RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing

Stephan Kleber*
Universität Ulm

Abbildung 1: Einige Beispiele der Testszenen, die von der RPU gerendert wurden.

Zusammenfassung

Auf der SIGGRAPH 2005 wurde das Paper „RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing“ [Woop et al. 2005] vorgestellt.

Die Ray Processing Unit (RPU) ist eine voll programmierbare SIMD-Recheneinheit, die versucht auf die speziellen Anforderungen des Raytracing einzugehen, wobei die Berechnung in Echtzeit unterstützt werden soll. Dabei folgt das Design dem aktueller Grafikprozessoren (GPU) und verfolgt das Ziel des Renderns von dynamischen Szenen mittels Raytracing effizient und kostengünstig mit wenig Hardwareaufwand.

Um die beschriebenen Konzepte zu testen, wurde ein Prototyp erstellt, der bereits einige Szenen, mehr oder weniger in Echtzeit, berechnen kann (vgl. Abbildung 1).

Es handelt sich bei dieser RPU um eine Weiterentwicklung des SaarCOR, der ein „fixed-function“ Raytracing-Prozessor [Schmittler et al. 2004] ist, der in der Lage ist Raytracing in Echtzeit durchzuführen.

Da das Projekt immer noch in einer relativ frühen Phase seiner Entwicklung ist, können zahlreiche Unzulänglichkeiten und praktische Probleme festgestellt werden. Einige davon sind implementierungsbedingt, andere prinzipieller Natur. Jedoch erscheint das Konzept zunächst hoffnungvoll und einer weiteren Entwicklung und Beobachtung wert.

1 Einleitung

Sven Woop, Jörg Schmittler und Philipp Slusallek, jeweils Saarland Universität, stellten auf der SIGGRAPH 2005 das Paper „RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing“ [Woop et al. 2005] vor. Dort beschreiben die Autoren ihre Arbeit am Prototyp eines Prozessors, der Raytracing in Echtzeit mit wenig Hardwareaufwand ermöglichen soll. Wie leider an vielen aktuellen Papers solcher Konferenzen (beispielsweise SIGGRAPH, EUROGRAPH) zu sehen ist, werden oft ungelöste Probleme und Nachteile der vorgestellten Technik ausgeblendet. Daher bemüht sich die vorliegende Arbeit um eine objektivere Sicht auf die Leistungen des Entwicklerteams der RPU.

*e-mail: stephan.kleber@informatik.uni-ulm.de

1.1 Motivation und Abgrenzung

Im Licht der Beobachtungen des vorigen Abschnittes will diese Arbeit eine Übersicht über die Ausführungen im vorgenannten SIGGRAPH-Paper geben und dabei kritisch auf die Eigenschaften des Prototyp eingehen. So sollen insbesondere die angepriesenen Neuerungen sowie die Leistungsfähigkeit hinterfragt werden.

Was vorliegende Arbeit jedoch nicht kann und will, ist, sich als abschließende Kritik zu verstehen. Der Prototyp, der an der Saarland Universität erstellt wurde, wird weiterentwickelt und einige der Probleme können womöglich schnell gelöst sein. Vorliegende Arbeit stellt daher auch keine Lösungen bereit, sondern weist nur auf Unklarheiten hin.

2 Argumente für Raytracing

2.1 Prinzip des Raytracing

Die zentrale Idee des Raytracing besteht darin den physikalischen Vorgang des Sehens mathematisch nachzubilden. Dazu wird eine Gerade von der Lichtquelle in den Raum „geschossen“. Diese Gerade wird auf Schnittpunkte mit Primitiven, also Objekten oder Teilen von Objekten untersucht. Abhängig von der Oberflächendefinition des getroffenen Objekts werden dann weitere Strahlen berechnet. Dieser Vorgang wird rekursiv und in der Regel bis zu einer vorgegebenen Rekursionstiefe fortgesetzt. Strahlen, die schließlich zum Betrachter geworfen werden, bestimmen das Aussehen der Grafik. Auf diese Weise können fotorealistische Szenen beleuchtet und gerendert werden, die unter Umständen viele tausend oder Millionen von Primitiven und eine Vielzahl von Lichtquellen enthalten. Doch schon bei einer einzigen Lichtquelle trifft der Großteil der Strahlengeraden kein Objekt und wird daher unnötig berechnet.

