Multi Level Ray Tracing Algorithmus

Kathrin Becker

18. April 2010

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren wurden CPUs und GPUs moderner Desktop Rechner zunehmend leistungsfähiger, während Speicheroperationen weiterhin relativ teuer blieben. Aus diesem Grunde werden Verfahren aus dem Bereich Rastergraphiken, die in der Regel wenig Speicheroperationen benötigen und rechenintensiv sind, zunehmend auch für Echtzeitberechnungen eingesetzt [5]. Zu diesen Verfahren zählt das *Ray Tracing*, dessen Vorteile insbesondere darin liegen, dass geometrisch korrekte Darstellungen realisiert und *Shader* verwendet werden können [6].

Gegenstand der vorliegenden Arbeit ist die Vorstellung und kritische Analyse des *Multi Level Ray Tracing Algorithmus (MLRTA)*, der im Jahr 2005 von Rehtov, Soupikov und Hurley entwickelt wurde [5]. Er erweitert das traditionelle *Ray Tracing* um Optimierungen auf verschiedenen Ebenen: Durch Elimination überflüssiger Operationen wird der Rechenaufwand für die Darstellung einer Szene verringert. Zudem ist der Algorithmus so konzipiert, dass Leistungsmerkmale moderner CPUs, wie z.B. *Multi Processing* und SIMD, durch parallel arbeitende Threads (*Multi Threading*) genutzt werden können. Durch derartige Optimierungen kann der für die Darstellung einer Szene benötigte Rechenaufwand reduziert werden, was schließlich die Attraktivität von Echtzeit *Ray Tracing* für Implementierungen auf Desktop Rechnern erhöht.

In Abschnitt 2 werden zunächst die Grundlagen erläutert, auf denen der *MLRTA* basiert. Dazu zählen das traditionelle *Ray Tracing* Verfahren sowie k dimensionale Suchbäume (KD-Bäume), welche vom *MLRTA* zur Organisation der in einer Szene enthaltenen Objekte verwendet werden. Desweiteren werden Strategien zur Gruppierung von Strahlen erläutert, die dadurch motiviert werden, dass eine Traversierung oberer Suchbaumebenen für einzelne Strahlen oftmals nicht effizient ist. Anschließend wird das *Frustum Culling* Verfahren eingeführt, mit dem für eine Strahlengruppe ein Teilbaum ermittelt werden kann, der weiterhin alle Objekte enthält, die von jener geschnitten werden.

In Abschnitt 3 werden darauf aufbauend schließlich der *MLRTA*und verschiedene Strategien, mit denen dieser optimiert werden kann, erläutert. Außerdem wird auf die Performance des *MLRTA*s eingegangen, indem an verschiedenen Testszenen erzielte Ergebnisse betrachtet und mit den Ergebnissen des OpenRTs aus [6] verglichen werden.

Im letzten Abschnitt wird schließlich diskutiert, in welchen Fällen *Ray Tracing* durch den *MLRTA* für Echtzeitberechnungen verwendet werden kann. Auf den aufgedeckten Problemen basierend werden schließlich Ideen genannt, mit denen der *MLRTA* noch optimiert werden könnte.

2 Grundlagen

2.1 Ray Tracing

Ray Tracing ist ein Verfahren, das ursprünglich im Bereich Rastergraphiken eingesetzt wurde. Das Verfahren basiert auf wenigen Schritten: Wie in Abbildung 1 zu sehen ist, werden fiktive Strahlen, die einer gemeinsamen Quelle entstammen und jeweils ein Pixel der Bildebene schneiden, verfolgt. Dabei werden Schnittpunkte mit den Objekten der Szene berechnet. Auf einem Pixel der Bildebene wird schließlich das Element dargestellt, das den ersten Schnittpunkt mit dem durch jenen Pixel verlaufenden Strahl hat. Eine Ausnahme stellen Objekte dar, die durchlässig sind und somit den Lichtstrahl nicht vollständig absorbieren. In diesem Fall müssen die vom Strahl getroffenen Elementen so lange berücksichtigt werden, bis der Strahl vollständig absorbiert wurde. Auf derartige Ausnahmen werden wir in dieser Arbeit jedoch nicht eingehen.

Bevor die Schnittpunkttests durchgeführt werden, werden die Objekte der Szene in der Regel in Primitive (wie zum Beispiel Dreiecke) zerlegt. In der Praxis laufen die Schnittpunkttests somit also zwischen den Bausteinen der Objekte (Primitive) und den Strahlen ab.

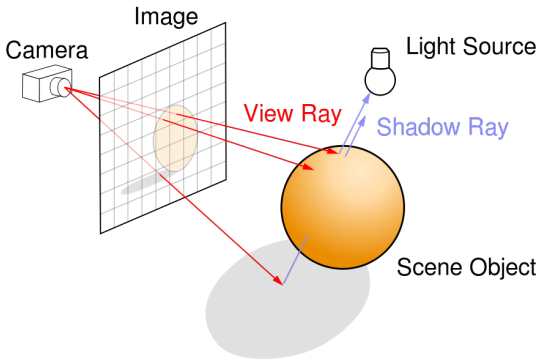

Abbildung 1: Rendern eines Bildes mittels Raytracing. Quelle: Wikimedia Commons, Autor: Henrik, URL: `http://en.wikipedia.org/wiki/Raytracing/`, Stand: 06.04.2010

2.1.1 Shading

Falls eine Szene verschiedene Lichtquellen enthält, so werden möglicherweise bestimmte Primitive stärker beleuchtet als andere. Zudem besteht die Möglichkeit, dass die Objekte Schatten aufeinander werfen. In solchen Situationen entspricht der auf der Bildebene darzustellende Farbwert nicht mehr dem des assoziierten Primitivs, vielmehr sind weitere Berechnungen erforderlich. Diese werden innerhalb der Computergrafik unter dem Oberbegriff *Shading* zusammengefasst.

Wir werden uns im folgenden mit der Darstellung von Schattierungen befassen, da diese Funktionalität auch im *MLRTA* implementiert und an Beispielszenen getestet wurde. Dazu betrachten wir wiederum Abbildung 1. In der dargestellten Szene ist eine Lichtquelle enthalten und ein Objekt, das Schatten auf andere Objekte werfen könnte. Betrachtet man den (fiktiven) unteren roten Strahl, so erkennt man, dass sich das von ihm getroffene Primitiv im Schatten des Szenenobjekts befindet. Rechnerisch lässt sich dies durch Aussendung eines fiktiven Strahls vom Primitiv zur Lichtquelle und Schnittpunkttests ermitteln, denn sofern jener Strahl vor dem Schnittpunkt mit der Lichtquelle ein weiteres Objekt schneidet, befindet sich das Primitiv im Schatten dieses Objektes und muss auf der Bildebene schattiert dargestellt werden. Fiktive Strahlen von Primitiven zu Lichtquellen werden als *Shadow Rays* oder auch als sekundäre Strahlen bezeichnet, während die im ersten Schritt verwendeten Strahlen als primäre Strahlen sowie auch als *View Rays* bezeichnet werden.

