten Meßwerte geben die über 50 Darstellungszyklen gemittelten Bildwiederholungen pro Sekunde an; sämtliche Prozentangaben beziehen sich ebenfalls auf dieses Maß. Der Vorgang der Bildschirmanzeige wird als Bestandteil der Ausführungskette aufgefaßt und ist in den Zeitnahmen inbegriffen.

Ray-Casting

In Abbildung 6.2 sind die Ergebnisse der Ray-Casting-Berechnungen bei einer Bildauflösung von 640 × 480 Pixels aufgeführt. Direkt zu entnehmen ist den Werten, daß die implizite Hierarchietraversierung in keiner Messung die schnellsten, sondern statt dessen in den Großaufnahmen immer und in den Halbtotalen mit nur einer Ausnahme die langsamsten Bildwiederholraten in den Vergleich einbringt. Im Durchschnitt der Resultate für alle Szenenmodelle kann sich diese Variante nur in den Totalen – und hier allein gegenüber der SST – behaupten. Grundsätzlich reagiert die Traversierung ohne Stapelspeicher in ähnlicher Weise auf die Veränderungen der Kame-

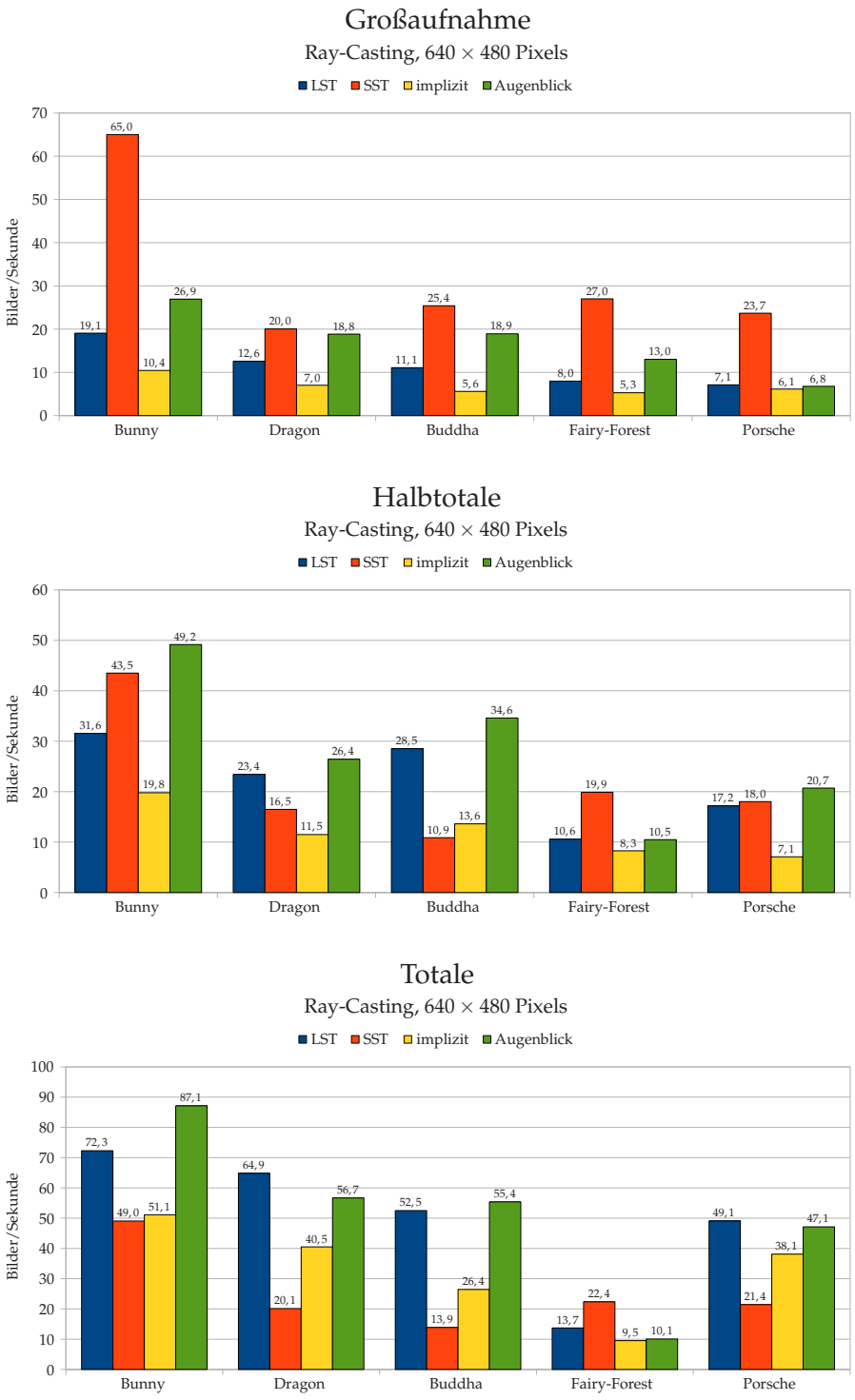


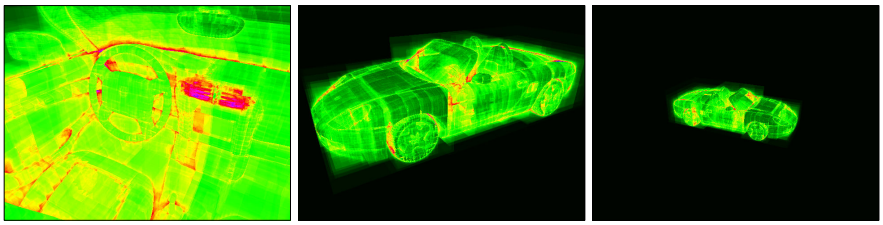
Abbildung 6.2: Zeitnahmen im Ray-Casting-Modus bei 640 × 480 Pixels

raeinstellungen wie die LST, bleibt dieser Methode aber in jedem Szenario meist deutlich unterlegen. In abgeschwächter Form lassen sich deshalb die Erkenntnisse aus der folgenden Gegenüberstellung von SST und LST auch auf die implizite Traversierung ausweiten, die hierbei nur noch allgemein einbezogen wird.

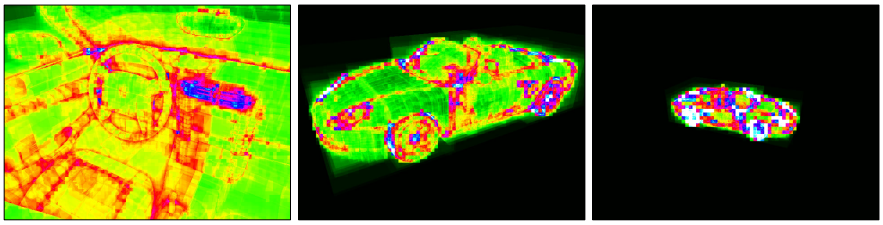
Ein Vergleich der unterschiedlichen Betrachterpositionen läßt Präferenzen für Großaufnahmen auf seiten der SST und für Totalen auf seiten der LST ausmachen – die Szene Fairy-Forest stellt hier eine noch zu begründende Ausnahme dar. In den Zeitnahmen für die Halbtotalen treten die Beschaffenheiten der Szenenmodelle in den Vordergrund: Geringe Polygonzahlen, wie sie in Bunny und Fairy-Forest vorherrschen, kommen der SST entgegen, während Dragon und Buddha mit ihren großen, gleichmäßig in hoher Dichte verteilten Mengen an Dreiecken mit der LST schneller zur Anzeige gebracht werden; bei der Darstellung des Modells Porsche, das Charakteristika aufweist, die gerade zwischen denen der zuvor genannten Szenen angesiedelt sind, kann sich in der Halbtotale keine der beiden Methoden absetzen.

