GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Interaktives GPU Raytracing mit einem k-d-Baum

SEMINARARBEIT

zum Seminar „Computergrafik“

von

Mathias Steinbach

Matrikelnummer: 20851666

Wintersemester (09/10)

Bearbeitungszeitraum: 20. Oktober 2009 – 31. März 2010
Leitung: Prof. Dr. Winfried Kurth
Dipl. Inf. Reinhard Hemmerling

Vorwort

Foley et al. [1] haben 2005 einen Algorithmus vorgestellt, der es möglich mach k-d-Bäume auf der GPU zu traversiert. 2007 haben Horn et al. [2] auf der I3D Verbesserungen für diesen Algorithmus präsentiert. Sie haben gezeigt, dass damit Raytracing in Echtzeit auf der GPU möglich ist.

Die vorliegende Seminararbeit stellt eine Aufarbeitung der wissenschaftlichen Papers von Foley et al. und Horn et al. dar. Das Ziel ist es, Fortschritte im Bereich Beschleunigungsstrukturen für GPU Raytracing anschaulich und verständlich darzustellen.

Inhaltsverzeichnis

Vorwort II

Inhaltsverzeichnis III

Kapitel 1 Einleitung 1

Kapitel 2 Grundlagen 3

2.1. Raytracing 3

2.2. k-d-Baum 4

2.3. k-d-Baum Traversierung für Szenen 4

Kapitel 3 k-d-Baum Traversierung auf der GPU 7

3.1. Traversieren ohne Stapelspeicher: *ka-restart* 7

3.2. Optimierungen von *ka-restart* 8

Kapitel 4 Umsetzung 10

Kapitel 5 Auswertung 11

5.1. Push-Down 11

5.2. Short-Stack 11

5.3. Packets 11

5.4. Gesamtleistung 12

5.5. GPU-Auslastung 13

5.6. Vergleich mit CPU und Cell-Prozessor 14

Kapitel 6 Ausblick 15

Kapitel 7 Zusammenfassung 16

Abkürzungen und Formelzeichen 17

Abbildungen, Algorithmen, Diagramme und Tabellen 18

Literaturverzeichnis 19

Anhang A Algorithmen 20

Kapitel 1
Einleitung

Das Erzeugen von Bildern aus virtuellen 3D-Szenen ist Aufgabe der Grafikkarte. Ihr Grafikprozessor (GPU) ist heute in der Lage, selbst aus großen Szenen, mehrere Bilder pro Sekunde (fps) in hoher Auflösung (z. B. Full HD) zu berechnen. Dabei werden die 3D-Szenen im einfachsten Fall nur gerastert. Die Rasterprozessoren der GPU rechnen hierbei aus, welcher Pixel welches Objekt der 3D-Szene repräsentiert. Für eine realistische Darstellung der Szene sind jedoch weitere Verarbeitungsschritte notwendig. Einer der wichtigsten und aufwendigsten berechnet die Auswirkungen von Lichtquellen auf das Bild. Diese Aufgabe wird von den GPU-Shadern erfüllt [3]. Die Anzahl der Shader und ihre Funktionen wurden in den letzten Jahren enorm gesteigert. Die Folge ist, Szenen können in immer höheren Auflösungen immer realistischer dargestellt werden. Die aktuell eingesetzten Algorithmen sind jedoch noch weit davon entfernt, Szenen wirklich fotorealistisch, wie es beispielsweise Abb. 1 tut, wiederzugeben. Der Hauptgrunde hierfür ist die Rechenzeit. Die GPU muss mindestens 15 fps berechnen, damit Bewegungen in einer Szene flüssig dargestellt werden. Physikalische Effekte wie Brechung, Streuung und Reflexion von Licht können in dieser kurzen Zeit nur approximiert werden. Eine möglichst genaue Simulation der genannten Effekte wird beim Raytracing durchgeführt (siehe 2.1).

Abb. 1 (Quelle [4])Darstellung einer 3D-Szene mit dem Raytracing Verfahren

Abb. 1 wurde mit POV-Ray berechnet. Die Rechenzeit für ein einzelnes Bild kann hier je nach Auflösung Minuten, Stunden oder gar Tage betragen. Da Raytracing nicht nur aufwendig ist, sondern auch algorithmisch sehr komplex, wurden Raytracer meist auf der CPU implementiert. Um die Rechenzeit für ein Bild zu verkürzen, können mehrere Recheneinheiten genutzt werden. Dieses Konzept der parallelen Bildberechnung wird bereits von der GPU umgesetzt. Es liegt also nahe das Raytracing auch auf der GPU auszuführen. Möglich wird dies, durch die gestiegene Funktionalität der Shader.

Shader können frei programmiert werden. Anfangs waren nur arithmetische Operationen möglich. Ab der Version 3.0 können Shader sogar Schleifen (also bedingte Sprünge) ausführen [5]. Trotz immer besser werdender Shader können noch längst nicht alle Programmierkonzepte aus dem Bereich der CPU-Programmierung umgesetzt werden. Für das Implementieren von Raytracing auf der GPU müssen bestehende Algorithmen so abgeändert werden, dass die Shadern sie ausführen können. Wichtig hierbei ist, durch diese Anpassung darf die Laufzeit der Algorithmen nicht zunehmen.

Diese Seminar-Arbeit konzentriert sich nur auf eine einzelne Komponente des Raytracing-Verfahrens, der Suche nach dem ersten Objekt, das von einem Lichtstrahl getroffen wird. Um diese Suche möglichst effizient zu gestalten wird ein k-d-Baum eingesetzt. Die Traversierung dieser Beschleunigungsstruktur ist Gegenstand der folgenden Kapitel. Die algorithmische Umsetzung ist jeweils in Anhang A zu finden.

Kapitel 2
Grundlagen

In diesem Kapitel werden die grundlegenden Konzepte des Raytracings und der k-d-Baum Traversierung vorgestellt. Die beiden Technologien werden nur soweit erläutert wie es für das Verständnis der Arbeit notwendig ist.

2.1. Raytracing

Beim Raytracing wird die Ausbreitung von Licht in einer Szene simuliert. Das zugrunde liegende Prinzip ist in Abb. 2 dargestellt. Ausgehend vom Augpunkt wird für jeden Pixel ein Strahl in die Szene gesendet. Je nach Materialeigenschaften, die das getroffene Objekt besitzt, werden neue Strahlen erzeugt. Diese können aufgrund von Brechung, Streuung oder Reflektion entstehen. Lichtquellen erzeugen weitere Strahlen. Diese werden dann nach dem gleichen Prinzip durch die Szene verfolgt. Mit Hilfe von stochastischen Berechnungen und den Schnittpunkten der Objekte können Farbwerte für die Pixel berechnet werden [6].

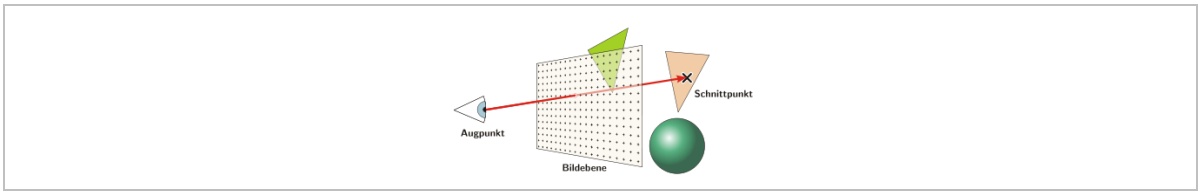

Abb. 2 (Quelle: [6]) Die Idee beim Raytracing: Vom Augpunkt aus wird ein Strahl durch einen Pixel der Bildebene in die 3D-Szene gesendet. Der Pixel wird mit der Farbe des Objektes eingefärbt, das vom Strahl zuerst geschnitten wird.