Weitaus effizienter ist die Methode des Raycasting. Hierbei wird vom betrachtenden „Auge“ aus, durch jedes darzustellende Pixel eine Gerade berechnet und deren Schnittpunkte mit Objekten der Szene berechnet. Dieses Verfahren verfolgt also die Lichtstrahlen zurück letztendlich bis zu einer Lichtquelle. Dabei können - je nach verwendetem Shader an der entsprechenden Oberfläche - Reflexionen, Refraktionen und Absorptionen auftreten, die das Aussehen des Pixels verändern.

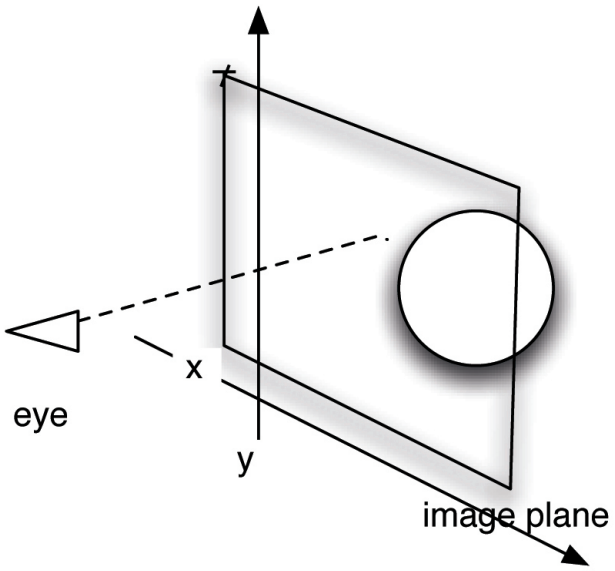

Abbildung 2: Raycasting: Der Strahl trifft vom Auge aus auf die Sphäre. Die Bildebene ist in die Pixel des Ergebnisbildes untergeteilt. Für jedes Pixel wird ein Primärstrahl berechnet.

Dieses Verfahren ist aber immer noch so rechenaufwändig, dass Raytracing, das in der gängigen Terminologie meist eigentlich das Raycasting meint, nur mit großem Hardwareaufwand in Echtzeit betrieben werden kann. Daher wurde für Spiele und andere Technologien, die auf eine Echtzeitberechnung von Szenenrenderings auf einfacher Hardware angewiesen sind, bisher der Weg der Rasterung (Scanline Rendering) gewählt. Hierbei wird die Raumtiefe durch den Z-Buffer simuliert, der die räumliche Reihenfolge der Objekte beinhaltet, wodurch Objektüberlagerungen sehr einfach auf die Bildfläche übertragen werden können. Jedoch kann die Rasterung nicht die optischen Effekte nachbilden, die vom Raytracing ganz selbstverständlich abgedeckt werden.

Um solche Effekte bei der Rasterung zu erzeugen, müssen etwa Light- und Shadowmaps und aufwändige Shader vordefiniert werden, die solche Strahlengänge approximieren. Der heutige Stand der Entwicklung hat diese Verfahren gut im Griff und somit sind die Ergebnisse sehr realistisch. Der Aufwand bei der Vorbearbeitung von Szenen ist allerdings im Vergleich zum Raytracing, das ohne Maps auskommen kann und dessen Shader einfach gehalten sein können, da der Strahl für weitere Berechnungen entlang seines Verlaufs an andere Shader weitergereicht wird, relativ hoch.