Der deutliche Leistungsvorsprung der SST in den Großaufnahmen dokumentiert die erwartete Effizienz, die dem Schema der Verfolgung von Strahlenpaketen in Situationen hoher Kohärenz innewohnt. Sobald aber für die Strahlen eines Pakets zunehmend voneinander abweichende Traversierungspfade begangen werden müssen, wie dies beim Weg über die Halbtotalen hin zu den Totalen der Fall ist, verkehren sich die Vorteile des Verfahrens in ihr Gegenteil; innerhalb eines Thread-Blocks müssen dann sehr viel mehr Schnittpunktberechnungen durchgeführt werden als bei den Varianten, in denen die Traversierung für jeden Strahl unabhängig erfolgt. Veranschaulichen läßt sich dies durch eine Visualisierung der Anzahl der bei der LST und der SST jeweils durchgeführten Tests bei der Schnittpunktsuche; Abbildung 6.3 zeigt dies am Beispiel der Szene Porsche.

Daß in dem Modell Fairy-Forest der Weg von der Großaufnahme hin zur Totale nicht den sonst zu verzeichnenden Einbruch der Bildwiederholraten bei Verwendung der SST nach sich zieht, liegt zusätzlich durchgeführten Untersuchungen zufolge nicht allein daran, daß sich die Kameraeinstellungen hier bewußt nicht auf den gesamten Szenenkomplex beziehen, sondern auf das nur einen Teilraum einnehmende Arrangement um die Protagonistin; vielmehr verteilt sich hier die insgesamt bereits geringe Menge an Polygonen derart auf mehrere kleine und räumlich klar voneinander getrennte Gruppen, daß ihre jeweilige Dichte zu gering ausfällt, um das SST-Verfahren zu einer ineffizient hohen Anzahl an Schnittpunkttests in den Threads eines Blocks zu zwingen. Auffällig ist in diesem Zusammenhang, daß es den anderen Traversierungsmethoden hier nicht gelingt, aus den Veränderungen der Betrachterposition dieselben Vorteile zu ziehen wie in den anderen Szenen – da bei Fairy-Forest selbst in der Totalen noch nahezu jeder erzeugte Primärstrahl mit einem Polygon kollidiert und sich



(a) LST



(b) SST

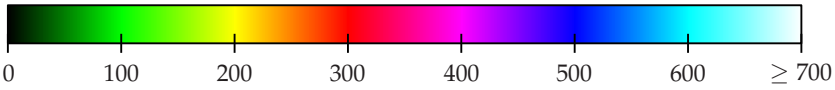


Abbildung 6.3: Visualisierung der Anzahl durchgeführter Schnittpunkttests. Der Skala sind die den jeweiligen Farben zugeordneten Gesamtzahlen an Schnittpunkttests zu entnehmen, die für das jeweilige Pixel im Bild mit BVs und Primitiven anfallen. Während die LST unabhängig von der Kameraeinstellung für dieselben Bereiche des Modells eine einheitliche Anzahl an Schnittpunkttests durchführt, zeigt die SST diesbezüglich eine starke Abhängigkeit von der Entfernung zum Modell. Zusätzlich ist in (b) die Blockstruktur auszumachen, die dem Verfahren der SST zugrundeliegt.

allgemein der Traversierungsaufwand für alle verfolgten Strahlen nicht in demselben hohen Maß reduziert wie bei den anderen Szenenmodellen, ändern sich hier die erreichten Bildwiederholraten über die verschiedenen Kameraeinstellungen hinweg nur geringfügig.

In einer Zusammenfassung erlauben die im Ray-Casting ermittelten Ergebnisse für die GPU-basierten Verfahren folgende Feststellungen: In den Varianten der LST und der impliziten Traversierung, in denen für jeden Strahl ein separater Traversierungspfad gewählt wird, skalieren die erzielbaren Bildwiederholraten mit den Anteilen derjenigen Bereiche an dem erzeugten Bild, für welche die Vorgänge der Schnittpunktsuche komplex ausfallen – je weniger Pixels insgesamt an der Darstellung detaillierter Szenenobjekte beteiligt sind, desto höher fällt die Berechnungsgeschwindigkeit aus. Dies steht im Gegensatz zu dem Skalierungsverhalten der SST-Variante, wo Gemeinsamkeiten oder Divergenzen bei der Schnittpunktsuche für

die Strahlen eines Pakets über die Geschwindigkeit des Verfahrens bestimmen – wo also die in einem Thread-Block vorherrschenden Verhältnisse die maßgebliche Rolle spielen. Deshalb sinken die Bildwiederholraten bei dieser Traversierungsmethode von den Großaufnahmen hin zu den Halbtotalen, steigen von dort zu den Totalen jedoch durchweg wieder leicht an: Je geringer die Anzahl der Thread-Blocks ausfällt, in denen aufgrund ausbleibender Kohärenzen über die Maße viele Traversierungsschritte durchgeführt werden müssen, desto leistungsfähiger erweist sich diese Variante.

Der CPU-basierte Ray-Tracer Augenblick zeigt sich grundsätzlich in derselben Weise orientiert wie die LST und erzielt in diesem Vergleich vor allem in den Großaufnahmen und den Halbtotalen sowie durchweg bei dem einfachen Szenenmodell Bunny bessere Ergebnisse; zu den Totalen hin verlieren sich die Vorteile jedoch weitgehend. Die in der Gegenüberstellung der GPU-basierten Verfahren festgestellte Dominanz der SST in den Großaufnahmen und in der Szene Fairy-Forest über alle Kameraeinstellungen hinweg bleibt, wenn auch teilweise weniger stark ausgeprägt, unter Einbeziehung der Resultate von Augenblick weiterhin bestehen. Im Durchschnitt über alle in einer Kameraeinstellung berechneten Werte erreicht der GPU-Ray-Tracer daher – dank der Leistungen der SST – in den Großaufnahmen deutlich höhere Bildwiederholraten als Augenblick, während in den Halbtotalen und Totalen, wenn auch mit weniger Abstand, das CPU-basierte Ray-Tracing die besseren Werte produziert.

Im Überblick über alle Messungen im Ray-Casting für eine Bildauflösung von 640×480 Pixels zeigt sich nur das SST-Verfahren in der Lage, durchweg zweistellige Bildwiederholraten zu produzieren; Augenblick fällt in einem, die LST-Variante in zwei Szenarios auf niedrigere Geschwindigkeiten ab. Insgesamt in den meisten Fällen kann der CPU-Ray-Tracer echtzeitkonforme Leistungen aufweisen; in diesem Vergleich plaziert sich auch die LST vor der SST. Die niedrigsten Werte in diesen Vergleichen und für jede Kameraeinstellung über alle Szenenmodelle hinweg werden von der impliziten Traversierung erreicht.