2.1.2 Ray Tracing für Echtzeitvisualisierungen

Wie im letzten Abschnitt aufgezeigt wurde, besteht beim *Ray Tracing* ein direkter Zusammenhang zwischen der Anzahl der ausgesendeten Strahlen und der Auflösung des Bildes. Falls Informationen über die Komplexität einzelner Szenenbereiche zur Verfügung stehen, besteht die Möglichkeit, in besonders komplexe Szenenbereiche eine höhere Strahlenzahl zu entsenden, was die Auflösung in den entsprechenden Bildbereichen erhöhen würde. Werden Lichtquellen innerhalb der Szene bewegt, so ist lediglich eine Nachberechnung der entsprechenden sekundären Strahlen erforderlich. Die Szene wird also weiterhin geometrisch korrekt dargestellt. Aufgrund dieser Möglichkeiten gewinnen Echtzeit Ray Tracing Implementierungen auf Desktop Rechnern zunehmend an Attraktivität [6].

Bei einer naiven *Ray Tracing* Implementierung würde man Schnittpunkttests zwischen allen Lichtstrahlen und allen Primitiven der Szene durchführen. Dieser Ansatz ist gerade bei einer hohen Szenenkomplexitẗ und einer großen Anzahl von Strahlen (hohe Auflösung) ausgesprochen rechenintensiv. Um *Ray Tracing* für Echtzeitberechnugen zu nutzen, sind also Optimierungen erforderlich, mit denen unnötige Rechenschritte eingespart werden können.

Organisiert man die in der Szene befindlichen Objekte in einer Datenstruktur, kann vor den eigentlichen Schnittpunkttests durch kostengünstige Berechnungen bereits eine Vielzahl von Objekten ausgeschlossen werden kann. Eine derartige Datenstruktur ist der KD-Baum, der im folgenden Kapitel eingeführt wird. In späteren Kapiteln werden wir erläutern, mit welchen weiteren Strategien und Algorithmen die erforderlichen Berechnungen effizienter gestaltet werden können.

2.2 KD-Bäume

k-dimensionale Bäume (kurz KD-Bäume) sind Datenstrukturen, die eine hierarchische Verwaltung der in einem *k*-dimensionalen Raum enthaltenen Punkte ermöglichen. Hierzu wird der Raum gemäß einer für den konkreten Anlass entwickelten Kostenfunktion schrittweise so lange in Unterräume unterteilt, bis ein Abbruchskriterium erfüllt ist. Die dabei entstehenden Unterräume werden durch die Knoten des KD-Baums repräsentiert: Der Wurzelknoten repräsentiert den gesamten Raum und seine Kinder die im ersten Teilungsschritt entstandenen Unterräume. Analog repräsentieren deren Kinder wiederum jeweils einen der Unterräume, die beim Teilen des von ihrem Elternknoten repräsentierten Unterraums entstanden sind. Den Blätter des Suchbaums entsprechen schließlich die feinsten Unterteilungen.

Wendet man *Ray Tracing* bei einer 3D-Szene an, so ist folglich ein 3-dimensionaler Suchbaum eine geeignete Datenstruktur. Wurden die Objekte der 3D-Szene in Primitive zerlegt, so können die von den Knoten des KD-Baums repräsentierten Unterräume statt Punkten auch jene Primitive enthalten. Die beim *MLRTA* verwendete Kostenfunktion basiert auf einem von MacDonald und Booth im Jahre 1990 entwickelten oberflächenbasierten Ansatz [4], bei welchem die Unterräume durch Hinzufügen achsenparalleler bzw. orthogonaler Ebenen gebildet werden und für jeden möglichen Unterraum die Anzahl der ihn schneidenden Primitve mit seiner Oberfläche multipliziert wird. Die Splitebene wird schließlich so platziert, dass dieser Wert für die beiden entstehenden Unterräume minimal ist. Für den *MLRTA* wurde dieser Ansatz erweitert, so dass auch 2*D*-Zellen und leere Zellen im Suchbaum gespeichert werden können. Dadurch befinden sich Elemente, deren Wichtigkeit vom rein oberflächenbasierten Ansatz unterschätzt werden würde, weiter oben im Suchbaum. Auch große Zellen, die keine Objekte enthalten, werden auf diese Weise frühzeitig erzeugt und werden somit oftmals in den oberen Ebenen des Suchbaums platziert. Dadurch können frühzeitig große und leere Bereiche der Szene ausgeschlossen werden. Außerdem wird die Erzeugung sehr kleiner Zellen unterbunden, weil diese nur selten von Strahlen getroffen werden und diese somit in den meisten Fällen unnötige Traversierungsschritte einfordern würden.

Beim *MLRTA* unterliegt der Bau eines KD-Baums somit folgenden Regeln:

1. Es wird immer dann eine leere Zelle erzeugt, wenn ihr Volumen mindestens 10 Prozent der aktuellen Zelle ausmacht.

2. Falls eine mögliche Teilungsebene existiert, die vollständig von koplanaren Dreiecken bedeckt ist, so wird diese ausgewählt und die Dreiecke werden vollständig der kleineren der beiden Unterzellen zugeordnet.

3. Als Terminierungskriterium wird nicht nur die Kostenfunktion herangezogen, bei welcher nicht mehr unterteilt wird, sobald die Kosten für ein Aufteilen des

Raums höher sind als für keine, sondern zusätzlich wird ein minimales Zellvolumen festgesetzt, das nicht unterschritten werden darf.

2.3 Gruppierung von Strahlen

Um *Ray Tracing* bei einer Szene, deren Primitive in einem wie in Abschnitt 2.2 erläuterten KD-Baum enthalten sind, anzuwenden, müssen für jeden Strahl zahlreiche Traversierungsschritte durchgeführt werden. Das Traversieren des Suchbaums ist eine rechenintensive Operation, die bei benachbarten Strahlen in den oberen Suchbaumebenen meist ähnliche Ergebnisse liefert.