2.2 Argumente für die Hardwareumsetzung

Um nun die beschriebenen Vorteile des Raytracing auch bei Anwendungen nutzen zu können, die aufgrund der Echtzeitanforderungen bisher der Rasterung vorbehalten waren, haben sich die Autoren des zu besprechenden RPU-Papers um eine Umsetzung des Raytracing-Algorithmus in einem spezialisierten Prozessor bemüht. Aufbauend auf den Erfahrungen mit dem SaarCOR-Projekt [Schmittler et al. 2004] [Schmittler 2006], haben die Autoren eine GPU-ähnliche Architektur geschaffen, die sich einige Eigenschaften der Daten- und Berechnungsstrukturen, die beim Raytracing auftreten, zunutze macht, um durch Parallelisierung und Speicherzugriffsoptimierungen, die durchaus bekannten Probleme bei der Umsetzung des Raytracing anzugehen. Der dabei entstandene Prototyp zeigt, dass dieser Ansatz einer Raytracing-Implementierung in Hardware schon jetzt in einem gewissen Rahmen echtzeitfähig ist. Die teilweise aufwändige Aufbereitung der Szenen für eine spätere Rasterung entfallen beim Raytracing, was eine Erleichterung für die Entwicklung entsprechender Software bedeutet.

3 Design und Architektur

3.1 Herausforderungen

Auf dem Weg zu einem echtzeitfähigen Raytracing-Prozessor gibt es einige Probleme, deren Lösungen angesprochen werden.

Eine große Herausforderung stellt die **räumliche Indexstruktur** dar, die die zu berechnende Szene so unterteilen soll, dass es möglich wird, schnell in Frage kommende Objekte ausfindig zu machen. Dies ist deshalb nötig, da der Test auf einen Schnittpunkt einer Lichtstrahlgeraden mit einem Objekt ressourcenintensiv ist und nach Möglichkeit nur bei einer geringen Zahl von Objekten ausgeführt werden sollte. Durch einen solchen Index kann dann mit den Informationen über die Strahlen so navigiert werden, dass nur noch die indizierten Räume mit den darin enthaltenen Objekten betrachtet werden müssen, die ein Strahl tatsächlich durchläuft.

Eine weitere Anforderung an eine Hardwareeinheit, die zum Raytracing in komplexen Szenen verwendet werden soll, ist ein **flexibler Kontrollfluss**. Die Abhängigkeit der Shader, von rekursiv zu berechnenden Strahlengängen und anderer Informationen, die zum aufsummierten Wert eines Bildpunktes führen, ist so groß, dass ein sehr frei programmierbarer Prozessor unumgänglich ist, um allen Ansprüchen an die Hardware gerecht zu werden.

Ein großes Problem stellt die Menge der zu verarbeitenden Daten und damit der Zugriff auf den Speicher dar. Da die Szenendaten, die von der Hardware für das Raytracing benötigt werden, sehr umfangreich sein, dass heißt mehrere Gigabyte an Speicher belegen können, bedarf es einer **Optimierung beim Speicherzugriff**, um hier keinen all zu großen Flaschenhals entstehen zu lassen.

Um die damit zur Verfügung stehenden Daten auch zeitgerecht verarbeiten zu können, wurde ein **paralleler Ansatz** gewählt. Vektoroperationen werden zeitgleich, parallel zu weiteren Vektoroperationen oder Skalarberechnungen ausgeführt.

3.2 Indexstruktur (TPU)

Um die Szene für eine effiziente Bearbeitung zu strukturieren, wurde als Index der *k*d-Tree gewählt. Dieser wird in vorliegendem Design von der Software vorausberechnet, um dann von der Traversal Processing Unit (TPU) genutzt zu werden. Die TPU traversiert den Indexbaum und ruft den Shader auf, durch den am Schnittpunkt des Lichtstrahls mit einem Objekt dessen Oberflächen- oder Durchgangseigenschaften beschrieben werden. Eine gesonderte Einheit

für diese Aufgabe wurde deshalb vorgesehen, weil die nötigen Berechnungen linear erfolgen müssen und daher der parallele Subprozessor der RPU, der für die Shader genutzt und später noch näher beschrieben wird, unnötig von seiner eigentlichen Aufgabe abgehalten würde, was die Effizienz der ganzen RPU mutmaßlich deutlich verringert hätte.

Ein Hauptproblem bei der Verwendung einer Indexstruktur stellen jedoch dynamische Szenen dar, in denen sich ein Objekt aus einem indizierten Bereich in einen anderen bewegt. In diesem Fall muss ein Teil oder gar die komplette Indexstruktur, im vorliegenden Fall der *k*d-Tree, neu erzeugt werden. Der Umstand, dass dieses prinzipielle Problem besteht und die Entwickler der RPU daraus weiterhin eine Aufgabe der Software machen, stellt den Anspruch in Frage, einen Grafikprozessor entwickelt zu haben, der Raytracing komplett in Hardware realisiert. Andererseits muss auf die Art des Prototyps Rücksicht genommen werden, für den ein FPGA herangezogen wurde, was die Komplexität der Hardware deutlich beschränkt.