Ray-Tracing

Kameraeinstellung	LST	SST	implizit	Augenblick
Großaufnahmen	52	61	64	65
Halbtotalen	71	75	79	76
Totalen	79	79	86	76

Tabelle 6.4: Durchschnittlich erlittene Verluste beim Ray-Tracing mit einer Lichtquelle gegenüber dem Ray-Casting bei 640×480 Pixels in Prozent

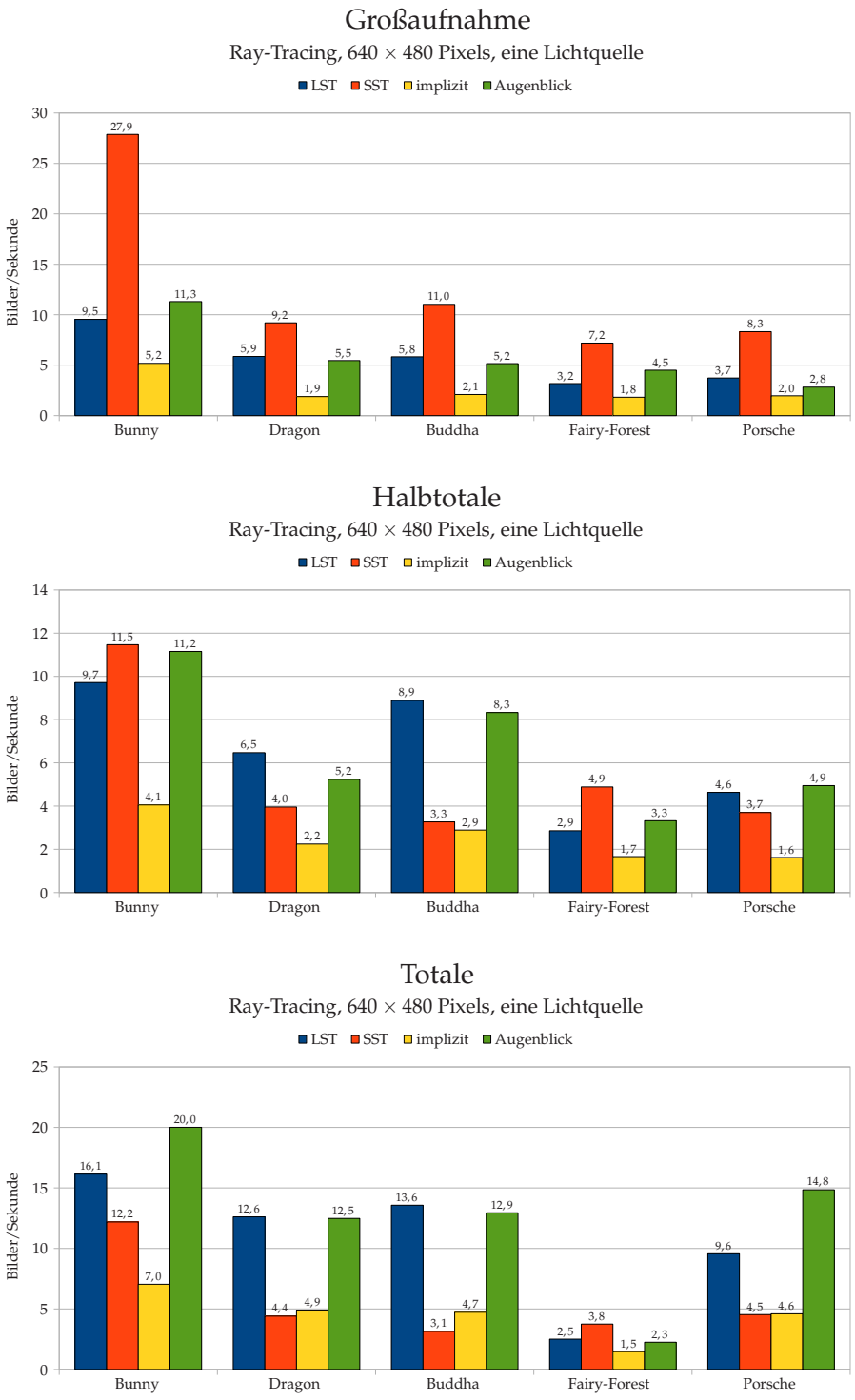


Abbildung 6.4: Zeitnahmen im Ray-Tracing-Modus bei 640 × 480 Pixels

Die Zeitnahmen der Ray-Tracing-Berechnungen mit einer Lichtquelle bei einer Bildgröße von 640×480 Pixels werden in Abbildung 6.4 zusammengefaßt. Die durchschnittlichen Einbußen in jeder Kameraeinstellung gegenüber den im Ray-Casting erzielten Bildwiederholraten sind der Tabelle 6.4 zu entnehmen. Aus den Werten geht hervor, daß die implizite Traversierung von den GPU-Ray-Tracing-Verfahren die empfindlichsten Verluste durch die jetzt zusätzlich notwendige Sekundärstrahlverfolgung erleidet. Damit setzt sich die LST durchweg weiter von dieser Methode ab, und auch die SST vermag ihre Position insbesondere in den Totalen gegenüber der Traversierung ohne Stapelspeicher zu verbessern. Dies ist bemerkenswert und aufschlußreich insofern, als sich bei der Verfolgung von Sekundärstrahlen generell weniger Kohärenzen ausnutzen lassen, was prinzipiell die deutlichsten Auswirkungen bei der SST erwarten läßt – dieses Traversierungsverfahren erweist sich jedoch als überraschend robust, denn verglichen mit der LST zeigt es diesbezüglich ein nur geringfügig schlechteres Skalierungsverhalten.

Der Schritt vom Ray-Casting zum Ray-Tracing verschiebt auch das Verhältnis zwischen den auf GPU und CPU ermittelten Werten: Wird wieder die LST-Variante zugrundegelegt, kann nun in den Großaufnahmen und den Halbtotalen allgemein von ausgeglichenen Ergebnissen gesprochen werden, während in den Totalen Augenblick seine Überlegenheit weiterhin bei Bunny und jetzt auch Porsche demonstriert.

Insgesamt erreicht unter den durch die Sekundärstrahlverfolgung verschärften Bedinungen die CPU-Variante am häufigsten zweistellige Bildwiederholraten, gefolgt von der SST und der LST. Die für jedes Verfahren aus allen Resultaten ermittelte minimale Geschwindigkeit fällt bei der SST am höchsten aus; auch die LST kann sich hier noch vor Augenblick positionieren. Die Traversierungsvariante ohne Stapelspeicher erreicht in all diesen Vergleichen erneut jeweils das schlechteste Ergebnis.