Gruppiert man Strahlen, so lässt sich für die Strahlengruppe ein Knoten im KD-Baum ermitteln, dessen Teilbaum weiterhin alle Schnittpunkte zwischen Strahlen der Gruppe und Primitiven der Szene enthält. Somit genügt es, die für das *Ray Tracing* erforderlichen Schnittpunkttests nur im von diesem Knoten aufgespannten Teilbaum durchzuführen. Aus diesem Grunde werden wir Knoten mit solch einer Eigenschaft im folgenden als *Einstiegs-* bzw. auch als *Startknoten* bezeichnen.

Da für jede Strahlengruppe ein eigener Thread gestartet werden kann, können Berechnungen parallelisiert ablaufen. Insbesondere kann durch geschickte Gruppierung ein möglichst tief im Suchbaum liegender (optimaler) Einstiegsknoten gefunden werden, so dass zusätzliche Rechenzeit eingespart werden kann.

Um einen Einstiegsknoten zu finden, ohne dabei mit jedem einzelnen Strahl der Gruppe Schnittpunkttests durchzuführen, könnte man für die Berechnungen die konvexe Hülle der Strahlengruppe verwenden. Wie in Abbildung 2 dargestellt ist, kann dies jedoch zu Problemen führen: Die konvexe Hülle der Strahlengruppe wird von den Strahlen a, b, c, d, e und h aufgespannt, wobei e und h nicht die Box schneiden und alle anderen Strahlen der konvexen Hülle nur die untere Box schneiden. Da f nicht in der konvexen Hülle der Strahlengruppe enthalten ist und die obere Box trifft, stimmen bei diesem Beispiel die Schnittpunkte der konvexen Hülle nicht mit denen der darin enthaltenen Strahlen überein.

Aus diesem Grunde bietet es sich an, ein Verfahren zu nutzen, mit dem man definitiv außerhalb der konvexen Hülle befindliche Elemente identifizieren kann. Die inverse Menge der Primitive enthält dann nämlich alle Kandidaten für Schnittpunkte zwischen den Strahlen der Gruppe und Primitiven der Szene. Ein solches Verfahren ist das im folgenden Abschnitt vorgestellte Frustum Culling.

2.4 Frustum Culling

Frustum Culling ist ein Verfahren, das wie auch *Ray Tracing* aus dem Bereich der Rastergraphiken stammt [5] und zum Identifizieren von Objekten eingesetzt wird, die für die Darstellung der aktuellen Szene nicht relevant sind. Als *view frustum* wird der Teilraum einer Szene bezeichnet, der im Sichtbereich einer fiktiven Kamera liegt. In der

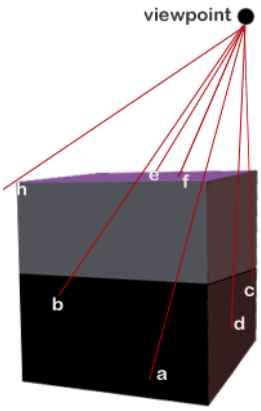

Abbildung 2: Berücksichtigt man nur die konvexe Hülle der Strahlengruppe, so wird der Schnittpunkt mit der oberen Box nicht entdeckt.

Regel hat dieser wie in Abbildung 3 dargestellt die Form eines Kegelstumpfes. Die das *view frustum* begrenzenden 6 Seitenflächen werden dabei als *clip planes* bezeichnet. Die *near clip plane* verhindert, dass sich Elemente zu nah an der Kamera befinden und die *far clip plane* begrenzt die Länge des Sichtbereiches. Die 4 Seitenflächen begrenzen zudem die seitliche Ausdehnung des *frustums*. [3]

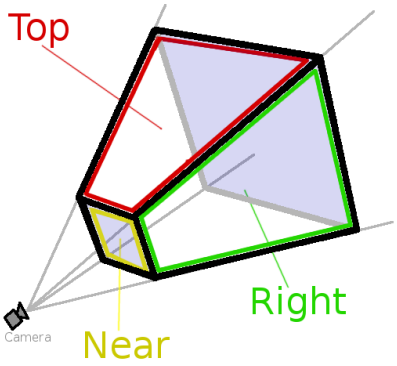

Abbildung 3: Darstellung eines View Frustums.

Somit befinden sich alle nicht vom *view frustum* umfassten Objekte außerhalb des Sichtbereichs. Damit sind sie für die Darstellung der Szene nicht relevant und könnten herausgeschnitten werden.

Um nicht in der Szene befindliche Objekte zu identifizieren, verpackt man diese zunächst in eine sogenannte *Axis Aligned Bounding Box (AABB)*. Man erhält auf diesem Wege somit relativ einen relativ großen *frustum* und relativ kleine *AABBs*. Anschließend wird für jede *clip plane* der zugehörige Normalenvektor ermittelt. Gemeinsam

mit dem die Kameraposition lokalisierenden Vektor beschreibt er dann eine Ebene, die die *clip plane* vollständig einbettet. Eine derartige Ebene sei im Folgenden als Kegelstumpfebene bezeichnet.

Nach Assarsson und Möller [2] untersucht man in einem nächsten Schritt die Lagebeziehung der Kegelstumpfebenen zu den *AABBs*. Hierfür berechnet man in positiver Richtung des Normalenvektors den Punkt der *AABB*, dessen Abstand möglichst weit von der Ebene entfernt ist (*p-edge*), gleichermaßen verfährt man in negativer Richtung (*n-edge*). Da die Schnittebene achsenparallel bzw. orthogonal gewählt wurde, kann durch einen Vergleich der Koordinatenwerte schnell bestimmt werden, ob beide Punkte auf der gleichen Seite der Ebene liegen, was bedeutet, dass die *AABB* nicht von der Kegelstumpfebene geschnitten wird. Je nach dem, auf welcher Seite sich der *p* und *n* value befinden, wird das Ergebnis *außerhalb* oder *innerhalb* zurückgegeben. Anderenfalls wird die *AABB* von der Kegelstumpfebene geschnitten und dem zu Folge wird als Ergebnis *schneidend* zurückgegeben [5].

Da für die Schnittpunkttests statt der *clip planes* die Kegelstumpfebenen verwendet werden, besteht die Möglichkeit, dass ein vom Frustum Culling Algorithmus gefundener Schnittpunkt außerhalb der eigentlichen *clip plane* liegt. In Abbildung 4 ist solch eine Situation dargestellt. Wird das Ergebnis *innerhalb* erzielt, so besteht die Möglichkeit, dass sich die *AABB* außerhalb einer anderen *clip plane* befindet. Folglich ist nur die Klassifikation *außerhalb* immer korrekt.