Im Folgenden sollen drei einfache Raytracing Verfahren vorgestellt werden:

a) Im einfachsten Fall wird nur der Schnittpunkt von einem Strahl mit einen Objekt bestimmt. Auf das Erzeugen von neuen Strahlen wird verzichtet. In diesem Fall wird nur bestimmt, welche Objekte sichtbar sind und welche verdeckt. Die berechneten Strahlen werden auch primäre Strahlen genannt.

b) Um Schatten darzustellen, können Schatten-Strahlen benutzt werden. Hierbei werden von allen Lichtquellen Strahlen in Richtung der in a) berechneten Schnittpunkte gesendet. Wenn ein Lichtstrahl einen Punkt trifft, d. h. der Punkt liegt nicht im Schatten anderer Objekte, wird die Helligkeit des Punktes in Abhängigkeit vom Einfallswinkel des Lichtstrahls berechnet. Je spitzer der Winkel desto heller der Punkt.

c) Ausgehend von den in a) berechneten Schnittpunkten wird je ein neuer Strahl nach dem Reflexionsgesetz erzeugt. Durch dieses Verfahren können Spiegelungen dargestellt werden.

2.2. k-d-Baum

Der k-dimensionale Baum ist im Allgemeinen ein unbalancierter Suchbaum. Er gilt als die beste allgemeine Beschleunigungsstruktur für CPU-Raytracer [1]. Abb. 3 stellt einen 2-dimensionalen Baum dar, der eine 2D-Szene speichert. Die Szene wird im ersten Schritt in zwei Teile geteilt. Die Teilung erfolgt senkrecht zur ersten Koordinatenachse. In der Computergrafik wird meist so geteilt, dass in beiden Teilen möglichst gleichviele Objekte liegen. In anderen Anwendungsbereichen kann das Teilungskriterium variieren.

Im nächsten Schritt werden die zwei zuvor neu entstanden Teile weiter geteilt. Die Teilung erfolgt senkrecht zur zweiten Koordinatenachse. Dieser Vorgang wird bis zur k-ten Koordinatenachse wiederholt. Nach der k-ten wird wieder mit der ersten Koordinatenachse begonnen. Die neu entstandenen Gebiete werden solange weiter geteilt, bis entweder keine Objekte mehr darin enthalten sind oder eine maximal zulässige Anzahl von Objekten erreicht ist [7].

Somit sind die Objekte in den Blättern des Baumes zu finden. Die restlichen Knoten speichern den Abstand der Schnittebene vom Koordinatenursprung.

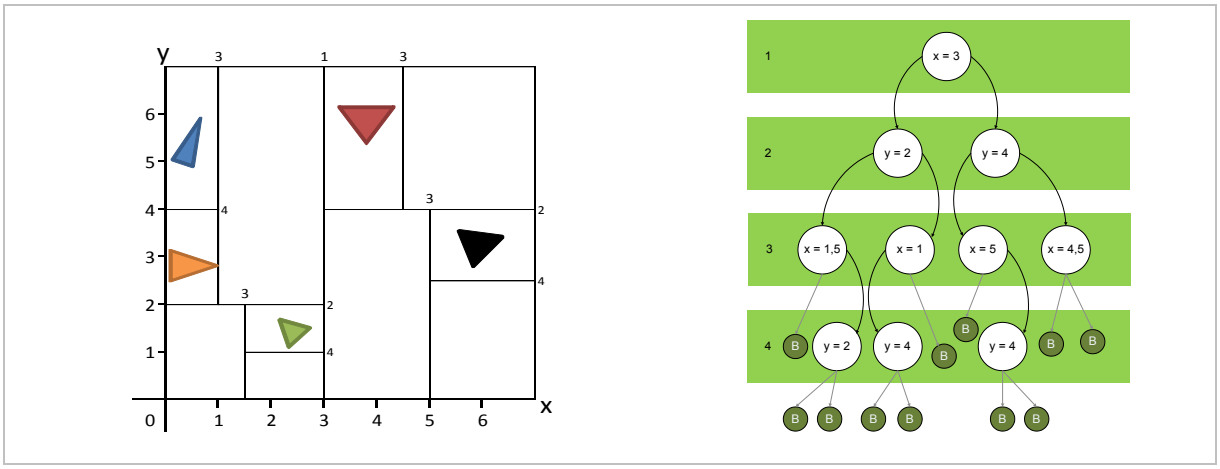

Abb. 3 Eine 2D-Szene, bestehend aus 5 Dreiecken, in einen 2D-Baum einsortiert. Links die geometrische Darstellung, rechts als Baum. Die Blätter beinhalten entweder Szenen-Objekte oder sind leer. Alle anderen Knoten speichern den Abstand der Schnittebene zum Koordinatenursprung.

2.3. k-d-Baum Traversierung für Szenen

Beim Raytracing muss für jeden Strahl überprüft werden, welches Objekt (in der Praxis meist Dreiecke) er zuerst schneidet. Große Szenen können aus mehreren hunderttausend Objekten bestehen. Sind diese Objekte unsortiert, müsste jeder Strahl mit jedem Dreieck auf einen möglichen Schnittpunkt getestet werden. Der Aufwand für das Auffinden des Objekts, welches von einem Strahl zuerst geschnitten wird, läge bei $O(n_0)$. n_0 sei hierbei die Anzahl der Objekte. Durch die Verwendung eines k-d-Baums kann dieser Aufwand auf $O(\log n_0)$ reduziert werden.

Die Suche im k-d-Baum beginnt bei der Wurzel. Hierfür müssen noch einige Informationen bereitgestellt werden:

- Ursprung $\vec{u}$ und Richtung $\vec{r}$ (normiert) des Strahls
- Die Ausdehnung der Szene

Mit Formel (2.1) können weitere Informationen ermittelt werden:

- Die Strecke vom Ursprung bis zum Eintritt in die Szene: s_{min}
- Die Strecke vom Ursprung bis zum Verlassen der Szene: s_{max}

Für die Navigation im Baum muss noch die Strecke vom Ursprung bis zur Schnittebene (s_E) bestimmt werden. Abb. 4 stellt die benötigen Informationen noch einmal grafisch dar.

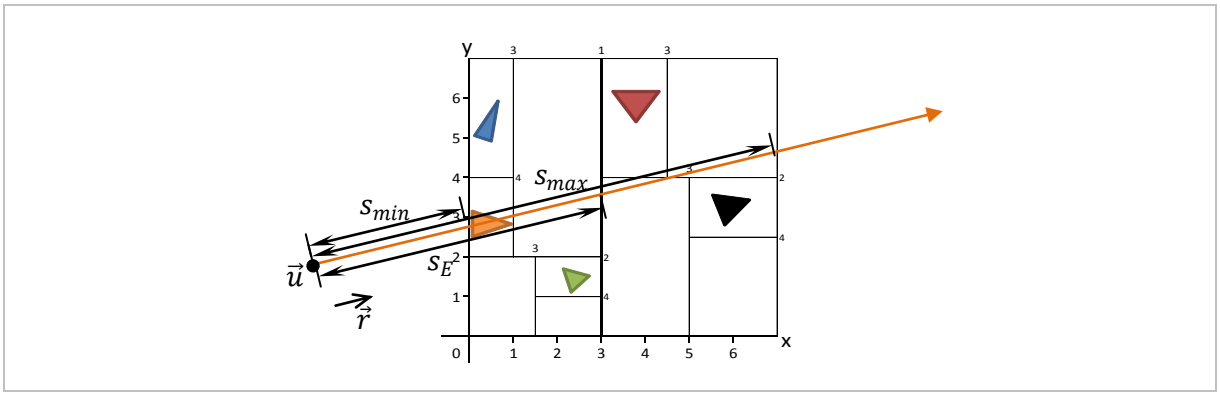

Abb. 4 Darstellung der benötigten Information für die Schnittpunktsuche mit Hilfe eines k-d-Baums

Die Länge von s_E kann nach den Strahlensatz mit folgender Gleichung sehr schnell bestimmt werden:

$$s_E = \frac{E - \vec{u}_i}{\vec{r}_i} \tag{2.1}$$

E ist hierbei die Position der Schnittebene. Dieser Wert ist in den Knoten des k-d-Baums gespeichert. Der Index i ist die Nummer der Koordinatenachse, auf der die Schnittebene senkrecht steht.