Kameraeinstellung	LST	SST	implizit	Augenblick
Großaufnahmen	27	35	29	33
Halbtotalen	30	30	35	28
Totalen	32	33	39	32

Tabelle 6.5: Durchschnittlich erlittene Verluste beim Ray-Tracing mit zwei Lichtquellen gegenüber dem Ray-Tracing mit einer Lichtquelle bei 640×480 Pixels in Prozent

In den über alle Szenen gemittelten Einbußen, die im Ray-Tracing beim Hinzufügen einer zweiten Lichtquelle zu verzeichnen sind, spiegeln sich grob die bisher ermittelten Ergebnisse wider: Die in Tabelle 6.5 angegebenen Werte weisen die LST und Augenblick als diesbezüglich insgesamt

unempfindlichste Verfahren aus, während die implizite Traversierung besonders in den Totalen vergleichsweise stark und die SST vor dem Hintergrund ihrer speziellen Abhängigkeit von Kohärenzen bemerkenswert gering an Geschwindigkeit verlieren. Hierzu sei angemerkt, daß nur die Traversierung ohne Stapelspeicher in einem Szenario einen Wert knapp unterhalb der für Interaktivität geforderten Rate von einem Bild pro Sekunde produziert; allen anderen Verfahren gelingt es, im Ray-Tracing mit zwei Lichtquellen durchweg interaktive und jeweils einmal auch zweistellige Bildwiederholraten zu erreichen.

Höhere Bildauflösungen

Bildauflösung	Modus	LST	SST	implizit	Augenblick
1280×960	Ray-Casting	71	43	70	69
	Ray-Tracing	71	46	68	68
1920×1440	Ray-Casting	86	63	86	85

Tabelle 6.6: Durchschnittlich in den Halbtotalen erlittene Verluste bei Vergrößerung der Auflösung gegenüber jener von 640×480 Pixels in Prozent

Zur Analyse des Skalierungsverhaltens bei Änderungen der Bildauflösung werden in Abbildung 6.5 aus Gründen der Übersichtlichkeit nur noch die Resultate für die Halbtotalen aufgeführt – bezüglich der Bildinhalte darf diese Einstellung als typisches Szenario gelten. Ergebnisse für die anderen Betrachterpositionen werden, sofern relevant, im Wort erwähnt. Der Tabelle 6.6 sind zusätzlich die durchschnittlichen Einbußen zu entnehmen, die sich jeweils gegenüber den zuvor ausführlich diskutierten Messungen in der Bildauflösung von 640×480 Pixels ergeben.

Bevor auf die herausstechenden Resultate der SST detailliert eingegangen wird, seien die ebenfalls offenkundigen Parallelen zwischen den beiden anderen für das GPU-Ray-Tracing implementierten Traversierungsvarianten erwähnt: Daß sich die Verhältnisse der LST und der impliziten Traversierung untereinander nur sehr geringfügig ändern, entspricht den Erwartungen, da, wie bereits beschrieben, für diese Methoden dasselbe Kriterium für die Skalierung gilt; durch die Vergrößerung der Menge an Pixels im Bild ändert sich nicht das Verhältnis, in dem die aufwendigen Traversierungsvorgänge zu den einfachen stehen. In ähnlicher Weise skaliert auch das CPU-basierte System Augenblick. Die nicht mehr aufgeführten Ergebnisse für die anderen Kameraeinstellungen fallen analog aus.

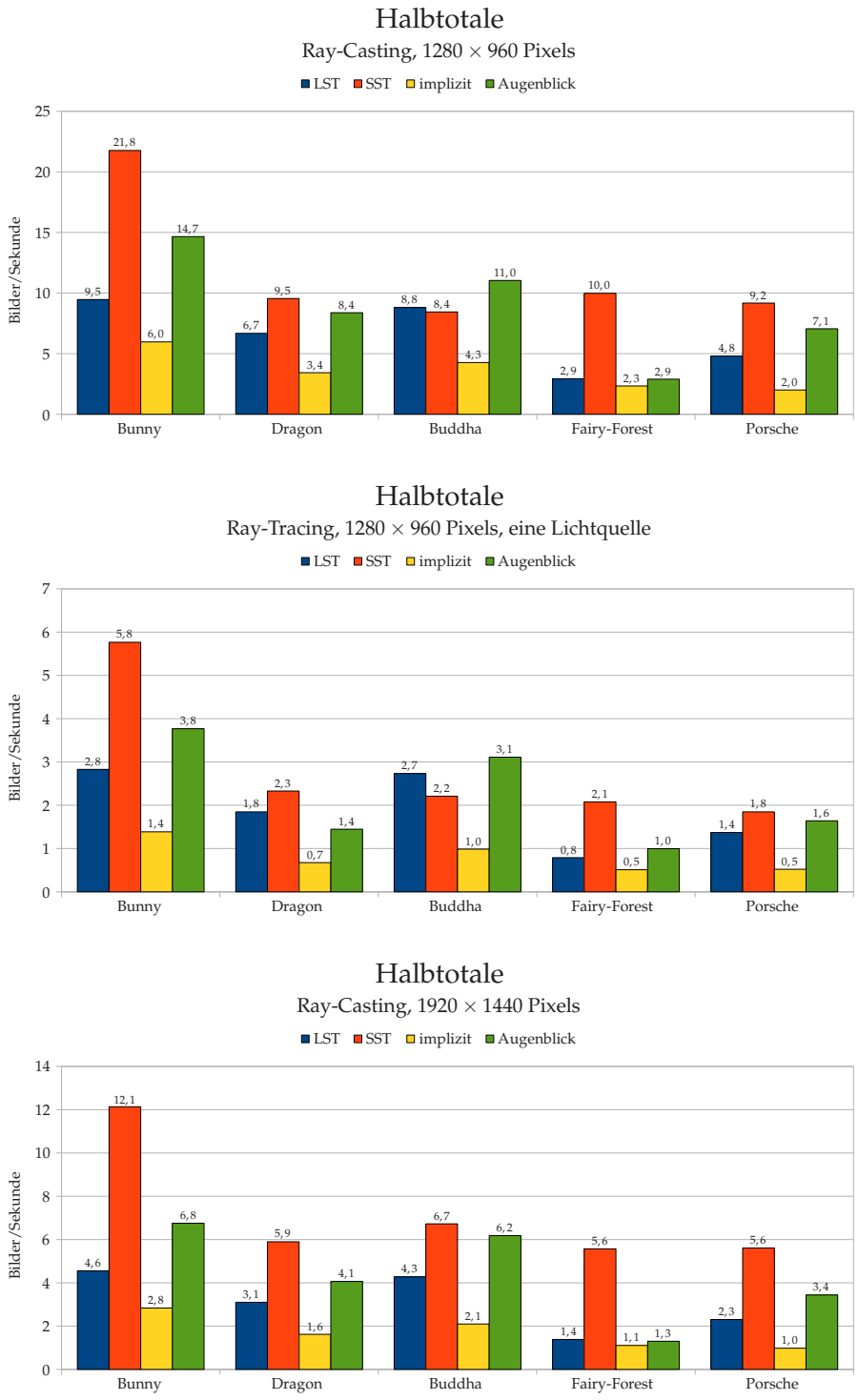


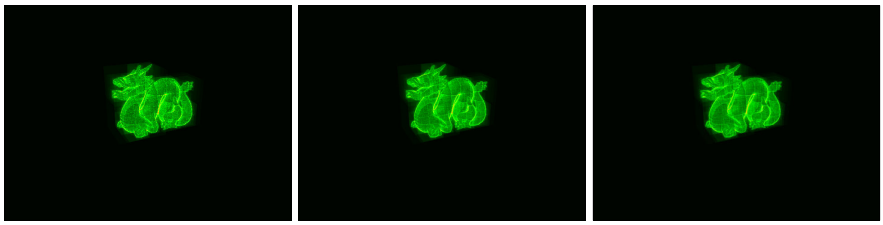
Abbildung 6.5: Repräsentative Zeitnahmen bei 1280 × 960 und 1920 × 1440 Pixels