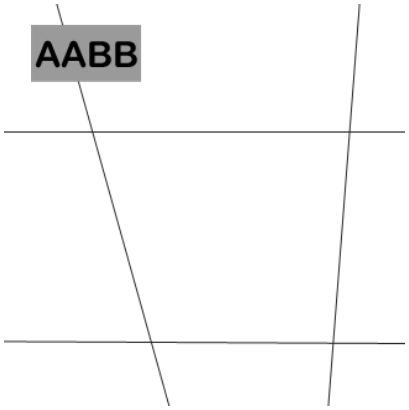

Abbildung 4: Querschnitt durch einen Kegelstumpf (Frustum) und eine AABB. Dargestellt sind auch die Kegelstumpfebenen, von denen eine die AABB schneidet, was zu einer Fehlklassifikation führt.

Umso kleiner eine *AABB* ist, umso unwahrscheinlicher ist es, dass sie von einer Kegelstumpfebene geschnitten wird. Ist eine *AABB* im Verhältnis zum *View Frustum* groß, so erhöht sich das Risiko einer Fehlklassifikation.

2.4.1 Analyse der Eignung des Frustum Culling Verfahrens für Ray Tracing bei Verwendung von KD-Bäumen und Strahlengruppen

In Abschnitt 2.3 wurde bereits angedeutet, dass sich *Frustum Culling* dazu eignet, Elemente, die nicht von einer Strahlengruppe getroffen wurden, zu identifizieren.

Da beim *Frustum Culling* Verfahren im Gegensatz zum *Ray Tracing* nicht mit Strahlen operiert wird, sondern mit *clip planes*, muss für eine Strahlengruppe zunächst ihre konvexe Hülle bestimmt werden. In Abbildung 2 würde diese von den 5 Strahlen *a*, *b*, *c*, *d* und *e* aufgespannt werden. Die Tatsache, dass das *view frustum* eine zusätzliche *clip plane* hat, spielt nach Assarsson und Möller für das *Frustum Culling* keine Rolle [2]. Da die Strahlen in der Regel die Bildebene schneiden, kann dieser Bereich als *near clip plane* angesehen werden. Eine *far clip plane* existiert nur, wenn die gesamte Szene von einer solchen Ebene begrenzt wird. Als *AABBs* können die im KD-Baum enthaltenen Unterräume verwendet werden. Mittels *Frustum Culling* könnten also die *AABBs*, die nicht vom *view frustum* getroffen werden, aus dem KD-Baum herausgeschnitten werden. Da der Suchbaum hierarchisch angeordnet ist, entspräche dies einem Abschneiden der Baumspitze. Die Wurzel des verbleibenden Teilbaums wäre dann der für die Gruppe optimale Einstiegsknoten.

Im Gegensatz zu den die Objekte umschließenden *Bouding Boxes* sind die im KD-Baum enthaltenen Unterräume mitunter sehr groß und die konvexe Hülle der Strahlengruppe ist in der Regel deutlich kleiner als ein *view frustum*, welches die gesamte darzustellende Szene umfasst. Nach den in Abschnitt 2.4 gewonnen Erkenntnissen wäre bei der betrachteten Vorgehensweise das Risiko hinsichtlich Fehlklassifikationen hoch. Aus diesem Grunde wurde das inverse *Frustum Culling* entwickelt [5], durch welches das Risiko hinsichtlich Fehlklassifikationen bei Verwendung kleiner *frustums* und großer *Bounding Boxes* herabgesetzt werden kann.

2.4.2 Inverses Frustum Culling

Beim inversen *Frustum Culling* wird nicht mehr über die Kegelstumpfebenen iteriert, sondern über die Seitenflächen der im KD-Baum enthaltenen Unterräume (der *AABBs*). Da ein Unterraum jeweils mittels einer Ebene unterteilt wird, genügt es zu prüfen, ob diese von der konvexen Hülle der Strahlengruppe getroffen wird oder nicht. Wird sie entweder gar nicht oder nur außerhalb der *AABB* getroffen, so kann einer der Unterräume aus dem KD-Baum gestrichen werden. Anderenfalls werden beide Unterräume von der konvexen Hülle der Strahlengruppe getroffen und der optimale Einstiegsknoten ist gefunden.

Die notwendigen Tests laufen in zwei getrennten Schritten ab. Der erste ist sehr grob, so dass in vielen Fällen auch dann ein Schnitt zwischen *AABB* und konvexer Hülle der Strahlengruppe vermutet wird, wenn tatsächlich keiner vorliegt. Aus diesem Grunde ist ein feinerer zweiter Schritt erforderlich, der eine geringere Anzahl von

Fehlklassifikationen beinhaltet. Laut [5] wurde empirisch wohl die beste Performance erzielt, wenn der erste Schritt beim Traversieren des Suchbaums auf die inneren Knoten des KD-Baums angewendet wird und der zweite auf die verbleibenden Blätter.

Im folgenden werden die beiden Schritte nacheinander im Detail beschrieben.

Schritt 1 Im ersten Schritt wird versucht, anhand der Splitebene und der Richtung der Strahlen einen Ast des KD-Baums auszuschließen. Die Situation kann man sich dabei so wie in Abbildung 5 dargestellt vorstellen:

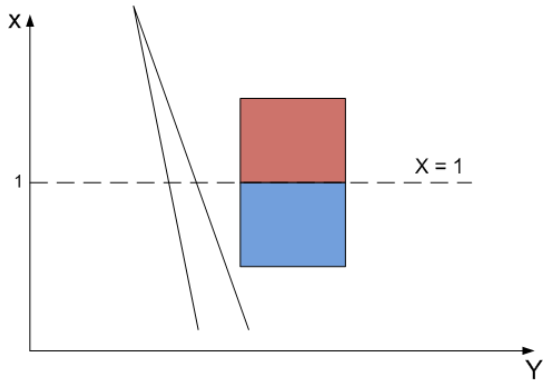

Abbildung 5: 2D Darstellung eines Unterraums, der mittels der Splitebene $x = 1$ in zwei Teilräume unterteilt wird. Außerdem dargestellt ist die konvexe Hülle einer Strahlengruppe, die die Splitebene außerhalb der *Bounding Box* schneidet.