Mit der Strecke s_E kann nun bestimmt werden, in welchen Zweigen weitergesucht werden muss. Das Ergebnis der Suche soll das Objekt sein, welches der Strahl zuerst schneidet. Dazu muss bestimmt werden, welcher Zweig des Baums zuerst vom Stahl durchlaufen wird. Das geschieht indem die i-te Komponente von $\vec{r}$ ausgewertet wird. Je nachdem wie der Baum aufgebaut wurde, kann mit dem Vorzeichen von $\vec{r}_i$ entschieden werden, welcher der Zweige der Nahe und welcher der Ferne ist. Vernachlässigt man den Ursprung des Strahls, so durchquert der Strahl zuerst den nahen und dann den fernen Zweig. Abb. 5 stellt dar, unter welchen Bedingungen im nahen bzw. fernen Zweig weitergesucht werden muss:

Fall a) Der Strahl verlässt die Szene bevor er die ferne Seite erreicht ($s_E \geq s_{max}$). Hier muss nur auf der nahen Seite weitergesucht werden.

Fall b) Der Ursprung liegt auf der fernen Seite. In diesem Fall entfernt sich der Strahl von der Schnittebene und kann somit gar nicht mehr die nahe Seite erreichen. Für $s_E < 0$ muss also nur auf der fernen Seite weitergesucht werden. Gleiches gilt für $s_E \leq s_{min}$. Der Strahl erreicht zuerst die Schnittebene und dann erst die Szene.

Fall c) Der Strahl erreicht zuerst die Szene oder befindet sich schon in ihr. Als zweites erreicht er die Schnittebene und dann erst verlässt er die Szene. Es gilt $s_{min} < s_E < s_{max}$. In diesem Fall muss in beiden Zweigen weitergesucht werden, zuerst im Nahen. Nur wenn die Suche im nahen Zweig erfolglos war, muss sie im fernen Zweig fortgesetzt werden.

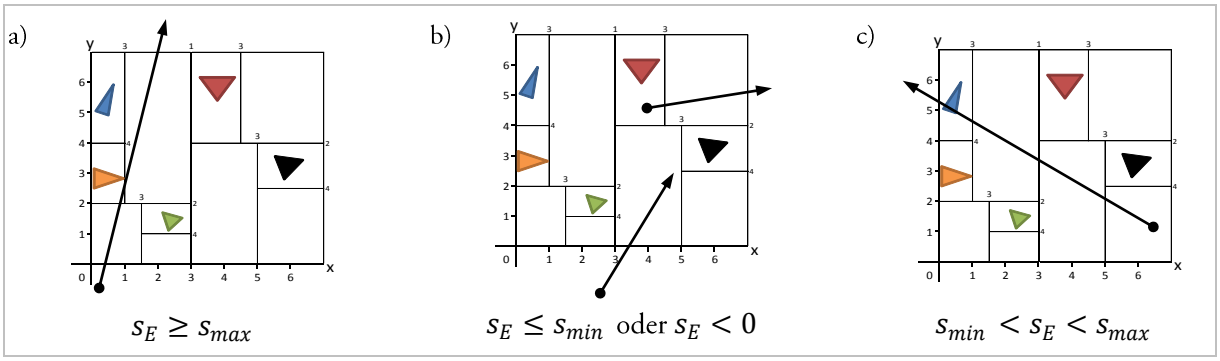

Abb. 5 Unterschiedliche Strahlen durchqueren dieselbe Szene. Ausgehend von der Wurzel, muss im Fall a) nur im nahen Zweig, im Fall b) nur im fernen Zweig und im Fall c) in beiden Zweigen weitergesucht werden.

Im Fall c muss die Szene aufgeteilt werden. Dieses „Aufteilen“ wird erreicht, indem s_{min} und s_{max} verändert werden. Für die Suche auf der nahen Seite wird $s_{max} = s_E$ gesetzt, für die Suche auf der fernen Seite $s_{min} = s_E$.

Im Fall a und b bleiben s_{min} und s_{max} unverändert. Mit diesen neuen Startwerten wird nun, ausgehend von den Kindern des gerade untersuchten Knotens, das Verfahren solange wiederholt bis der gerade untersuchte Knoten ein Blatt ist.

In den Blättern sind die Szenenobjekte gespeichert. Hier findet nun die Schnittpunktberechnung statt. Gibt es keinen Schnittpunkt, muss im fernen Zweig des Knotens weitergesucht werden, bei dem zuletzt Fall 3 eingetreten ist.

Algorithmus 1 ist eine iterative Implementierung für das Traversieren von k-d-Bäumen. Auf dem Stapelspeicher (Stack) liegt immer jener Knoten ganz oben, bei dem zuletzt Fall c eingetreten ist.

Kapitel 3
k-d-Baum Traversierung auf der GPU

Mit der GPU ist es möglich Algorithmus 1 für hunderte Strahlen gleichzeitig auszuführen. Das Problem hierbei ist, dass jede Instanz von Algorithmus 1 einen eigenen Stapelspeicher benötigt. Moderne GPUs besitzt zwar einen sehr großen globalen Speicher (512 MB und mehr), jedoch nur einen kleinen Zwischenspeicher, auf den sehr schnell zugegriffen werden kann. Ein kompletter Stapelspeicher für jede Instanz von Algorithmus 1 passt nicht in den Zwischenspeicher und die Zugriffszeiten auf den globalen Speicher sind zu lang. 2005 haben Foley et al. [1] eine Variante von Algorithmus 1 vorgestellt, die ohne Stapelspeicher auskommt. 2007 haben Horn et al. [2] auf der I3D Verbesserungen für diesen Algorithmus präsentiert. Dieses Kapitel stellt die Änderungen von Foley et al. und Horn et al. vor.

3.1. Traversieren ohne Stapelspeicher: *kd-restart*

Der Stapelspeicher in Algorithmus 1 wird nur für den Fall benötigt, das in beiden Kindern eines Knotens weitergesucht werden muss. Tritt dieser Fall ein wird $s_{max} = s_E$ gesetzt, d. h. s_{max} wird kleiner. Abb. 6 stellt diesen Sachverhalt grafisch dar.

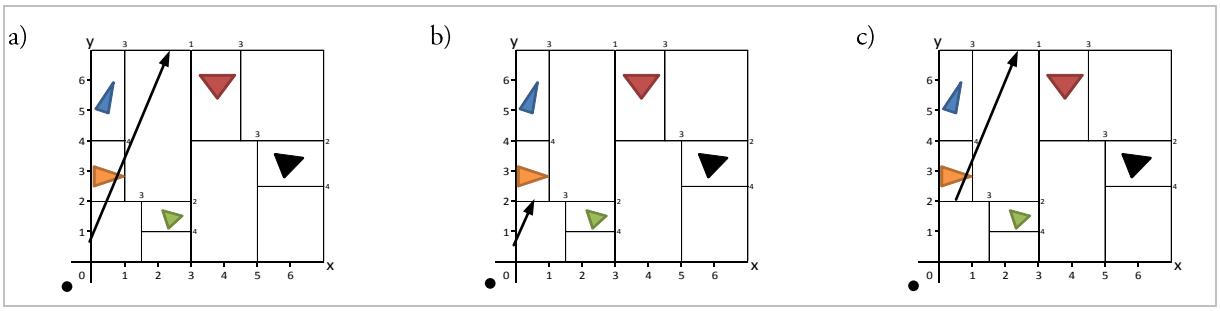

Abb. 6 Dargestellt ist jeweils die Distanz $s_{max} - s_{min}$. a) stellt die Ausgangssituation vor den Start von Algorithmus 1 dar, b) die Situation nachdem der Algorithmus erstmals ein Blatt besucht hat. In c) ist die Situation dargestellt, nachdem erstmals Daten vom Stapelspeicher genommen wurden.