Die SST weist nicht nur im Vergleich zu allen anderen Verfahren, sondern auch in Anbetracht der Verhältnisse, in denen sich die Anzahl der Pixels jeweils erhöht, besonders bemerkenswerte Skalierungseigenschaften auf: Für die Berechnung der vierfachen Menge an Werten sinkt die Bildwiederholrate im Durchschnitt um nicht einmal die Hälfte, für eine neunmal größere Pixelanzahl um nicht einmal zwei Drittel. Der angedeutete Effekt tritt in den Totalen noch stärker in Erscheinung, und in den Großaufnahmen, in denen die SST bereits alle durchgeführten Vergleiche dominiert, fallen die Verluste, wenn auch weniger ausgeprägt, noch immer kontinuierlich geringer aus als für alle anderen Verfahren.

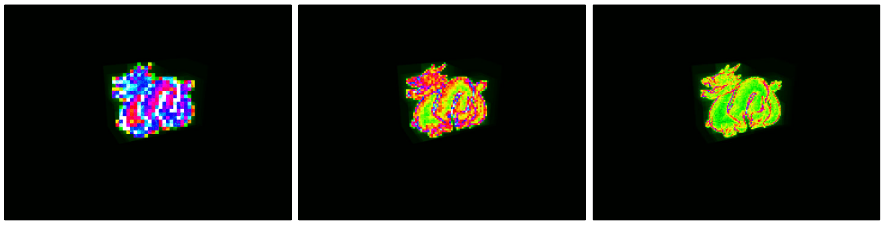
Das beobachtete Phänomen läßt sich anhand bereits festgestellter Tatsachen erklären: Für die Leistungsfähigkeit der SST stellen die Zusammenhänge innerhalb eines Strahlenpakets, also innerhalb der von einen Thread-Block verwalteten Pixels, ein entscheidendes Kriterium dar, während für die anderen Traversierungsmethoden und – das ergeben die durchgeführten Messungen – auch für das CPU-basierte Ray-Tracing-System Augenblick die in Pixelzahlen ausgedrückten Verhältnisse im gesamten Bild ausschlaggebend sind. Im Gegensatz zu jenem relativen Maß wandeln sich die Zustände in einem Pixelblock konstanter Größe bei Veränderung der Bildauflösung sehr wohl: Der Effekt ist identisch mit der vergrößerten Darstellung eines Bildausschnitts – und dies ist gerade das Szenario, in dem sich die SST besonders hervortut, nämlich das der Großaufnahme. Mit der Erhöhung der Bildauflösung steigt auch die für dieses Verfahren so essentielle Kohärenz unter den Strahlen eines Pakets; das grundsätzliche Mehr an Berechnungen insgesamt fällt deshalb weniger stark ins Gewicht.

Wie zuvor in Abbildung 6.3 auf Seite 93 die Nachteile der Verfolgung von Strahlenpaketen durch eine Visualisierung der Anzahl durchgeführter Schnittpunkttests aufgezeigt werden, lassen sich auf dieselbe Weise auch die jetzt hervortretenden Vorteile dieser Methode veranschaulichen: Abbildung 6.6 zeigt anhand der Totale in der Szene Dragon die Verringerung des in den Thread-Blocks betriebenen Aufwands bei der Schnittpunktsuche, der mit der Erhöhung der Bildauflösung einhergeht.

Zusammengefaßt zeigt sich in den höheren Auflösungen die SST am leistungsfähigsten: Im Ray-Casting werden damit sowohl die mit Abstand höchste Mindestgeschwindigkeit sowie am häufigsten zweistellige Resultate erreicht. Im Ray-Tracing mit einer Lichtquelle bei einer Bildauflösung von 1280×960 Pixels produziert ausschließlich dieses Verfahren in allen Kameraeinstellungen durchweg interaktive Bildwiederholraten, während Augenblick zweimal, die LST dreimal und die implizite Traversierung in der Mehrzahl aller Fälle Ergebnisse unterhalb des geforderten Wertes von einem Bild pro Sekunde erzielen. Der Vergleich zwischen den GPU- und CPU-basierten Lösungen fällt somit unter der Voraussetzung, daß die hier sehr leistungsfähige SST zur Anwendung kommt, deutlich zugunsten des Systems aus, das die Bildsynthese auf dem Graphikprozessor vollzieht.



(a) LST



(b) SST

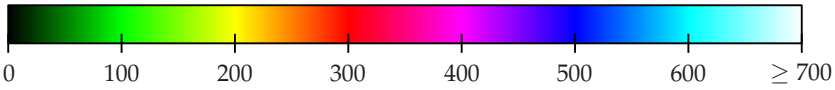


Abbildung 6.6: Visualisierung der Anzahl durchgeführter Schnittpunkttests. Jeweils von links nach rechts werden die Bedingungen bei den Bildauflösungen 640×480, 1280×960 sowie 1920×1440 gezeigt. Klar zu erkennen ist in (b) die günstige Entwicklung der Anzahl anfallender Schnittpunkttests bei der SST, während die LST, wie (a) zu entnehmen ist, aus den geänderten Bedingungen keine Vorteile ziehen kann.

Zusammenfassung

Das GPU-basierte Ray-Tracing-System, dessen Entwicklung in dieser Arbeit dokumentiert wird, erreicht mit seiner Methode der kooperativen Traversierung mit einem gemeinsamen Stapelspeicher in allen vorgestellten Szenarios mindestens interaktive Geschwindigkeiten. Besonders profiliert sich diese Variante bei Großaufnahmen und generell solchen Kameraeinstellungen, in denen Szenenobjekte bildfüllend angezeigt werden, sowie bei der Darstellung von Szenen, in denen sich die Polygone ungleichmäßig verteilen. Dichte Polygonmengen, die auf wenige benachbarte Pixels abgebildet werden, können durch die SST jedoch nur vergleichsweise langsam zur Anzeige gebracht werden. Das Verfahren zeigt schließlich ein ausgesprochen günstiges Skalierungsverhalten bei Erhöhung der Bildauflösung.

Die andere einen Stapelspeicher verwendende Variante LST weist ein insgesamt eher als traditionell zu bezeichnendes Verhalten auf: Das Verfahren skaliert linear mit einer Erhöhung der Bildauflösung und erreicht umso höhere Bildwiederholraten, je weiter sich der Betrachterstandpunkt

von Szenendetails entfernt. Mit dieser Methode positioniert sich das GPU-basierte System direkt gegenüber dem CPU-Ray-Tracer Augenblick und zeigt in diesem Vergleich eine geringfügig unterlegene Leistungsfähigkeit.