Der aktuelle (innere) Knoten des KD-Baums repräsentiert einen Raum, der in zwei Unterräume aufgeteilt ist. In der Abbildung sind diese rot und blau dargestellt und werden durch die Splitebene $x = 1$ geteilt. Schneidet die konvexe Hülle der Strahlengruppe die Splittebene innerhalb der Box, so schneidet sie auch beide Unterräume. Liegt der Schnitt mit der Splitebene jedoch außerhalb, kann einer der beiden Unterräume ausgeschlossen werden, indem die Richtung der Strahlengruppe betrachtet wird. In Abbildung 5 kann folglich der rote Unterraum (und somit auch alle Kinder des zugehörigen Knotens) ausgeschlossen werden.

Wie bereits erwähnt, würde der inverse *Frustum Culling* Algorithmus einen Schnittpunkt zwischen der Strahlengruppe und dem blauen Unterraum nicht ausschließen können, obwohl der blaue Raum offensichtlich nicht von den Strahlen geschnitten wird. Stattdessen würde der dem blauen Unterraum zugehörige Teilbaum unter Durchführung des ersten Schritts weiter traversiert werden. Die verbleibenden Blätter des Baumes würden anschließend mit dem feineren zweiten Schritt überprüft werden.

Schritt 2 Im zweiten Schritt werden nur noch Blätter betrachtet, also jeweils *AABBs*, die nicht mehr durch Splitebenen unterteilt sind. Für jede Seite der *AABB* werden dazu sogenannte *clipping tests* durchgeführt. Um auf Schnitte mit allen 6 Seiten der *AABB* zu

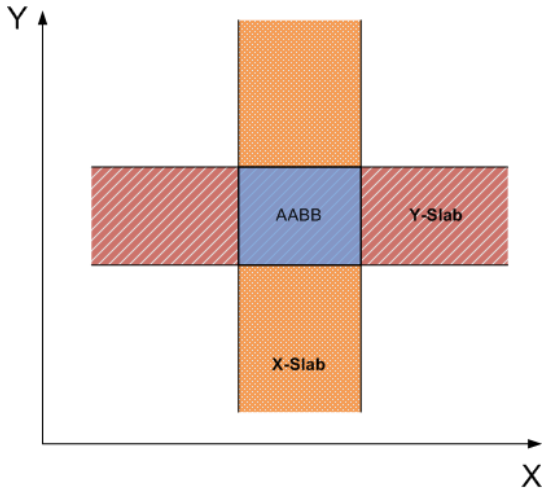

Abbildung 6: 2D Darstellung eines x- und y-Slabs einer *AABB*

testen, werden drei 2-dimensionale Fälle betrachtet, nämlich die Projektionen auf die x, y und z Ebene. Der Fall $z = 0$ wird im Folgenden exemplarisch genauer betrachtet.

Für diesen *clipping test* wird die *AABB* durch drei so genannte *Slabs* beschrieben. Ein *Slab* ist dabei als der Bereich zwischen zwei parallelen Ebenen definiert. Abbildung 6 stellt einen x-slab und einen y-slab für eine *AABB* dar.

Der zweite Test startet damit, dass für jeden Strahl die Distanz vom Ursprung zum Eintrittspunkt in den Slab und zum Austrittspunkt aus dem Slab berechnet wird. Falls nun eine der beiden folgenden Bedingungen erfüllt ist, so wird angenommen, dass das *frustum* die *AABB* außerhalb der *AABB* schneidet:

1. (Minimum der y Eintrittswerte) $>$ (Maximum der x Austrittswerte)
2. (Minimum der x Eintrittswerte) $>$ (Maximum der y Austrittswerte)

Abbildung 7 zeigt den zweiten Fall. Das Minimum der Eintrittswerte in den x-Slab ist grün eingezeichnet, während das Maximum der Austrittswerte aus dem y-Slab in rot dargestellt ist. Da der grüne Strahl länger ist als der rote, wird ein Schnitt der Strahlen mit der *AABB* ausgeschlossen.

3 Der Multi Level Ray Tracing Algorithmus

In diesem Abschnitt wird zunächst erläutert, wie der *MLRTA* grundsätzlich arbeitet. Anschließend werden Erweiterungen vorgestellt, durch die der Algorithmus noch weiter optimiert werden kann. Im letzten Abschnitt wird die Performance des *MLRTA* untersucht, indem an Testszenen erzielte Ergebnisse betrachtet und mit denen des OpenRTs aus [6] verglichen werden.

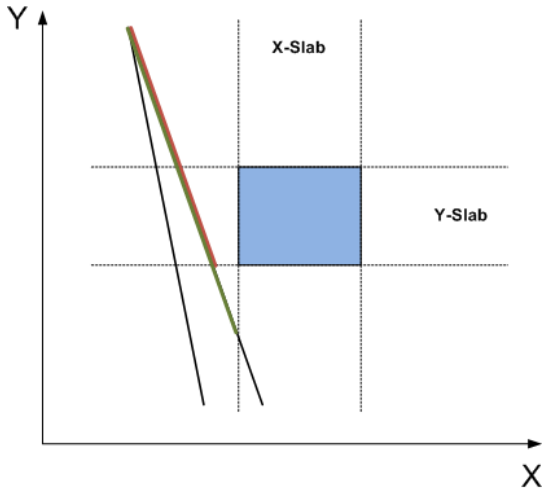

Abbildung 7: 2D Darstellung der x- und y-Slabs sowie der konvexen Hülle einer Strahlengruppe. Das Minimum der Eintrittswerte in den x-Slab ist grün, das Maximum der Austrittswerte aus dem y-Slab ist rot markiert.

3.1 Funktionsweise

Der *MLRTA* arbeitet unter der Prämisse, dass die Strahlen bereits gruppiert wurden, und basiert auf zwei voneinander unabhängigen grundlegenden Operationen: Als erstes wird für jede Strahlengruppe mittels inversem Frustum Culling ein optimaler Einstiegsknoten berechnet. Anschließend wird die Strahlengruppe entweder weiter aufgeteilt (weitere Einstiegsknotensuchen) oder es werden mittels einer Schnittpunktsuche die Blätter ermittelt, welche von den Strahlen der Gruppe tatsächlich getroffen werden. Letztere entspricht einer zunächst einer naiven *Ray Tracing* Implementierung, die jedoch -sofern ein tief im Suchbaum liegender Einstiegsknoten gefunden wurden- nur noch mit einer geringen Zahl von Strahlen an dem vom Einstiegsknoten aufgespannten (und somit in der Regel kleinen) Teilbaum durchgeführt wird. Da für das inverse *Frustum Culling* nur die konvexe Hülle der Strahlengruppe relevant ist, wird die genaue Position der Strahlen nur für die Schnittpunktsuche benötigt. In der Literatur wird die Suche nach einem Einstiegsknoten auch als *EP-search* (entry point search) und die Schnittpunktsuche als *XP-search* (crossing point search) bezeichnet [5]. Da wir die grundlegende Funktionsweise von *Ray Tracing* bereits diskutiert haben, werden wir uns im Folgenden mit der Einstiegsknotensuche (EP-Suche) befassen. Anschließend werden wir den *MLRTA* an einem kleinen Beispiel exerzieren.