Foley et al. verzichten nun auf die Stapelspeicher-Operationen. Sie starten den Algorithmus einfach neu. Dabei wird s_{min} auf das veränderte s_{max} gesetzt, und s_{max} wird auf seinen ursprünglichen Wert gesetzt. Die Anzahl der besuchten Blätter verändert sich auf diese Weise nicht, jedoch erhöht sich die Anzahl der besuchten Knoten enorm. Falls s_{max}, nachdem ein Blatt durchsucht wurde, immer noch seinen ursprünglichen Wert hat, ist die komplette Szene durchsucht und es wurde nichts gefunden. In Algorithmus 2 sind die Änderungen von Foley et al. hervorgehoben. Der sogenannte „neustart“ wird jedoch nicht rekursiv implementiert, sondern nur durch eine while-Schleife simuliert.

3.2. Optimierungen von *kd-restart*

Der Stapelspeicher in Algorithmus 1 sorgt dafür, dass die Suche direkt in einem fernen Zweig fortgesetzt werden kann. Der sogenannte Neustart erfolgt hier nicht bei der Wurzel sondern beim gespeicherten Knoten. kd-restart kennt diesen Knoten nicht und muss bei der Wurzel beginnen. Je tiefer ein Baum ist, umso mehr büßt kd-restart an Geschwindigkeit gegenüber Algorithmus 1 ein. Horn et al. [2] haben sich diesem Problem angenommen und Verbesserungen für kd-restart entwickelt. Unter den folgenden drei Punkten werden diese Verbesserungen vorgestellt.

3.2.1. Push-Down

Nach jedem Neustart durchläuft kd-restart dieselben Knoten des Baums. Das geschieht solange, bis kd-restart den Knoten erreicht, bei dem er erstmals im nahen und im fernen Zweig weitersuchen muss. Damit die Knoten auf den oberen Ebenen des k-d-Baums möglichst wenig durchlaufen werden, haben Horn et al. den Neustart auf genau diesen Knoten verschoben (Push-Down). Die notwendigen Änderungen an kd-restart sind in Algorithmus 3 realisiert. Die Wurzel wird solang auf den aktuellen Knoten gesetzt, bis auf beiden Seiten weitergesucht werden muss.

3.2.2. Short-Stack

Im Folgenden soll der zwar kleine aber sehr schnelle Speicher der GPU genutzt werden. Ein kompletter Stapelspeicher kann nicht umgesetzt werden, jedoch reicht der Speicher für einen kurzen Stapelspeicher (Short-Stack) mit fester Kapazität.

Der Short-Stack verhält sich dabei folgendermaßen: Wird ein Datensatz auf einen vollen Short-Stack gelegt, so fällt der unterste Datensatz heraus und kann somit nicht mehr gelesen werden. Deshalb kann es passieren, dass der Algorithmus Daten von einen leeren Short-Stack nehmen will. In diesem Fall wird kd-restart so ausgeführt als gäbe es keinen Short-Stack. Der Algorithmus startet wieder bei der Wurzel bzw. bei den Knoten, den die Push-Down Strategie zur Wurzel gemacht hat.

Der Short-Stack verhindert den Neustart, wenn auf den unteren Ebenen des Baums in beiden Kindern weitergesucht werden muss. Die Umsetzung von Push-Down und Short-Stack zeigt Algorithmus 4.

3.2.3. Packets

Packets werden auch auf anderen Plattformen eingesetzt, um die Laufzeit beim Raytracing zu verkürzen. Die Idee dahinter ist, dass fast parallele Strahlen, die denselben Ursprung haben, auch fast identische Pfade durch den k-d-Baum nehmen. Dieses Verhalten ermöglicht es, ein ganzes Bündel von Strahlen auf einmal abzuarbeiten. Während alle Strahlen eines Bündels in der Nähe der Wurzel noch dieselben Knoten besuchen, nimmt die Wahrscheinlichkeit, dass Strahlen unterschiedliche Knoten besuchen, in Richtung der Blätter stark zu. Diese kann unter Umständen auch zu einen Mehraufwand führen, der das Raytracing nicht beschleunigt sondern sogar bremst, siehe hierzu Kapitel 5.

Die Verarbeitung der Strahlenbündel erfolgt mit Vektoren (Horn et al. bezeichnen sie als Packets). Für ihre Verwendung spricht die Hardware Architektur der GPU. Der Zugriff auf den Knoten eines Baums ist sehr aufwendig. Mit Packets muss für alle Strahlen in einem Bündel nur einmal ein Knoten gelesen wer-

den, ohne Packets muss der Algorithmus für jeden Strahl jeden Knoten neu lesen. Weiterhin sind Operation auf Vektordaten durch die Hardware beschleunig (SIMD).

Wie oben erwähnt kann es vorkommen, dass die Strahlen eines Bündels unterschiedliche Suchpfade durchschreiten. In diesem Fall müssen die Strahlen, die nicht im aktuellen Suchpfad sind, ausgeblendet werden. Das geschieht durch eine Maske. Sie protokolliert, welche Strahlen den aktuellen Knoten durchlaufen.

Alle vorgestellten Verbesserungen von Horn et al. sind in Algorithmus 5 umgesetzt.

Kapitel 4
Umsetzung

Horn et al. haben nicht nur die k-d-Baum-Traversierung sondern sogar einen kompletten Raytracer auf der GPU umgesetzt. Der Raytracer ist in der Lage Schattenstrahlen und reflektierende Strahlen (siehe hierzu Kapitel 2 - Raytracing) zu berechnen. Die Implementierung erfolgte auf einer X1900 XTX Grafikkarte. Die Prozessoren der GPU aus dem Hause ATI arbeiten mit einen 650 MHz Tackt und können im Idealfall rund 31.2 Milliarden Maschinenbefehle pro Sekunde ausführen. Dabei kann jeder Befehl gleichzeitig auf vier unterschiedliche Daten ausgeführt werden (SIMD der Breite 4). Der Speicher ist mit 750 MHz getacktet. Die X1900 XTX verfügt über Pixel Shader 3.0. Ihre Programmierung erfolgte mit Brook, Direct3D 9.0c und Shader-Maschinenbefehlen.

Die Packets wurden, bedingt durch die GPU, so implementiert, dass ein Thread vier Strahlen berechnet. Für mehr Strahlen pro Packet reichten die Register nicht aus. Der Short-Stack kann drei Datensätze fassen.

Der k-d-Baum wurde nach der Oberflächen Heuristik von Havran und Bittner [11] erzeugt. Dabei haben Horn et al. die Baumhöhe an das Verhältnis zwischen Aufwand für die Schnittpunktberechnung und Aufwand für die Knoten-Traversierung angepasst.

Horn et al. nutzen die Rasterungsfunktion der GPU um erste Schnittpunkte der Strahlen, die vom Augpunkt ausgehen, auszurechnen. Der Tiefen-Puffer definiert dabei die z-Koordinate. Bei der Berechnung von Schattenstrahlen werden Schnittpunkte, die nicht im Schatten liegen, mit dem Phong Shading Algorithmus eingefärbt.