Die ohne Stapelspeicher arbeitende implizite Traversierung erweist sich insgesamt als das schwächste der drei implementierten Verfahren: Sie skaliert prinzipiell in derselben Weise wie die LST, erreicht dabei jedoch in allen Szenarios langsamere Geschwindigkeiten. Die statische Vorgehensweise, in der Knoten stets in derselben festgelegten Reihenfolge traversiert werden, ist insbesondere den Anforderungen der Verfolgung von Sekundärstrahlen nicht gewachsen.

LST und implizite Traversierung gemeinsam weisen gegenüber der SST den Nachteil auf, daß sie ihre Leistung nur mit Szenenbeschreibungen entfalten können, in denen zusätzliche Ressourcen für eine günstige Ausrichtung im Speicher in Anspruch genommen werden. Dies geht nicht aus den hier veröffentlichten Zeitnahmen hervor, muß aber erwähnt und in einer Gegenüberstellung berücksichtigt werden.

6.3 Beurteilung der Implementation

Mit dem im Rahmen dieser Arbeit implementierten Ray-Tracing-System werden die in Kapitel 4.1 auf Seite 52 formulierten Entwicklungsziele erreicht: Praxisrelevante Szenen lassen sich in gängigen Auflösungen durchweg mit interaktiven und vereinzelt auch mit echtzeitkonformen Bildwiederholraten darstellen. Die Fähigkeiten eines klassischen Ray-Tracers, namentlich Schattenwurf, Spiegelungen und Brechungen, werden in frei bestimmbarem Umfang unterstützt; daß Materialien nicht gleichzeitig reflektierende und lichtbrechende Eigenschaften aufweisen können, muß jedoch als – wenn auch systembedingter – Nachteil gewertet werden.

Die Leistungsauswertung bescheinigt dem GPU-basierten Ray-Tracing-System eine hohe allgemeine Geschwindigkeit. Die Implementation präsentiert sich dabei besonders flexibel bezüglich der Wahl der darzustellenden Szene, der Kameraeinstellung und der Bildauflösung: Mit den zwei leistungsfähigen und gleichsam gegenläufig skalierenden Traversierungsvarianten LST und SST werden stets praxistaugliche Resultate in Szenarios mit unterschiedlichen Anforderungsprofilen erzielt.

Der SST kann aufgrund ihrer auffälligen Leistungsfähigkeit unter traditionell schwierigen Bedingungen – Großaufnahmen sowie der Bildsynthese in hohen Auflösungen –, der LST hinsichtlich der gezeigten Fähigkeiten in klassischen Szenarios hohes Potential attestiert werden. Die dritte implementierte Methode zur Traversierung der BVH erzielt im Vergleich schlechte Resultate. Alle Varianten erfüllen jedoch gleichermaßen ihren Zweck bei der im folgenden Kapitel vorgenommenen Bewertung der bei der Entwicklung eingesetzten CUDA-Technik.

Im Vergleich mit einem optimierten CPU-basierten Ray-Tracing-System erweist sich die in dieser Arbeit vorgestellte Implementation als allgemein konkurrenzfähig. In den wichtigen Disziplinen der bildfüllenden Darstellung von Szenendetails und der Berechnung von Bildern in hohen Auflösungen kann sich der GPU-Ray-Tracer in seiner unter solchen Bedingungen besonders starken SST-Variante Vorteile erarbeiten, die im Vorfeld nicht zu erwarten waren.

Schließlich wird demonstriert, daß sich der implementierte Ray-Tracer dank seiner klar definierten Schnittstelle auch in ein bestehendes System durch die Nutzung standardisierter Plug-In-Mechanismen integrieren läßt. Über den Nachweis hinaus wird im Rahmen der vorliegenden Arbeit jedoch keine weitere Entwicklung oder Optimierung in dieser Hinsicht vorangetrieben.

6.4 Beurteilung der NVIDIA-CUDA-Technik

Die Resultate der Auswertung weisen die NVIDIA-CUDA-Technik insgesamt als sehr leistungsfähig aus. Insbesondere die Tatsache, daß gerade jene Traversierungsmethoden die besten Ergebnisse produzieren, die auf den speziellen und in der GPU-Programmierung neuartigen Konzepten CUDAs basieren, drückt aus, daß NVIDIA hier ein diesbezüglich bemerkenswert ausgereiftes Produkt anbietet: Der in dieser Arbeit gezielt angestrebte Vergleich mit einem ausschließlich traditionelle GPGPU-Techniken anwendenden Verfahren zeigt auf, daß CUDA nicht nur in der Theorie die GPU-Programmierung sinnvoll zu erweitern und sogar zu erneuern vermag, sondern auf diese Weise auch Möglichkeiten bietet, die Rechenleistung moderner GPUs noch effektiver für allgemeine Zwecke zu nutzen.

Getrennt von dem Nachweis der hohen Leistungsfähigkeit einer mit CUDA entwickelten Implementation ist die Beurteilung der Entwicklungsumgebung selbst vorzunehmen; sie stützt sich auf die während des Implementierungsvorgangs gesammelten Erfahrungen:

Der Einstieg in die Entwicklung mit CUDA gestaltet sich unkompliziert: Eine adäquate, da prägnante und praxisbezogene Einführung gibt der Programmierleitfaden, und verhältnismäßig schnell können Ergebnisse produziert werden. Als vorteilhaft erweist sich, daß der zur Programmierung eingeführte C-Dialekt lediglich die klar angegebenen Einschränkungen und darüber hinaus in keiner Weise syntaktische und nur minimale wiederum deutlich gekennzeichnete semantische Abweichungen[1] von den bekannten Spracheigenschaften aufweist. Auf dieselbe Weise hilfreich ist, daß der Compiler-Treiber nvcc das Aufrufschema des GNU-Compilers gcc nachbildet.

[1] Als Beispiel für eine Neuauslegung der Semantik sei das Schlüsselwort `extern` bei der Deklaration eines dynamisch verwalteten Speicherbereichs im Shared Memory erwähnt.

Die Lernkurve steigt jedoch abrupt steil an, sobald eine tiefere Einsicht in die Fähigkeiten und Vorgehensweisen CUDAs angestrebt wird, wie sie zum Beispiel notwendig ist, um eine Optimierung von Kernels hinsichtlich der Speicherzugriffe vorzunehmen: Die verfügbare Dokumentation nimmt früh den Charakter einer technischen Referenz an und versäumt es, Bedeutungsschwerpunkte zu setzen und den weiteren Lernfortschritt angemessen zu führen. Unausgewogen ist weiterhin die Erklärung der vielfältigen Komponenten der Entwicklungsumgebung; beispielsweise kommt die dynamische Verwaltung des Shared Memorys nur kurz und ohne Erläuterung ihres Zwecks und ihrer Möglichkeiten zur Erwähnung, und die Beschreibung der Verwendung des Constant-Memorys erschöpft sich in einem wortkarg kommentierten Quelltextfragment. Daß derartiges Wissen aus den – in variierendem Stil und uneinheitlicher Qualität dokumentierten – Beispielprogrammen des CUDA-Entwickler-SDKs bezogen werden kann, ist nicht als gleichwertige Lösung zu bewerten, zumal gewisse Fähigkeiten CUDAs sogar ausschließlich auf diesem Weg zu entdecken sind.