Damit die mittels inversem *Frustum Culling* operierende EP-Suche gestartet werden kann, müssen folgenden Punkte erfüllbar sein:

1. Für jede im KD-Baum enthaltene Ebene lässt sich ein Rechteck berechnen, das alle möglichen Schnittpunkte mit der Strahlengruppe enthält.

2. Alle Strahlen haben die gleiche Richtung, d.h. die x, y und z Projektionen der Strahlen haben jeweils das gleiche Vorzeichen.

Sofern diese Bedingungen erfüllt sind, werden zur Ermittlung des optimalen Einstiegsknotens folgende Schritte durchgeführt:

1. Der Suchbaum wird mittels Tiefensuche traversiert. Alle Knoten, die Bifurkationen enthalten, werden in einem Last In First Out (LIFO) Stack gespeichert. Sobald das erste Blatt mit möglichen Schnittpunkten gefunden wurde, wird dieses als Kandidat für einen optimalen Einstiegsknoten gespeichert. Der oberste Knoten wird daraufhin vom Stack entnommen und der noch nicht besuchte vom entnommenen Knoten ausgehende Pfad (mittels Tiefensuche) traversiert. Falls auch dieser potentielle Schnittpunkte enthält, wird der entnommene Knoten zum Kandidaten für den optimalen Einstiegsknoten.

2. Es wird solange entweder der Baum mittels Tiefensuche traversiert oder ein neuer Knoten vom Stack entnommen, bis ein Abbruchskriterium eintritt. Sofern bei der Tiefensuche Blätter mit möglichen Schnittpunkten gefunden wurden, wird der zuletzt vom Stack entnommene Knoten zum Kandidaten für den optimalen Einstiegsknoten.

3. Der Algorithmus terminiert, wenn der LIFO Stack leer ist und auch die Tiefensuche beendet ist. Zurückgegeben wird der aktuelle Kandidat für den optimalen Einstiegsknoten.

Beispiel Abbildung 8 stellt einen KD-Baum dar, der den Ablauf des *MLRTA* für eine fiktive Strahlengruppe enthält. Innere Knoten, die als *outside EP* gekennzeichnet sind, stellen Unterräume dar, die von der Strahlengruppe nicht getroffen werden (z.B. 22 und 42). Blätter, die mögliche Schnittpunkte enthalten, sind rot dargestellt, leere Blätter weiß.

Die Einstiegsknotensuche startet bei Knoten 1. Da Knoten 22 keine Schnittpunkte enthält, wird die Suche an Knoten 21 fortgeführt. Da der Knoten eine Bifurkation enthält, wird eine Referenz zu Knoten 21 auf dem LIFO gespeichert und Knoten 31 traversiert. Da Knoten 42 keine Schnittpunkte enthält, kann dieser Ast gestrichen und die Suche an Knoten 41 fortgeführt werden. Dieser Knoten enthält wiederum eine Bifurkation, so dass auch eine Referenz zu diesem auf dem LIFO abgelegt werden muss. Knoten 51 hat nur ein relevante Kind, nämlich das Blatt 61. Dies wird zum Kandidaten für einen Einstiegsknoten. Anschließend wird Knoten 41 vom Stack entnommen und die Traversierung wird an Knoten 52 und 63 fortgeführt. Da 63 ein Blatt ist, das möglicherweise von der Strahlengruppe geschnitten wird, wird nun Knoten 41 zum Kandidaten für einen Einstiegsknoten. Da der von Knoten 64 aufgespannte Teilbaum für die Einstiegsknotensuche nicht relevant ist, wird Knoten 21 vom Stack entnommen.

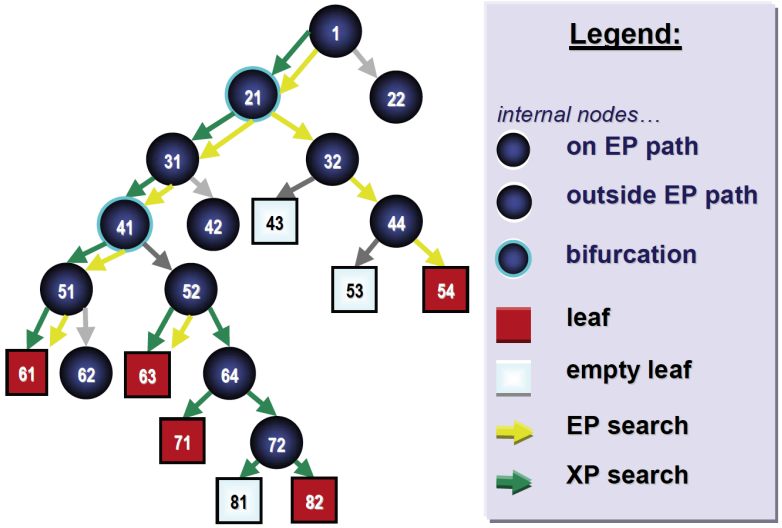

Abbildung 8: KD-Baum mit Pfaden, die vom *MLRTA* traversiert werden würde. Quelle: [5, page 1181]

Da das Blatt 54 potentielle Schnittpunkte enthält und der LIFO nun leer ist, wird Knoten 21 als optimaler Gruppen-Einstiegsknoten zurückgegeben. In diesem Fall ist damit in etwa noch der halbe KD-Baum für die Schnittpunktsuche relevant.

Bevor im nächsten Schritt die Schnittpunktsuche startet, kann die Gruppe noch gesplittet werden und für die einzelnen Gruppen im von Knoten 21 aufgespannten Teilbaum in getrennten Threads jeweils ein möglicherweise tiefer liegender Einstiegsknoten bestimmt werden.

Wir splitten die Strahlengruppe jedoch nicht, sondern starten unmittelbar die Schnittpunktsuche, für welche im Gegensatz zur Einstiegsknotensuche die genau Lage jedes Strahls bekannt sein muss.