Kapitel 5
Auswertung

Kapitel 5 analysiert die Auswirkungen der im Kapitel 3 beschriebenen Verbesserungen. Außerdem wird Untersucht wie sich der GPU Raytracer im Vergleich mit anderen Hardware-Plattformen schlägt und wo der Flaschenhals der GPU liegt.

5.1. Push-Down

In Abb. 5 auf Seite 6 ist zu erkennen, dass immer wenn ein Strahl nur auf einer Seite einer Schnittebene verläuft, er mit hoher Wahrscheinlichkeit die darunterliegende Schnittebene schneidet. Dies führ dazu, dass die Wurzel nur ein kleines Stück nach unten verschoben wird.

Der Vorteil von Push-Down besteht darin, dass die obersten Knoten nicht doppelt durchlaufen werden. Auf diese Knoten greifen, durch die zeitgleiche Ausführung, viele Instanzen von kd-restart zu. Unterschiedliche Threads können jedoch nicht immer auf gleiche Datensätze zugreifen. Die Folge sind Konflikte. Durch Push-Down wird die Wahrscheinlichkeit für diese Konflikte reduziert.

Ein Nachteil ist, dass in jeder Iteration eine Bedingung mehr geprüft werden muss. Das führt dazu, dass Push-Down die Laufzeit von kd-restart nur geringfügig verbessert. Tabelle 1 zeigt, dass je komplexer die Szene – also je tiefer der Baum – umso weniger verbessert Push-Down die Leistung des Raytracers.

5.2. Short-Stack

Horn et al. haben gemessen, dass kd-restart im Durchschnitt 166% mehr Knoten als die CPU-Implementierung besucht. Durch einen Short-Stack der Größe eins konnte der Mehraufwand auf 25% reduziert werden. Ein Short-Stack der Größe drei verringert den Mehraufwand sogar auf 3%. Der Short-Stack(3) benötigt pro Instanz 36 Byte (108 Byte mit Packets). Damit passt er problemlos in den schnellen Speicher der GPU.

Mit Short-Stack(3) steht kd-restart, im Bezug auf den Aufwand, anderen Implementierungen in nichts mehr nach. Der Geschwindigkeitszuwachs beim Raytracing liegt hier bei circa 20% (siehe Tabelle 1)

5.3. Packets

Die Verwendung von Packets bringt nur dann etwas, wenn alle Strahlen in einem Bündel möglichst parallel sind. Sobald für einzelne Strahlen in unterschiedlichen Zweigen gesucht werden muss, entsteht ein enormer Mehraufwand. Diagramm 1 stellt diesen Sachverhalt noch einmal dar. Es ist erkennbar, dass die ersten Schnittpunkte mit Hilfe von Packets deutlich schneller gefunden werden als ohne. Jedoch schon

nach der ersten Reflexion dauert die Berechnung der reflektierten Strahlen mit Packets genau solange wie ohne. Nach der vierten Reflexion dauert die Berechnung mit Packets sogar fast doppelt so lang. Die Ursache hierfür ist, dass nach jeder Reflexion die Strahlen in einem Bündel weiter auseinander driften und somit immer unterschiedlichere Pfade durch den k-d-Baum nehmen.

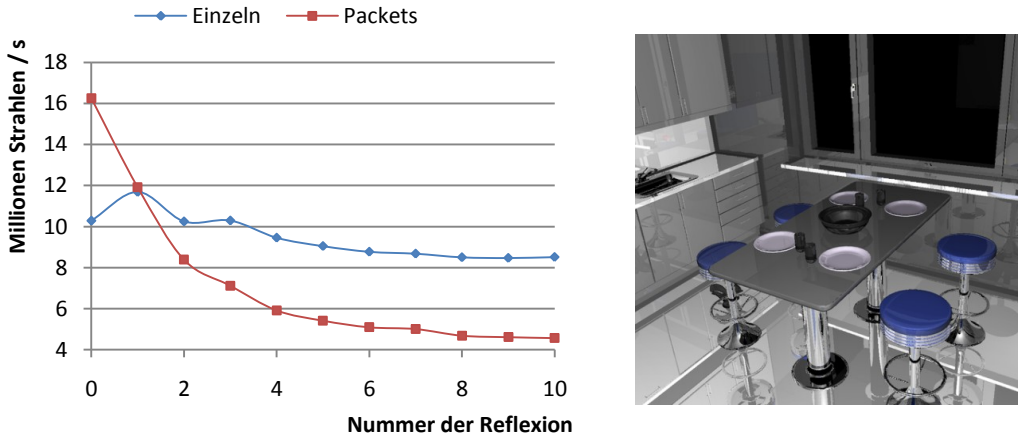

Diagramm 1 Dargestellt ist die Anzahl der Strahlen, die pro Sekunde nach der n-ten Reflexion in der Küchenszene berechnet werden können.

Die Ursache für die geringe Leistungsabnahme von kd-restart ohne Packets liegt an den Speicherzugriffen. Je öfter die Strahlen reflektiert werden, desto ungeordneter ist der Zugriff auf den k-d-Baum. Das hat zur Folge, dass die Cache-Funktion der GPU nicht mehr in dem Maße genutzt wird wie zu Beginn.

Für die Berechnung von primären und Schatten-Strahlen sich Packets jedoch ideal geeignet. Hier befinden sich noch alle Strahlen in einem Lichtkegel und sind somit möglichst parallel. Tabelle 1 stellt den Zuwachs an Leistung dar. Auf anderen Plattformen werden sogar 16 bis 64 Strahlen zu einem Bündel zusammengefast, um so noch schneller zu traversieren.

5.4. Gesamtleistung

Der nicht optimierte kd-restart läuft auf der X1900 XTX rund 20 mal schneller als auf der von Foley et al. verwendeten X800 XT PE. Das liegt zum einen daran, dass die X1900 mehr Rechenleistung hat (rund 3,75 mal mehr) zum anderen aber auch an dem Pixel Shader 3.0. Hier können erstmals auch Schleifen auf den GPU Prozessoren ausgeführt werden. Ein direkter Vergleich mit den von Foley et al. ermittelten Messungen ist somit nicht möglich.

Der optimierte kd-restart kann zwischen 65% und 130% mehr Strahlen pro Sekunde berechnen als der ursprüngliche (beide ausgeführt auf der X1900 XTX). Wie sehr sich die einzelnen Verbesserungen dabei auf die Rechengeschwindigkeit auswirken, stellt Tabelle 1 dar.

			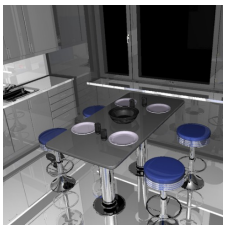		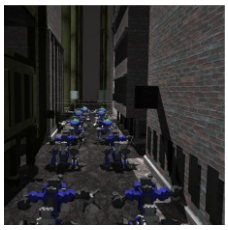		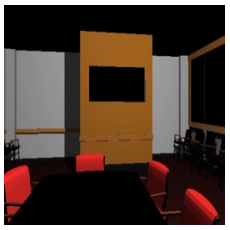	
	PS	SS	PS	SS	PS	SS	PS	SS
kd-restart	38,3	34,8	8,6	17,1	7,7	9,5	9,1	15,2
+ Packets	17,5	35,5	13,3	21,1	14,0	13,5	13,6	15,9
+ Push-Down	88,8	74,7	12,5	21,4	14,7	12,8	13,9	16,1
+ Short-Stack	91,3	121,3	16,3	27,3	17,9	16,2	15,2	18,8

Tabelle 1 (Quelle[2])Die Berechnungsrate in Millionen Strahlen pro Sekunde für primäre Strahlen (PS) und Schatten-Strahlen (SS). Die Zahlen wurden für die vier dargestellten Szenen in einer Auflösung von 1024x1024 Pixel ermittelt. Jede Zeile enthält die Verbesserungen der darüber liegenden Zeilen.