Als unberechenbar erweisen sich die Effekte von Laufzeitfehlern in Kernels: Werden die durch CUDA angestoßenen Berechnungen auf derjenigen GPU ausgeführt, der auch die Aufgabe der Monitordarstellung zukommt, erstrecken sich die Möglichkeiten im Fall eines Fehlers von einfachen Programmabbrüchen über nachhaltige Artefakte in der allgemeinen Bildschirmanzeige bis hin zum abrupten Stillstand des Computersystems. Mangels einer zusätzlichen zur Ausführung von CUDA-Programmen fähigen GPU kann keine Aussage darüber gemacht werden, inwiefern Laufzeitfehler auf einem dedizierten CUDA-Device weniger drastische Effekte bewirken. Schwierig gestaltet sich in diesem Zusammenhang auch die Problemanalyse: Die Bedeutung der einsehbaren Fehlerkennzahlen ist nicht Bestandteil der Dokumentation. Dabei hilft die Device-Emulation nur bedingt: Zwar werden hier manche Laufzeitfehler, zum Beispiel eine aufgrund der Programmstruktur nicht von allen Threads eines Blocks erreichbare Synchronisationsbarriere, mit einem entsprechenden Warnhinweis aufgedeckt; zuverlässig greifen diese Mechanismen jedoch nicht, und so entstehen Szenarios, in denen die Emulation fehlerfreien Betrieb zeigt, die Ausführung auf einem Device jedoch in der beschriebenen unberechenbaren Weise abbricht. Die Bemühungen von seiten des Herstellers, die traditionell problembehaftete Suche und Behebung von Fehlern in GPU-Programmen zu erleichtern, zeigen zwar vielversprechende Ansätze, jedoch sind die bislang angebotenen Lösungen unzureichend angesichts der Ambitionen, die NVIDIA mit der CUDA-Technik verfolgt.

Die im Rahmen dieser Arbeit eingesetzten sowie alle während des Entwicklungsvorgangs getesteten Funktionen CUDAs arbeiten fehlerfrei bis auf eine Ausnahme, in der ein identisch formulierter Zugriff auf den Constant-Memory reproduzierbar einmal korrekt ausgeführt wird und einmal einen vorzeitigen Programmabbruch bewirkt. Insgesamt kann der Lauf-

zeitkomponente dennoch eine hohe Stabilität attestiert werden, da sämtliche anderen im Verlauf der Entwicklung aufgetretenen Unzuverlässigkeiten stets auf Programmierfehler zurückzuführen waren. In dieser Hinsicht erfüllt CUDA durchaus den erhobenen Anspruch, ein ausgereiftes Produkt für den Einsatz in Industrie und Wissenschaft darzustellen.

Nicht immer zeigt die Übersetzung eines CUDA-Programms im einsehbaren produzierten Assembler oder auch im Profiling jene Resultate, die durch gewisse wohlbegründete Änderungen im Quelltext angestrebt werden. Hier wird die Möglichkeit vermißt, tiefere Einsicht in die Arbeitsweise CUDAs zu nehmen. Da solche Phänomene jedoch prinzipiell im Umgang mit jedem Werkzeug, das automatisiert Optimierungen vornimmt, auftreten können und somit freilich auch in der CPU-Programmierung nicht ausgeschlossen sind, sei der Umstand an dieser Stelle lediglich angemerkt und nicht als Anlaß zu einer negativen Bewertung genommen.

Insgesamt ist eine positive Beurteilung der CUDA-Entwicklungsumgebung gerechtfertigt. Zwar sind eine Vervollständigung und stilistische Vereinheitlichung der Dokumentation wünschenswert, und insbesondere für den Umgang mit Laufzeitfehlern müssen dringend befriedigende Lösungen gefunden werden – solche Defizite sind aber immer auch vor dem Hintergrund der hohen Komplexität und Flexibilität CUDAs zu bewerten, die jene bisheriger Möglichkeiten zur GPU-Programmierung weit übertreffen. Festzuhalten bleibt, daß NVIDIA mit dem Produkt CUDA erfolgreich in maßgeblichen Bereichen neue Wege eröffnet, den Graphikprozessor in verallgemeinerter Weise zu programmieren und einzusetzen.

Kapitel 7

Ausblick

In dieser Arbeit wird aufgezeigt, daß auf der GPU Ray-Tracing-Berechnungen in Geschwindigkeiten ausgeführt werden können, die jene eines dahingehend optimierten CPU-basierten Ray-Tracers erreichen und teilweise übertreffen. Eingesetzt wird hierbei die NVIDIA-CUDA-Technik, die nicht nur Traditionen fortführt, sondern auch innovative Elemente in die GPU-Programmierung einbringt; es wird durch Ergebnisse belegt, daß gerade diese neuen Konzepte der hohen Leistungsfähigkeit des implementierten Ray-Tracing-Systems erst den Weg ebnen. Die bei der Entwicklung und Auswertung gewonnenen Erkenntnisse über die Implementation auf der einen und die CUDA-Technik auf der anderen Seite lassen zahlreiche Möglichkeiten zu Erweiterungen und Verbesserungen in den Sinn kommen:

Die klar identifizierbaren individuellen Vorteile der beiden Traversierungsmethoden LST und SST ließen sich kombinieren, indem das in einer Situation jeweils leistungsfähigere Verfahren auf der Basis einer zu entwickelnden Heuristik ausgemacht und eingesetzt würde. Die hierzu notwendigen Berechnungen könnten parallel zur fortgeführten Bildsynthese asynchron auf der CPU ausgeführt werden. In derselben Weise ließe sich die BVH während der Bildberechnungen neu konstruieren, was das Ray-Tracing auch dynamischer Szenen in den hier gezeigten interaktiven Geschwindigkeiten zuließe. Ebenfalls denkbar sind Ansätze, in denen ein leistungsfähiger CPU-Ray-Tracer wie das in dieser Arbeit angesprochene Augenblick und das GPU-System parallel vollständige Bilder oder auch korrespondierende Halbbilder berechnen, um in Kooperation zu insgesamt höheren Bildwiederholraten zu gelangen.

Die von NVIDIA speziell für den professionellen Einsatz von CUDA lancierte Produktreihe Tesla umfaßt Lösungen, in denen mehrere Graphikprozessoren in einem Verbund für parallele Berechnungen eingesetzt werden – die Leistung des hier implementierten Systems erführe in der Ausführung auf derartigen Systemen unmittelbar eine Vervielfachung. Aber auch die Einbindung zusätzlicher CUDA-fähiger Graphikkarten ließe sol-

che Beschleunigungen mit vergleichsweise einfachen Mitteln zu. Weiterhin bieten die Graphikprozessoren der aktuellen Generation GT200 zahlreiche Neuerungen und Verbesserungen hinsichtlich des Einsatzes als CUDA-Device gegenüber der im Rahmen dieser Arbeit bei der Entwicklung und den Zeitnahmen eingesetzten GPU: Neben der obligatorischen Steigerung der allgemeinen Leistungsfähigkeit durch eine Vergrößerung der Anzahl paralleler Recheneinheiten, Anhebung der Taktfrequenzen sowie Optimierungen der Architektur sind hier zum Beispiel die Bedingungen für das Coalescing weitestgehend entschärft worden, so daß diese besonders effiziente Form des Zugriffs auf den Global Memory automatisch in mehr Fällen zum Einsatz kommt. Davon könnten im hier entwickelten Ray-Tracing-System unter Umständen die LST und die implizite Traversierung profitieren, denen eine Bandbreitenlimitierung nachzuweisen ist. Aber auch differenziertere Methoden zur Synchronisation der Threads eines Blocks werden mit den neuen Graphikprozessoren eingeführt; hieraus könnte eventuell bei der Verfolgung von Strahlenpaketen in der SST Nutzen gezogen werden.