3.3 Kontrollfluss (SPU)

Um mächtige Shader generieren zu können, die möglichst allen Ansprüchen gerecht werden, ist die dafür vorgesehene Shader Processing Unit (SPU) mit Befehlen für bedingte Anweisungen (conditional branches) und Rekursion (masked recursion) ausgestattet. Ein zusätzlicher Registerstack, der bei den Sprunganweisungen implizit verwendet wird, erleichtert die Programmierung dadurch, dass praktisch ohne weiteres Zutun immer der korrekte Kontext für die anstehende Berechnung zur Verfügung steht. Damit ist ein flexibler Kontrollfluss gewährleistet, der es der SPU ermöglicht, jeden gewünschten Algorithmus auszuführen.

3.4 Speicherzugriffe (SPU)

Die Idee, wie die Speicherzugriffe möglichst effizient gestaltet werden, entspringt der Beobachtung des Raytracing-Algorithmus. Die Daten, die bei einer Berechnung der Strahlen vom Auge direkt zum Objekt benötigt werden, sind leicht einzuschätzen und im Cache vorzuhalten. Dies liegt daran, dass diese Strahlen eine hohe Kohärenz aufweisen, wie [Woop et al. 2005] es formuliert. Die Strahlengeraden treffen meist im Bündel auf ein Objekt, so dass dessen Daten nur einmal geladen werden müssen. Durch den parallelen, weil vektoriellen Charakter der Daten, ergeben sich viele gleichartige und benachbarte Speicherzugriffe, was das Caching weiter begünstigt, da eine Speicherzeile in der Regel mehrere komplette Vektoren enthält.

Dies gilt für Sekundärstrahlen nicht mehr. Diese Strahlen werden von einem Shader erzeugt, um beispielsweise über eine Reflexion die Farbe der Oberfläche zu bestimmen oder eine Refraktion zu berechnen. Der Ausgangpunkt des Strahls ist dabei der Endpunkt des Primärstrahls auf der Oberfläche des direkt zu sehenden Objekts. Durch die Vielzahl von denkbaren Richtungen, die dieser Strahl, abhängig von der Position der Kamera in der Szene, annehmen kann, ist eine günstige Speicherposition des nächsten Objektes, das für eine Schnittberechnung benötigt wird, extrem unwahrscheinlich, so dass Speicheroptimierungen der besagten Art nicht greifen können.

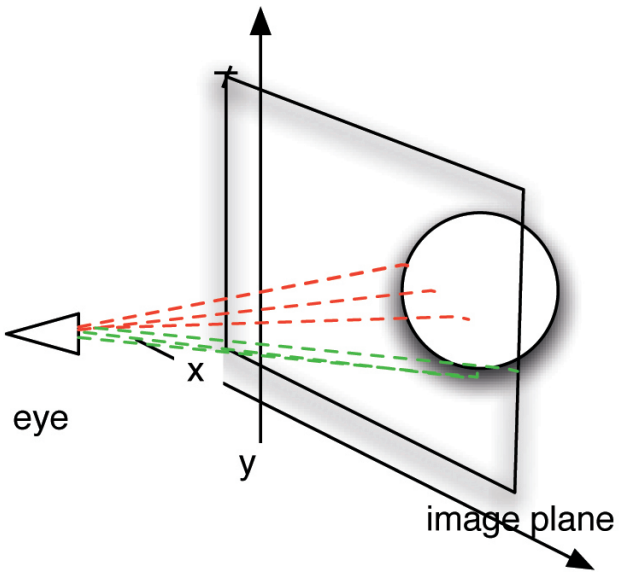

Abbildung 3: Strahlenbündel bei Primärstrahlen: Für die Strahlen, die das Objekt treffen, greifen die Speicheroptimierungen.