Durch das einschalten der Rasterfunktion der GPU konnten Horn et al. die Gesamtleistung noch weiter steigern.

5.5. GPU-Auslastung

Horn et al. haben die Auslastung der GPU untersucht. Dafür haben sie die maximal mögliche Instruktionsrate der ATI X1900 XTX mit ihrer tatsächlichen verglichen. Mit Packets erreichte der GPU-Raytracer rund 40 % der maximalen Instruktionsrate, ohne diese, rund 70%. Die Ursachen dafür können vielfältig sein.

1. Die Anzahl der Threads könnte zu gering sein, um abhängige Operationen zu überbrücken.
2. Die Anzahl der Threads könnte zu gering sein, um Speicher-Latenzen zu verbergen.
3. Die Bandbreite für das Laden der Daten in den Texturspeicher könnte zu gering sein.
4. Zuviele Thread benutzen unterschiedliche Ausführungspfade.

Punkt eins wird von Horn et al. ausgeschlossen, da es genügend Strahlen, und damit auch Threads gibt. Um 2. und 3. auszuschließen haben die Autoren den Speichertakt um ein Drittel verringert. Auch mit verringertem Speichertakt, ist die Leistung nicht wesentlich eingebrochen.

Bei der X1900 XTX teilen sich 48 Threads je einen Programmzähler. Nur wenn alle Threads dieselben Instruktionen in derselben Reihenfolge ausführen, kann die maximale Instruktionsrate erreicht werden. Ist dies nicht der Fall, müssen Programmzustände zwischengespeichert und Instruktionen doppelt geladen werden. Bei der Suche im k-d-Baum ist es jedoch völlig normal, dass die Ausführungspfade variieren. Schließlich wird in unterschiedlichen Pfaden des Baumes gesucht. Um zu belegen, dass Punkt vier für die verringerte Instruktionsrate verantwortlich ist, haben Horn et al. Änderungen am Assemblercode vorgenommen. Sie haben alle Speicherzugriff und Rechenoperationen durch „MOV“ Befehle ersetzt und gleichzeitig dafür gesorgt, das immer noch der gleiche Ausführungspfad von den Threads genutzt wird. Trotz des Verzichts auf Berechnungen und Speicherzugriffen, hat die Ausführungszeit nicht abgenommen.

5.6. Vergleich mit CPU und Cell-Prozessor

Horn et al. haben ihren GPU-Raytracer mit dem CPU-Raytracer von Wald [8] und dem Cell-Raytracer von Benthin et al. [9] verglichen. Wald benutzt einen Vier-Kern Opteron Prozessor mit 2,4 GHz. Er konnte damit 8,7 Millionen Schnittpunkt pro Sekunde berechnen, Benthin et al. erreichten auf dem Cell-Prozessor 57.2. Die Angaben beziehen sich auf die Berechnung von primären Strahlen mit SIMD Befehlen. Das in 5.3 geschilderte Problem der nicht parallelen Strahlen tritt hier nicht auf.

Obwohl die Instruktionsrate des Cell-Prozessors nur rund 62% der X1900 XTX beträgt, rechnet er etwa viermal schneller als die GPU. Erst durch die Nutzung von weiteren GPU Eigenschaft, wie Rasterung und Schattierung, konnte die Gesamtleisung der GPU mit der von Cell und CPU konkurrieren. Diagramm 2 stellt die Leistung der drei Prozessorarchitekturen gegenüber. Eine der Hauptursachen hierfür ist, das CPU und Cell besser mit den unterschiedlichen Ausführungspfaden der einzelnen Threads umgehen können.

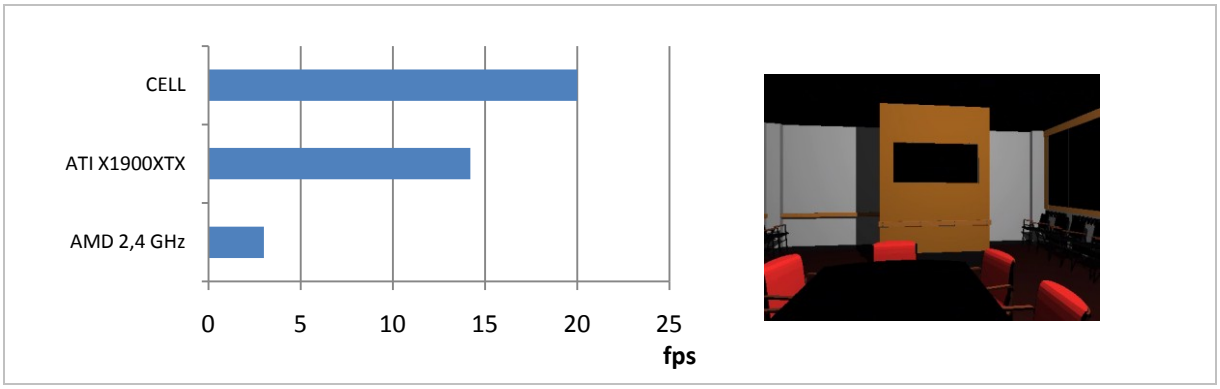

Diagramm 2 Renderzeiten von Cell-Prozessor, CPU und GPU im Vergleich. Die Werte wurden bei der Berechnung der Konferenzraum-Szene ermittelt.

Kapitel 6
Ausblick

Die Leistung von GPU-Raytracern könnte sich laut Horn et al. (2007) noch weiter verbessern, wenn die GPU auch ganzzahlige Datentypen unterstützte (das ist heute schon der Fall) und nicht mehr so empfindlich auf unterschiedliche Ausführungspfade der Threads reagierte (Das ist mit der kommenden Grafikkartengeneration von Nvidia der Fall [10]).

Horn et al. glauben das der Raytracer durch ein besseres Softwareengineering weiter verbessert werden kann. So könnte die Rasterung durch eine Sortierung des Speichers noch schneller werden, Strahlen werden in separaten Schritten erzeugt und berechnet und Schattenstrahlen könnten viel einfacher berechnet werden als andere Strahlen.

Dass Horn et al. recht behalten sollten, zeigt Nvidia. 2008 präsentierte der Chip-Designer auf der Siggraph eine Raytracing Engine. Mit dem Quadro Plex 2100 D4 kann Nvidia 30 fps bei einer Full HD Auflösung berechnen [12]. Abb. 7 zeigt die Testszene, die aus zwei Millionen Polygonen besteht.

Abb. 7 (Quelle [12])Raytracing in 3D-Spielen ist mit aktueller Hardware möglich

Der nächste Schritt besteht für Horn et al. darin, Oberflächen realistischer darzustellen. Dafür wollen sie fortgeschrittene Schattenmodelle einbeziehen, wie z. B. Texturen oder Oberflächenstrukturen (Bump-Maps). Für diese Anwendungen ist die GPU ideal geeignet.

Kapitel 7
Zusammenfassung

Horn et al. konnten zeigen, dass Raytracing auf der GPU mit fortgeschrittenen Implementierungen auf CPU und Cell konkurrieren kann. Ihr GPU-Raytracing-Algorithmus kann mehrere Bilder pro Sekunde berechnen. Somit können Bewegungen der Kamera in der Szene flüssig dargestellt werden. Bewegungen der Szenen-Objekte werden jedoch noch nicht berücksichtigt, hierfür müsste der k-d-Baum für jedes Bild neu erzeugt werden.

Ein kurzer Stapelspeicher (Short-Stack) für nur 3 Datensätze, reicht aus, um den kd-restart Algorithmus von Foley und Sugermann zu verbessern. Die Traversierung auf der GPU ist damit nicht wesentlich aufwendiger als auf anderen Plattformen, die einen kompletten Stapelspeicher umsetzen können.