Diese freie Auflistung ließe sich weiter fortführen – und tatsächlich finden sich kaum Anhaltspunkte dafür, daß entweder das implementierte Ray-Tracing-System oder die NVIDIA-CUDA-Technik im Rahmen der Implementierung auf unüberwindbare Grenzen gestoßen wären. Die vorliegende Arbeit endet deshalb mit der Anregung, die zahlreichen sich bietenden Ansätze und Möglichkeiten aufzugreifen, um die bisher erreichten Ergebnisse zu verbessern oder darauf aufbauend neue Entwicklungen vorzunehmen.

Literaturverzeichnis

[App68] APPEL, A.: *Some Techniques for Shading Machine Renderings of Solids*. In: *Proceedings of the AFIPS Spring Joint Computer Conference 1968*, Band 32, Seiten 37–45, 1968.

[Boh98] BOHN, C.-A.: *Kohonen feature mapping through graphics hardware*. In: *Proceedings of the 3rd International Conference on Computational Intelligence and Neurosciences 1998*, Seiten 64–67, 1998.

[CHH02] CARR, N. A., J. D. HALL und J. C. HART: *The ray engine*. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware 2002 (HWWS '02)*, Seiten 37–46. Eurographics Association, 2002.

[Cla76] CLARK, JAMES H.: *Hierarchical geometric models for visible surface algorithms*. Communications of the ACM, 19(10):547–554, 1976.

[Gei06] GEIMER, M.: *Interaktives Ray Tracing*. Der Andere Verlag, Tönning, Februar 2006.

[GPSS07] GÜNTHER, J., S. POPOV, H.-P. SEIDEL und P. SLUSALLEK: *Realtime Ray Tracing on GPU with BVH-based Packet Traversal*. In: *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, Seiten 113–118, September 2007.

[GS87] GOLDSMITH, J. und J. SALMON: *Automatic Creation of Object Hierarchies for Ray Tracing*. IEEE Computer Graphics and Applications, 7(5):14–20, Mai 1987.

[Hav00] HAVRAN, V.: *Heuristic Ray Shooting Algorithms*. Doktorarbeit, Czech Technical University in Prague, November 2000.

[HE99a] HOPF, M. und T. ERTL: *Accelerating 3D Convolution using Graphics Hardware*. In: *Proceedings of the Conference on Visualization '99 (VIS '99): Celebrating Ten Years*, Seiten 471–474. IEEE, 1999.

[HE99b] HOPF, M. und T. ERTL: *Hardware-based Wavelet Transformations*. In: *Proceedings of the Workshop of Vision, Modelling, and Visualization 1999 (VMV '99)*, Seiten 317–328, November 1999.

[HICK+99] HOFF III, K. E., T. CULVER, J. KEYSER, M. LIN und D. MANOCHA: *Fast computation of generalized Voronoi diagrams using graphics hardware*. In: *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, Seiten 277–286. ACM, August 1999.

[HMS06] HUNT, W., W. R. MARK und G. STOLL: *Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic*. In: *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, Seiten 81–88, September 2006.

[HSHH07] HORN, D. R., J. SUGERMAN, M. HOUSTON und P. HANRAHAN: *Interactive k-D Tree GPU Raytracing*. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games (I3D 2007)*, Seiten 167–174. ACM, 2007.

[ICSI85] IEEE COMPUTER SOCIETY, STANDARDS COMMITTEE und AMERICAN NATIONAL STANDARDS INSTITUTE: *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*, August 1985.

[KK86] KAY, T. L. und J. T. KAJIYA: *Ray tracing complex scenes*. In: *Proceedings of SIGGRAPH 86*, Computer Graphics Proceedings, Annual Conference Series, Seiten 269–278. ACM, August 1986.

[LRDG90] LENGYEL, J., M. REICHERT, B. R. DONALD und D. P. GREENBERG: *Real-time robot motion planning using rasterizing computer graphics hardware*. In: *Proceedings of SIGGRAPH 90*, Computer Graphics Proceedings, Annual Conference Series, Seiten 327–335. ACM, August 1990.

[MT97] MÖLLER, T. und B. TRUMBORE: *Fast, minimum storage ray-triangle intersection*. Journal of Graphics Tools, 2(1):21–28, 1997.

[NVI08a] NVIDIA: *NVIDIA CUDA – Compute Unified Device Architecture – Programming Guide, Version 2.0*, Juni 2008.
`http://www.nvidia.com/cuda`.

[NVI08b] NVIDIA: *NVIDIA CUDA Developer SDK, Version 2.0*, Juni 2008.
`http://www.nvidia.com/cuda`.

[PBMH02] PURCELL, T. J., I. BUCK, W. R. MARK und P. HANRAHAN: *Ray Tracing on Programmable Graphics Hardware*. ACM Transactions on Graphics, 21(3):703–712, Juli 2002.

[PGSS07] POPOV, S., J. GÜNTHER, H.-P. SEIDEL und P. SLUSALLEK: *Stackless KD-Tree Traversal for High Performance GPU Ray Tracing*. Computer Graphics Forum, 26(3), September 2007.

[Pho75] PHONG, B. T.: *Illumination for computer generated pictures*. Communications of the ACM, 18(6):311–317, 1975.

[RW80] RUBIN, S. M. und T. WHITTED: *A 3-dimensional representation for fast rendering of complex scenes*. In: *Proceedings of SIGGRAPH 80*, Seiten 110–116. ACM, Juli 1980.

[Smi98] SMITS, B.: *Efficiency issues for ray tracing*. Journal of Graphics Tools, 3(2):1–14, 1998.

[TAS00] TRENDALL, C. und A. A. STEWART: *General calculations using graphics hardware with applications to interactive caustics*. In: *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, Seiten 287–298. Springer-Verlag, Juni 2000.

[TS05] THRANE, N. und L. O. SIMONSEN: *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Diplomarbeit, University of Aarhus, August 2005.

[Wal04] WALD, I.: *Realtime Ray Tracing and Interactive Global Illumination*. Doktorarbeit, Universität des Saarlandes, 2004.

[Wal07] WALD, I.: *On fast construction of SAH based bounding volume hierarchies*. In: *Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing*, 2007.

[WBMS05] WILLIAMS, A., S. BARRUS, R. K. MORLEY und P. S. SHIRLEY: *An efficient and robust ray-box intersection algorithm*. Journal of Graphics Tools, 10(1):49–54, 2005.

[Whi80] WHITTED, T.: *An improved illumination model for shaded display*. Communications of the ACM, 23(6):343–349, 1980.