Bei der Schnittpunktsuche wird der vom Einstiegsknoten aufgespannte Teilbaum nun hinsichtlich tatsächlich existierender Schnittpunkte untersucht. Dabei kann Knoten 32 ausgeschlossen werden, weil die (nun bekannten Strahlen) nur den von Knoten 31 repräsentierten Unterraum treffen. Im Gegensatz zur Einstiegsknotensuche kann die Suche unterhalb von Knoten 52 nicht abgebrochen werden. Somit müssen auch die Knoten 64 und 72 sowie die Blätter 71 und 82 auf Schnittpunkte getestet werden. Die Blätter 61, 63, 71 und 82 enthalten also alle Primitive, die von der Strahlengruppe getroffen werden.

3.2 Erweiterungen

Wie in Abschnitt 2.3 angedeutet wurde, kann bereits die Gruppierung der Strahlen so erfolgen, dass für diese ein möglichst guter Einstiegsknoten gefunden werden kann. Ein

Verfahren, mit dem auf diese Weise die Suche nach einem optimalen Einstiegsknoten verbessert werden kann, ist das *Tile Splitting*.

Zur Optimierung der Schnittpunktsuche wurde der *Interval Traversal Algorithmus* entwickelt. Beide Verfahren werden im Folgenden vorgestellt.

3.2.1 Tile Splitting

Tile Splitting ist ein Verfahren, mit dem man Strahlen, die einen ähnlichen Verlauf haben, der gleichen Gruppe zuordnen kann. Dazu unterteilt man die Bildebene gemäß bestimmter, vorher festzulegender Kriterien in Kacheln und ordnet genau die Strahlen einer Gruppe zu, die die gleiche Kachel schneiden. In [5] werden hierzu die folgenden Kriterien vorgeschlagen:

1. Eine *Initial Tile Size (ITS)* wird festgelegt.
2. Eine *Minimal Tile Size (MTS)* wird festgelegt. Sobald diese bei einem Teilungsprozess unterschritten wird, wird automatisch die Schnittpunktsuche gestartet.
3. Ein Splitfaktor definiert, in wie viele Kacheln eine bestehende Kachel in einem Schritt geteilt wird.

Der Algorithmus läuft also wie folgt ab: Im ersten Schritt wird die Bildebene hinsichtlich der *ITS* in Kacheln unterteilt. Anschließend wird jede Kachel so lange rekursiv unterteilt, bis eine *MTS* erreicht ist, deren Unterschreitung schließlich die Schnittpunktsuche startet.

Ergebnisse bei Anwendung von MLRTA und Tile Splitting In [5] werden Ergebnisse beschrieben, die unter Anwendung des *MLRTAs* in Kombination mit *Tile Splitting* erzielt wurden. Die Tests wurden an 2500 verschiedenen Modellen durchgeführt, variiert wurden jeweils die *ITS*, die *MTS* und der Splitfaktor.

In den Tests lag die initiale Größe der Kacheln zwischen 16×16 und 256×256 Pixeln, die Splitparameter wurden sowohl auf 4 als auch auf 16 gesetzt. Für die minimale Kachelgröße wurden Werte zwischen 4×4 und 64×64 verwendet. Als Metrik für die Performance wurden die Anzahl der notwendigen Traversierungsschritte im KD-Baum verwendet. Das beste Ergebnis wurde bei einer initialen Kachelgrösse von 128×128 und einem Splitfaktor von 16 erzielt, genauere Werte wurden jedoch nicht genannt. Allerdings habe die Performance insgesamt nur um 10 Prozent geschwankt, was darauf hin deutet, dass sich universale Splitparameter bestimmt lassen, die bei den meisten Szenen zu guten Ergebnissen führen. Es wurde kein Vergleich mit einer zufälligen Gruppierung der Strahlen durchgeführt, was möglicherweise die Bedeutung des *Tile Splitting* hervorgehoben hätte.

3.2.2 Interval Traversal Algorithmus

Für die Suche nach einem optimalen Einstiegsknoten braucht die genaue Lage der Strahlen nicht bekannt zu sein. In vielen Fällen ist die tatsächliche Lage der Strahlen jedoch bekannt, zum Beispiel beim Einsatz von *Tile Splitting*. In diesen Fällen kann die Suche nach einem optimalen Einstiegsknoten parallel zu der Schnittpunktsuche ausgeführt werden.

Der Interval Traversal Algorithmus verbindet die Vorgehensweisen der Einstiegsknotensuche mit denen der Schnittpunktsuche, in dem er für jeden einzelnen Strahl der Gruppe Operationen (nämlich Abstandsberechnungen) durchführt und daraus Schlussfolgerungen für die gesamte Strahlengruppe zieht (Ermittlung der Blätter, die Schnittpunkte mit der Strahlengruppe haben). Für die im KD-Baum von inneren Knoten repräsentierten durch eine Splitebene separierten *AABBs* wird beim Interval Traversal Algorithmus folgende Terminologie verwendet: Die Teilzelle, die sich näher an der Strahlenquelle befindet, wird als *nahe Zelle* bezeichnet. Die andere Teilzelle wird analog als *ferne Zelle* bezeichnet.

Für die Schnittpunkttests wird zuerst für jeden Strahl der Gruppe die Distanz vom Strahlenursprung zur *AABB* und zur Splitebene berechnet. Das jeweilige Maximum und Minimum wird anschließend in einem Float Array so gespeichert, dass die folgenden Belegungen gelten:

- `t_cell[0]` enthält den minimalen Abstand zur *AABB*
- `t_cell[1]` enthält den maximalen Abstand zur *AABB*
- `t_plane[0]` enthält den minimalen Abstand zur Splitebene
- `t_plane[1]` enthält den maximalen Abstand zur Splitebene

Der Interval Traversal Algorithmus überprüft nun mittels der folgenden Bedingungen, ob nur eine der Teilzellen von der Strahlengruppe getroffen wurde. Falls das der Fall ist, können die Schnittpunkttests mit der entsprechenden Teilzelle fortgesetzt werden:

1. Es gilt `t_cell[1]` $<$ `t_plane[0]`, d.h. alle Strahlen verlassen den Unterraum, bevor sie die Splitebene schneiden. In diesem Fall wurde also nur die nahe Zelle getroffen. Somit ist `t_plane[0]` zu aktualisieren, die Werte in `t_cell[1]` hingegen bedürfen keiner Veränderung.
2. Es gilt `t_cell[0]` $>$ `t_plane[1]`, was bedeutet, dass nur die ferne Zelle von der Strahlengruppe getroffen wird. In diesem Fall müssen entsprechend nur die Werte von `t_plane[1]` aktualisiert werden.