3.5 Parallelität (SPU)

Zunächst wird ein Vektor, wie sie die Szenen- bzw. Objektinformationen überwiegend sind, auch als ein Vektor betrachtet und so lassen sich dessen Komponenten parallel berechnen. Dies macht sich die SPU zunutze und stellt sich somit als eine Single Instruction Multiple Data (SIMD) ALU dar. Allerdings ermöglicht sie zusätzlich bei einer Unterteilung ihrer 4 Vektorkomponenten in 2/2 oder 3/1 Komponenten eine weitere, zeitgleiche Operation, wenn nicht alle Komponenten von einer der verwendeten Instruktion benötigt werden. So ist beispielsweise die Instruktionszeile `dp3 R0.xy,R1,R2 + mov R4.w,R5.w` möglich. Dieser Befehl arbeitet auf einer 3/1-Teilung und führt mit den Registern R1 und R2 ein Skalarprodukt der Komponenten durch, schreibt die Ergebnisse in die x und y Komponenten des Registers R0 während gleichzeitig der Inhalt der w Komponente von R5 nach R4 in die w Komponente verschoben wird.

Zur Parallelität in Form von Vektordaten kommt, wie zuvor schon erwähnt, noch die Ähnlichkeit bzw. Kohärenz verschiedener Schnittpunkte, in der Szene, die es zu rendern gilt. Die Strahlengeraden treffen meist im Bündel auf ein Objekt, das auf seiner Oberfläche überall denselben Shader verwendet. Somit bietet es sich durch die Optimierungen im Bereich des Speicherzugriffs an, die unabhängigen Daten, die ohnehin in den Cache geladen werden, parallel in mehreren SPUs für notwendige, gleichartige Berechnungen, also bei Laden und Ausführen des gleichen Codes, zu nutzen. Auch das entspricht dem typischen Verhalten eines Raytracing-Algorithmus. Dazu werden nach Bedarf aus den zur Verfügung stehenden Daten Threads generiert, die an SPUs zur Berechnung weitergegeben werden. Alle einem Satz SPUs dabei parallel zugeführten Threads werden unter dem Begriff Chunk zusammengefasst, der eine Analogie zu den Strahlenbündeln (vgl. oben) darstellt. Sollten zwischen Threads Abhängigkeiten vorhanden sein, so wird dynamisch in einer der SPUs zur Ausführung des Threads, dessen Daten zuerst benötigt werden, umgeschaltet. Die Autoren nennen dieses Konzept „mixed thread and stream programming model“.

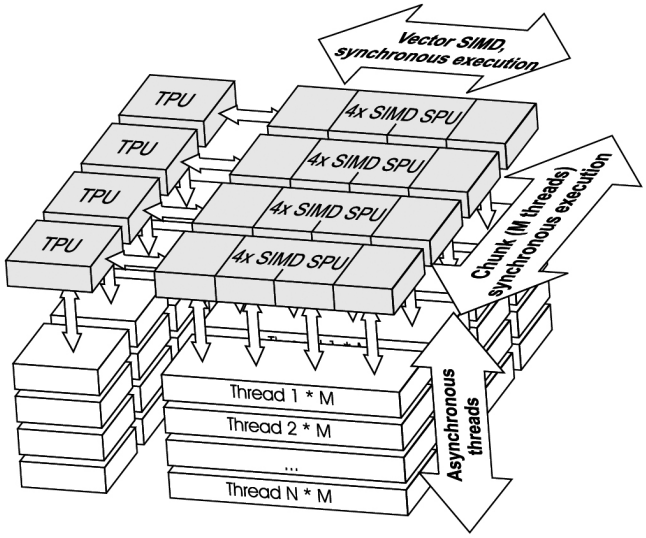

Abbildung 4: Parallele Vorgänge sind die 4-Komponenten-Vektor-Operationen der SPUs, sowie die Threads, die zu einem Chunk zusammengefasst werden.