Primäre Strahlen und Schatten-Strahlen können in Bündeln von vier Strahlen schneller berechnet werden. Aufgrund der begrenzten Register konnten Horn et al. jedoch nicht mehr Strahlen zu einem Bündel zusammenfassen.

Trotz der Verbesserungen von Horn et al. am kd-restart Algorithmus, ist die enorme Geschwindigkeitssteigerung gegenüber Foley et al. auf Innovationen in der Hardwarearchitektur zurückzuführen. Nvidia zeigt, dass die Leistung von GPU-Raytracern auf aktueller Hardware noch einmal stark gesteigert werden konnte [12].

Dem größten Problem, dem Horn et al. gegenüberstanden, ist die Divergenz der Threads. Durch die Suche im Baum durchlaufen die meisten Threads unterschiedliche Ausführungspfade. Die Architektur der GPU reagiert hierauf, anders als z. B. die CPU, mit einem Verlust an Rechenleistung. Gleichzeitig zeigt dieses Problem, dass GPU-Raytracing nicht durch die Speicherbandbreite gebremst wird.

Abkürzungen und Formelzeichen

Abkürzungen

CPU	Central Prozessing Unit
fps	Frames per Seconds
Full HD	Full High Definition (1920x1080 Pixle)
GPU	Graphics Processing Unit
k-d	k-dimensional
SIMD	Single Instruction, Multiple Data

Formelzeichen

E	Abstand der Schnittebene zum Koordinatenursprung
i	Nummer der Koordinatenachse
n_O	Anzahl der Objekte im k-d-Baum
$\vec{r}$	Richtung eines Strahls
s_E	Strecke vom Ursprung bis zur Schnittebene
s_{max}	Strecke vom Ursprung bis zum Verlassen der Szene
s_{min}	Strecke vom Ursprung bis zum Eintritt in die Szene
$\vec{u}$	Ursprung eines Strahls

Abbildungen, Algorithmen, Diagramme und Tabellen

Abb. 1 (Quelle [4])Darstellung einer 3D-Szene mit dem Raytracing Verfahren 1
Abb. 2 (Quelle: [6]) Die Idee beim Raytracing: Vom Augpunkt aus wird ein Strahl durch einen Pixel der Bildebene in die 3D-Szene gesendet. Der Pixel wird mit der Farbe des Objektes eingefärbt, das vom Strahl zuerst geschnitten wird. 3
Abb. 3 Eine 2D-Szene, bestehend aus 5 Dreiecken, in einen 2D-Baum einsortiert. Links die geometrische Darstellung, rechts als Baum. Die Blätter beinhalten entweder Szenen-Objekte oder sind leer. Alle anderen Knoten speichern den Abstand der Schnittebene zum Koordinatenursprung. 4
Abb. 4 Darstellung der benötigten Information für die Schnittpunktsuche mit Hilfe eines k-d-Baums 5
Abb. 5 Unterschiedliche Strahlen durchqueren dieselbe Szene. Ausgehend von der Wurzel, muss im Fall a) nur im nahen Zweig weiter gesucht werden, im Fall b) nur im fernen Zweig und im Fall c) in beiden Zweigen weitergesucht werden. 6
Abb. 6 Dargestellt ist jeweils die Distanz $smax - smin$. a) stellt die Ausgangssituation vor den Start von Algorithmus 1 dar, b) die Situation nachdem der Algorithmus erstmals ein Blatt besucht hat. In c) ist die Situation dargestellt, nachdem erstmals Daten vom Stapelspeicher genommen wurden 7
Abb. 7 (Quelle [12])Raytracing in 3D-Spielen ist mit aktueller Hardware möglich 15

Algorithmus 1 Iterative Umsetzung der Traversierung von k-d-Bäumen mit Hilfe eines Stapelspeichers 20
Algorithmus 2 (kd-restart nach Foley et al. [1]) Iterative Umsetzung der Traversierung von k-d-Bäumen ohne einen Stapelspeicher 21
Algorithmus 3 kd-restart mit Push-Down (nach Horn et al. [2]) 22
Algorithmus 4 kd-restart mit Push-Down und Short-Stack (nach Horn et al. [2]) 23
Algorithmus 5 kd-restart mit Push-Down und Short-Stack und Packets (nach Horn et al. [2]). Alle Variablen mit Index v sind Vektoren bei denn von SIMD-Funktionen Gebrauch gemacht wird. 24

Diagramm 1 Dargestellt ist die Anzahl der Strahlen, die pro Sekunde nach der n-ten Reflexion in der Küchenszene berechnet werden können 12
Diagramm 2 Renderzeiten von Cell-Prozessor, CPU und GPU im Vergleich. Die Werte wurden bei der Berechnung der Konferenzraum-Szene ermittelt. 14

Tabelle 1 (Quelle[2])Die Berechnungsrate in Millionen Strahlen pro Sekunde für primäre Strahlen (PS) und Schatten-Strahlen (SS). Die Zahlen wurden für die vier dargestellten Szenen in einer Auflösung von 1024x1024 Pixel ermittelt. Jede Zeile enthält die Verbesserungen der darüber liegenden Zeilen. 13

Literaturverzeichnis

[1] Foley, Tim; Sugerman, Jeremy: *Kd-Tree Acceleration Structures for a GPU Raytracer*. In: *Graphics Hardware 2005*, Los Angeles : ACM Press, 2005. S. 16-22.

[2] Horn, Daniel Reiter; Sugerman, Jeremy; Houston, Mike; Hanrahan: *Interactive k-d Tree GPU Raytracing*. In: *I3D 2007*, Seattle : ACM Press, 2007. S. 167-174.

[3] Seite „*Shader*“. In: *Wikipedia, Die freie Enzyklopädie*. Bearbeitungsstand: 1. Januar 2010, 17:36 UTC. URL: http://de.wikipedia.org/w/index.php?title=Shader&oldid=68689797 (Abgerufen: 17. Januar 2010, 21:10 UTC)

[4] Crowell, Benjamin: *Optics*. Californian : Light and Matter, 2009. - ISBN 0-9704670-5-2 S. 23

[5] Spooren, Florian: *Test: Shader Model 3.0 vs. 2.0.* In *Hardware-Info* URL: http://www.hardware-infos.com/tests.php?test=11 (Aufruf am 2.01.2010)

[6] Seite „*Raytracing*“. In: *Wikipedia, Die freie Enzyklopädie*. Bearbeitungsstand: 5. Dezember 2009, 18:48 UTC. URL: http://de.wikipedia.org/w/index.php?title=Raytracing&oldid=67647778 (Abgerufen: 9. Dezember 2009, 17:27 UTC)

[7] Seite „*K-d-Baum*“. In: *Wikipedia, Die freie Enzyklopädie*. Bearbeitungsstand: 20. November 2009, 13:45 UTC. URL: http://de.wikipedia.org/w/index.php?title=K-d-Baum&oldid=67046805 (Abgerufen: 9. Dezember 2009, 18:03 UTC)

[8] Wald, I.: *Realtime Ray Tracing and Interactive Global Illumination.* 2004. PhD thesis, Saarland University.

[9] Benthin, C.; Wald, I.; Scherbaum, M.; Friedrich, H.: *Ray Tracing on the CELL Processor.* In: Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing. 2006

[10] Farrar ,Timothy: *CUDA 3.0 and GT300 and Future Predictions.* URL: http://farrarfocus.blogspot.com/2009/05/cuda-30-and-gt300-and-future.html (Aufruf am 13.01.2010)

[11] Havran, V.; Bittner, J.: *On improving kd-trees for ray shooting*. In: *Proceedings of WSCG'2002 conference*, 2002. S. 209-217.