Auch hier stellen die Sekundärstrahlen eine Herausforderung dar, denn deren Berechnungen folgen nicht dem vorgenannten Schema, so dass der Vorteil der Aufteilung in Chunks verloren geht. Allerdings wurden in [Schmittler 2006] Konzepte angedeutet, die die Strahlenbündel der Sekundärstrahlen zur Berechnung wieder so gruppieren, dass das Prozessormodell wieder effizient funktionieren kann. [Woop et al. 2005] erwähnt dies zwar nicht explizit, aber wenn nicht bereits geschehen, so bieten sich diese Methoden zur Erprobung auch in der RPU an. Die Probleme beim Speicherzugriff für Sekundärstrahlen bleiben aber bestehen.

4 Implementierung

Das vorgestellte RPU-Design ist unter anderem darauf ausgelegt, gut skalierbar zu sein. Die Autoren stellen dies unter Beweis, indem sie mehrere komplette RPUs auf jeweils eine PCI-Karte zusammenstellen, die dann parallel an einer Szene arbeiten. Dazu werden einige Komponenten gemeinsam genutzt, wie die Speicherzugriffsschnittstelle und der Thread Generator. So nutzt der ganze RPU-Cluster die Chunk-Eigenschaften der Daten beim Speicherzugriff aus, wobei jede RPU für sich entsprechende Daten in ihrem Cache hält. Auf diese Art und Weise ist es nicht nur möglich RPUs, wie mit dem Prototyp gezeigt, in einem Rechner auf mehreren Erweiterungskarten verteilt zu implementieren, sondern es ist genauso denkbar, mehrere RPUs in einem IC, auf in mehreren ICs auf einer Erweiterungskarte oder sogar über PC-Cluster verteilt zu nutzen.

Hierbei muss bedacht werden, dass eine solche Anwendung beim aktuellen Stand der Entwicklung darauf angewiesen ist, dass jede übergeordnete Hardware-Einheit, welcher Art auch immer, den zugehörigen RPUs eine Kopie der kompletten Szene bereithalten muss. Somit müssen eventuell mehrere Kopien großer Szenen gespeichert werden, was die Speichernutzungseffizienz deutlich senkt.

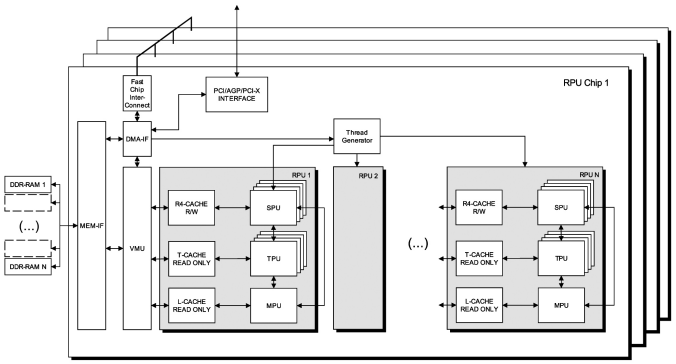

Abbildung 5: Schema der möglichen Skalierung. Es existiert nur eine gemeinsame Speicherschnittstelle, die zum Engpass werden kann. Allerdings hält im Versuchsaufbau jede der Karten eine eigene lokale Kopie der Szene.

4.1 Besonderheiten

Insbesondere gegenüber dem Scanline-Rendering hat der Ansatz des Raytracing den eingangs schon erwähnten Vorteil, dass eine komplett deklarative Szenendefinition möglich und somit keine Nachbearbeitung der Szenendaten nötig ist. Die RPU bietet über die flexibel programmierbaren Shader eine ausreichend Mächtige Möglichkeit, sowohl Texturen als auch BRDFs und Ähnliches zu implementieren.

Die Shader können aufgrund der großen Freiheit im Kontrollfluss prozedural aufgebaut werden. Somit lassen sich komplexe Codes modular schreiben und verwenden. Dies sorgt für zusätzliche Flexibilität und hilft redundanten Code zu sparen, was der Entwicklung und der Bereithaltung der Shader im Speicher zugute kommt. So können beispielsweise ein Geometry Shader und ein Lighting Shader unabhängig vorliegen, aber beide auf einen Objektpunkt angewandt werden, ohne dass die Shader ihre unabhängige Einsetzbarkeit verlieren.

Auch dynamische Szenen profitieren von diesem Vorteil, denn so können sich verändernde Objekte leicht an ihr neues Aussehen angepasst werden. Allerdings wird hiermit ein bereits genanntes Problem nicht gelöst, denn der Neuaufbau der Indexstruktur in einer dynamischen Szene wird, wie gesagt, nicht von der Hardware erledigt.

4.2 FPGA-Prototyp

Der Prototyp der RPU, die von den Autoren vorgestellt wurde, ist in einem Field Programmable Gate Array (FPGA) implementiert worden. Somit konnte die Hardware für diesen ersten Test sehr schnell und unkompliziert erstellt werden. Der verwendete Chip wird mit 66 MHz betrieben und hat damit im Vergleich zu heute gängigen GPUs eine sehr niedrige Taktung. Trotzdem lassen sich bereits Szenen in Echtzeit rendern.

Ein solcher IC enthält 4 SPUs, für die bis zu 32 Threads für die Berechnung bereitgehalten werden können. Zwischen diesen Threads kann nach Bedarf umgeschaltet werden, so dass sich Abhängigkeiten auflösen lassen. Jeder dieser SPUs ist eine TPU zugeordnet, so dass 4 unabhängige *k*d-Tree-Index-Traversierungen und die danach folgenden Shaderaufrufe parallel durchgeführt werden können. Die 4 SPUs nutzen gemeinsam einen Cache für die Szenendaten, die TPUs einen weiteren für Daten aus dem *k*d-Tree.

Neben diesen steht noch eine so genannte Mailboxed List Processing Unit (MPU) zu Verfügung. Diese hält in einem weiteren Cache Informationen über eine bestimmte Anzahl bei der Traversierung bereits durchgeführter Schnitte mit Objekten bereit, so dass diese nicht noch einmal berechnet werden müssen, falls jene Daten benötigt werden. Im konkreten Prototyp wird ein Cache für bis zu 4 solcher Datensätze angeboten.

Durch den begrenzten Platz auf dem FPGA und der daraus resultierenden, relativ niedrigen möglichen Komplexität der Schaltung, mussten einige vorgestellte Konzepte eingeschränkt oder auf Optimierungen verzichtet werden. So konnte die Speicherschnittstelle nur rudimentär verwirklicht werden. Es gibt keine Unterstützung für Bursts oder andere Effizienzsteigerungen beim Speicherzugriff. In den Tests ist für große Szenen eine entsprechende Leistungseinbuße zu sehen, die auf den geringen Speicherdurchsatz zurückzuführen ist. Ebenfalls aufgrund der geringen möglichen Komplexität, mussten die auf dem FPGA vorhandenen Fließkommaeinheiten verwendet werden, die nur eine Präzision von 24 Bit bei Operationen zulassen. Da für alle Strahl-Objekt-Schnitte, ein lokales Koordinatensystem verwendet wird und somit mehrere Matrixmutliplikationen notwendig sind, um Weltkoordinaten mit Objektkoordinaten verrechnen zu können, könnten sich bei dieser Präzision sichtbare Artefakte ergeben. Laut Aussage des Textes und anhand der zur Verfügung stehenden Bilder gibt es bei den Testszenen aber keine solchen Effekte.

Des weiteren besitzt der Chip überhaupt keine Integer-ALU, so dass es deutliche Einschränkungen speziell bei der Speicheradressierung gibt.

Der Code der Shader muss in der RPU resident sein. Durch den eingeschränkten Platz ist es nur möglich maximal 16 verschiedene Shader in der RPU zu nutzen. Ein Shader kann im gegebenen Fall maximal 512 Instruktionen enthalten, welche auf eine Länge von 80 Bit beschränkt sind. Zudem sind nur 4 Parameter-Register vorgesehen, die für die Übergabe von Parametern an den aufgerufenen Shadercode zur Verfügung stehen.

5 Ergebnisse

5.1 Leistungsfähigkeit des Prototypen

Der vorgestellte FPGA-Prototyp hat eine theoretische Spitzenleistung von 2,6 GFlops. Bei den Testszenen konnten bis zu 20 fps erzielt werden, allerdings nur, wenn ausschließlich Primärstrahlen berechnet werden sollten, also keine Rekursion stattfand. Je nach verwendeter Szene war die Leistung laut der Autoren besser als mit OpenRT auf einem Pentium 4, der mit 2,66 GHz getaktet ist, was bei einer Taktfrequenz von 66 MHz und den vorher beschriebenen Einschränkungen durchaus respektabel erscheint.