[12] Valich, Theo: *Nvidia demonstrates interactive real-time ray-tracer*. In: *TGDaily* (18. August 2008). URL: http://www.tgdaily.com/software-features/38927-nvidia-demonstrates-interactive-real-time-ray-tracer (Abruf am 9. Dezember 2009).

Anhang A Algorithmen

```
kdTravsieren( strahl, wurzel, sMin, sMax )
{
    stack.push( wurzel, sMin, sMax)
    while( stack.notEmpty() )
    {
        [knoten, sMin, sMax] = stack.pop()

        while( knoten.istKeinBlatt() )
        {
            i  = knoten.raumRichtung
            sE = ( knoten.wert – strahl.Ursprung[i] ) / strahl.richtung[i]

            [knotenNah, knotenFern] = ordne( strahl.richtung[i], knoten.links, knoten.rechts )

            //Auswerten:
            if( sE >= sMax )
                knoten = knotenNah
            else
            {   if( sE <= sMin  or  sE < 0)
                    knoten = knotenFern
                else
                {   Stack.push( knotenFern, sE, sMax )
                    sMax   = sE
                    knoten = knotenNah
                }
            }
        }

        //Suche nach Schnittpunkten im Blatt
        for( objekt : knoten.Objekte() )
            if ( schneidet( strahl, objekt ) )
                return objekt

    }
    return NULL
}
```

Algorithmus 1 Iterative Umsetzung der Traversierung von k-d-Bäumen mit Hilfe eines Stapelspeichers

```
kdRestart( strahl, wurzel, sMin, sMax )
{
    stack.push( wurzel, sMin, sMax)
    while( stack.notEmpty() )
    sMaxAlt = sMax
    sMax    = sMin

    while( sMax < sMaxAlt ) //Diese Schleife simuliert den Neustart
    {
        [knoten, sMin, sMax] = stack.pop()
        knoten = wurzel
        sMin   = sMax
        sMax   = sMaxAlt

        while( knoten.istKeinBlatt() )
        {
            i  = knoten.raumRichtung
            sE = ( knoten.wert - strahl.Ursprung[i] ) / strahl.richtung[i]

            [knotenNah, knotenFern] = ordne( strahl.richtung[i], knoten.links, knoten.rechts )

            //Auswerten:
            if( sE >= sMax )
                knoten = knotenNah
            else
            {   if( sE <= sMin  or  sE < 0)
                    knoten = knotenFern
                else
                {   Stack.push( knotenFern, sE, sMax )
                    sMax   = sE
                    knoten = knotenNah
                }
            }
        }

        for( objekt : knoten.Objekte() )
            if ( schneidet( strahl, objekt ) )
                return objekt

    }
    return NULL
}
```

Algorithmus 2 (kd-restart nach Foley et al. [1]) Iterative Umsetzung der Traversierung von k-d-Bäumen ohne einen Stapelspeicher

```
kdRestart( strahl, wurzel, sMin, sMax )
{
    sMaxAlt = sMax
    sMax    = sMin

    while( sMax < sMaxAlt ) //Diese Schleife simuliert den Neustart
    {
        knoten   = wurzel
        sMin     = sMax
        sMax     = sMaxAlt

        pushDown = true

        while( knoten.istKeinBlatt() )
        {
            i  = knoten.raumRichtung
            sE = ( knoten.wert - strahl.Ursprung[i] ) / strahl.richtung[i]

            [knotenNah, knotenFern] = ordne( strahl.richtung[i], knoten.links, knoten.rechts )

            //Auswerten:
            if( sE >= sMax )
                knoten = knotenNah
            else
            {   if( sE <= sMin  or  sE < 0)
                    knoten = knotenFern
                else
                {   sMax   = sE
                    knoten = knotenNah
                    pushDown = false
                }
            }

            if( pushDown )
                wurzel = knoten
        }

        for( objekt : knoten.Objekte() )
            if ( schneidet( strahl, objekt ) )
                return objekt

    }
    return NULL
}
```

Algorithmus 3 kd-restart mit Push-Down (nach Horn et al. [2])

```
kdRestart( strahl, wurzel, sMin, sMax )
{
    sMaxAlt = sMax
    sMax    = sMin

    while( sMax < sMaxAlt ) //Diese Schleife simuliert den Neustart
    {
        if( stack.empty() )
        {
            knoten   = wurzel
            sMin     = sMax
            sMax     = sMaxAlt
            pushDown = true
        }
        else
        {   [knoten, sMin, sMax] = stack.pop()
            pushDown = false
        }

        while( knoten.istKeinBlatt() )
        {
            i  = knoten.raumRichtung
            sE = ( knoten.wert - strahl.Ursprung[i] ) / strahl.richtung[i]

            [knotenNah, knotenFern] = ordne( strahl.richtung[i], knoten.links, knoten.rechts )

            //Auswerten:
            if( sE >= sMax )
                knoten = knotenNah
            else
            {   if( sE <= sMin  or  sE < 0)
                    knoten = knotenFern
                else
                {   stack.push( knotenFern, sE, sMax)
                    sMax   = sE
                    knoten = knotenNah
                    pushDown = false
                }
            }

            if( pushDown )
                wurzel = knoten
        }

        for( objekt : knoten.Objekte() )
            if ( schneidet( strahl, objekt ) )
                return objekt

    }
    return NULL
}
```

Algorithmus 4 kd-restart mit Push-Down und Short-Stack (nach Horn et al. [2])

```
kdRestartPac( strahl_v, wurzel, sMin, sMax )
{
    sMax_v    = sMax
    sMin_v    = sMin
    objekte_v = null
    fertig_v  = false

    while( not alle(fertig_v) and not alle(sMin_v<sMax_v)))
    {
        if( stack.empty() )
        {
            knoten   = wurzel
            imPfad_v  = sMin_v<=sMax_v and not fertig_v  //Maske für die Protokollierung
            pushDown = true
        }
        else
        {   [knoten, sMin_v, sMax_v, imPfad_v] = stack.pop()
            pushDown = false
        }

        while( knoten.istKeinBlatt() )
        {
            i  = knoten.raumRichtung
            sE_v = ( knoten.wert - strahl_v.Ursprung[i] ) / strahl_v.richtung[i]

            [knotenNah, knotenFern] = ordne( strahl_0.richtung[i], knoten.links, knoten.rechts )

            //Auswerten:
            willNah_v  = sE_v > sMin_v and imPfad_v
            willFern_v = sE_v <= sMax_v and imPfad_v

            if( jeder( willNah_v or not imPfad_v ) and niemand( willFern_v ) )
                knoten = knotenNah
            else
            {   if(  jeder( willFern_v or not imPfad_v ) and niemand( willNah_v ) )
                    knoten = knotenFern
                else
                {
                    knoten   = knotenNah
                    pushDown = false

                    stackImPfad_v = imPfad_v and willFern_v
                    stackSMin_v   = stackImPfad_v? max(sMin_v,sE_v):sMin_v
                    stack.push( knotenFern, stackSMin_v, sMax_v, stackImPfad_v )

                    live_v = willNah_v
                    sMax_v = willNah_v? min(sE_v, sMax_v):sMax_v
                }
            }
            if( pushDown )
                wurzel = knoten
        }

        for( objekt : knoten.Objekte() )
        {
            geschnitten_v = schneidet( strahl_v, objekt )
            objekte_v = geschnitten_v ? object:objecte_v
            if( alle( objekte_v not null ) )
                return objekte_v
        }
        fertig_v = fertig_v or ( objekte_v not null )
        sMin_v   = sMax_v
        sMax_v   = sMax
    }
    return objekte_v
}
```

Algorithmus 5 kd-restart mit Push-Down und Short-Stack und Packets (nach Horn et al. [2]). Alle Variablen mit Index v sind Vektoren bei denn von SIMD-Funktionen Gebrauch gemacht wird.