Das Konzept der TPU erhöht die Effizienz deutlich, da es eine „fixed function“ Einheit ist, die ihre feste Aufgabe unabhängig von anderen Operationen, die parallel in der RPU stattfinden können durchführen kann.

Im Vergleich mit dem SaarCOR-Prozessor, an dessen Entwicklung die Autoren des Papers zur RPU ebenfalls beteiligt waren, werden nur 50% dessen Leistung erreicht. Der SaarCOR wurde hierfür auf die selbe Taktung heruntergeregelt, die die RPU bietet. Er ist allerdings ein „fixed function” Prozessor und unterliegt somit Beschränkungen was Komplexität und Leistungsverlust angeht, die die RPU, durch ihre Programmierbarkeit, nicht mehr aufweist. Dieser Effizienzverlust wird mit 20 - 50% angegeben und durch die erweiterten Fähigkeiten der RPU gerechtfertigt. Diese Schätzung des Overheads folgt aus demselben Leistungsvergleich zwischen SaarCOR und RPU, den die Autoren interpretiert haben.

Bezüglich der momentanen Leistung sei laut der Autoren eine theoretische Steigerung bis zum 27-fachen, gemessen an der, aktueller GPUs, möglich. Da diese Zahl nur durch Multiplikation der Verhältnisse, der Anzahl von Shading Units und der Taktfrequenzen von RPU und typischer GPU, entstanden ist, muss diese Aussage als spekulativ gewertet werden. Allerdings wurde bei dieser Schätzung weder die Tatsache, dass die Fragment Shader Units der GPUs weit komplexer sind als die, der vorgestellten RPU, noch die größere Speicherbandbreite und Speicherzugriffsoptimierungen bei aktuellen GPUs berücksichtigt, was den Faktor der Leistungssteigerung wieder etwas realistischer aussehen lässt.

5.2 Schlussfolgerung und Ausblick

Aus den vorgestellten Eigenschaften lässt sich durchaus der Schluss ziehen, dass die RPU ein erster Schritt zu einem kostengünstigen, hardwarebasierten Raytracing-System darstellt. Das Paper zeigt, dass es schon mit aktueller Chiptechnologie möglich ist, Echtzeit-Raytracing mit vergleichsweise niedrigem Hardwareaufwand zu betreiben. Dennoch sind auf dem Weg zu einer serienreifen Unterstützung für Raytracing-Nutzer noch einige Stolpersteine zu beseitigen.

Ein großes Thema für zukünftige Verbesserungen ist die Indexstruktur. Gerade bei dynamischen Szenen macht sich der Aufwand für den Neuaufbau der Struktur negativ bemerkbar. Zusätzlich muss im vorliegenden Fall der Indexbaum softwareseitig vorbereitet werden. Von einem Raytracing-Prozessor sollte man erwarten können, dass auch diese Aufgabe von der Hardware übernommen wird.

Effekte und Bildoptimierungen wie Glanzreflexionen und Anti-Aliasing haben die Autoren noch nicht in den Griff bekommen und deren effiziente Berechnung stellt ein weiteres Ziel der Entwicklergruppe dar. Hier sollte besonderes Augenmerk auf mögliche Optimierungen bei Berechnungen von Sekundärstrahlen gelten.

Ausserdem wollen Woop et al. eine weitergehende Nutzbarkeit des Prozessors auszuloten. Als mögliche Anwendungsbereiche werden dabei allgemeine Sichtbarkeitstests, Kollisionserkennung und Nicht-optische globale Transportprobleme genannt.

Literatur

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. 2004. Realtime ray tracing of dynamic scenes on an fpga chip. In *Proceedings of Graphics Hardware*.

SCHMITTLER, J. 2006. *SaarCOR - A Hardware-Architecture for Realtime Ray Tracing (Preliminary)*. PhD thesis, Saarland Universität, Saarbrücken.

WOOP, S., SCHMITTLER, J., AND SLUSALLEK, P. 2005. *RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing*. SIGGRAPH.