Falls keine der beiden Bedingungen gilt, so werden beide Zellen von der Strahlengruppe getroffen. In diesem Fall werden die `t_cell` Werte der fernen Zelle in einem Array `t_farthest[2]` auf die in Listing 1 beschriebene Weise gespeichert und auf einen LIFO Stack gelegt.

```
1 t_farthest[0]= max(t_cell[0], t_plane[0]);
2 t_farthest[1] = t_cell[1];
```

Listing 1: Speicherung der Werte der fernen Zelle

Der Interval Traversal Algorithmus wird anschließend mit den `t_cell` Werten der nahen Zelle fortgesetzt. Diese werden wie in Listing 2 gesetzt.

```
1 t_cell[1]=min(t_cell[1],t_plane[1]);
2 \\t_cell[0] braucht nicht aktualisiert zu werden, da wir ohnehin die
3 \\ nahe Zelle verwenden
```

Listing 2: Aktualisierung der Werte der nahen Zelle (wenn beide Zellen getroffen werden)

3.3 Performance

Abbildung 9: Testszenen. Von links nach rechts (Anzahl sichtbarer Dreiecke, Gesamtzahl Dreiecke): **erw6** (0.5K,0.8K), **soda hall** (36K, 2195K), **conference** (54K,283K), **dragon** (2261K,7219K). Quelle: [1]

In diesem Abschnitt werden die Ergebnisse besprochen, die in [5] bei Anwendung des *MLRTA* auf die in Abbildung 9 dargestellten Szenen erzielt wurden. Wie in Abschnitt 2.1 beschrieben, wurden die Objekte vor dem Einsatz von *Ray Tracing* in Primitive (Dreiecke) zerlegt. Hinter den Namen der Szene befindet sich eine Angabe hinsichtlich der Anzahl der sichtbaren und der Gesamtzahl der enthaltenen Dreiecke. Es ist zu erkennen, dass die Szenen von links nach rechts komplexer werden und die Anzahl der nicht sichtbaren Elemente zunimmt.

Um zu analysieren, inwiefern mittels *MLRTA* Rechenoperationen eingespart werden können, betrachten wir in Abschnitt 3.3.1 die Anzahl der für die Berechnung der Testszenen erforderlichen Traversierungsschritte.

In Abschnitt 3.3.2 wird anschließend analysiert, wie lange handelsübliche Desktop Computer für die Berechnung einer Szene benötigen. Als Metrik verwenden wir hierfür die Framerate, die angibt, wie oft die Szene pro Sekunde berechnet werden kann.

In beiden Fällen wurde eine Auflösung von 1024×1024 verwendet. Getestet wurde jeweils die alleinige Verwendung primärer Strahlen sowie auch die gemeinsame Verwendung von primären und sekundären Strahlen.

3.3.1 Rechenaufwand zur Darstellung einer Szene

In [5] wurden zum Testen der in Abbildung 9 dargestellten Szenen Pakete mit 16 (4×4) gleichverteilten Strahlen verwendet. Als Metrik wurde die Anzahl benötigter Traversierungsschritte (ATS) verwendet, die sowohl die bei der Einstiegsknotensuche als auch die bei der Schnittpunktsuche benötigten Traversierungsschritte beinhaltet. Wenn für eine Gruppe mit 16 Strahlen k Traversierungsschritte erforderlich waren, so wurde die ATS für dieses Paket wie folgt bestimmt:

$$ATS = \frac{k}{(\frac{1024*1024}{16*16})} \tag{1}$$

Um einen Referenzwert hinsichtlich der durch den *MLRTA* eingesparten Traversierungsschritte zu erhalten, wurde am gleichen KD-Baum eine naive *Ray Tracing* Implementierung angewendet, bei der für jeden einzelnen Strahl der Suchbaum mittels Tiefensuche nach möglichen Schnittpunkten traversiert wurde. In diesem Fall konnte die ATS direkt durch Zählen der besuchten Zellen bestimmt werden.

Für beide Herangehensweise wurde zudem die Anzahl der benötigten Schnittpunkttests gemessen, die jedoch nichts mit der in 3.1 vorgestellten Schnittpunktsuche zu tun hat. Die beim *Ray Tracing* im Suchbaum ermittelten Blättern (Räume, die von den entsprechenden Strahlen geschnitten werden), enthalten Referenzen auf die in ihrem Raum enthaltenen Primitive. Die Schnittpunkttests werden somit also entweder zwischen den Strahlen eines Pakets und den in den gefundenen Blättern enthaltenen Primitiven oder (bei der naiven Implementierung) zwischen einem Strahl und den in den gefundenen Blättern enthaltenen Primitiven durchgeführt.

Die Ergebnisse, die bei alleiniger Verwendung primärer Strahlen erzielt wurden, sind in Tabelle 1 angegeben. Die Ergebnisse, welche bei gemeinsamer Verwendung primärer und sekundärer Strahlen erhalten wurden, befinden sich in Tabelle 2.

Es fällt auf, dass der *MLRTA* bei den untersuchten Szenen nur die Anzahl der Traversierungsschritte deutlich verringern konnte, während die Anzahl der erforderlichen Schnittpunkttests nahezu gleich blieb. Dies überrascht jedoch nicht, weil auch bei einer naiven Ray Tracing Implementierung (nach einer deutlich höheren Zahl von Traversierungsschritten) schließlich die Blätter gefunden werden, die Schnittpunkte mit den Strahlen enthalten. Die etwas unterschiedlichen Werte für die Anzahl der Schnittpunkttests resultieren daraus, dass bei Verwendung des *MLRTA* für ein Strahlenpaket unter

	erw6 (0.5K, 0.8K)	**soda hall** (36K, 2195K)	**conference** (54K, 283K)	**dragon** (2261K, 7219K)
ATS mit MLRTA	3.98	20.87	32.52	32.65
ATS ohne MLRTA	13.00	49.98	71.37	42.18
ATS bei EK-Suche	0.51	2.30	4.44	2.72
Anz. SP-Tests mit MLRTA	1.09	2.48	1.59	19.97
Anz. SP-Tests ohne *MLRTA*	1.09	2.55	1.52	19.94

Tabelle 1: Übersicht über den Rechenaufwand zur Darstellung der Szenen aus Abb. 9 mit und ohne MLRTA. Verwendet wurden ausschließlich primäre Strahlen. (Verwendete Abkürzungen: EK=Einstiegsknoten, SP=Schnittpunkt)

	erw6 (0.5K, 0.8K)	**soda hall** (36K, 2195K)	**conference** (54K, 283K)	**dragon** (2261K, 7219K)
ATS mit MLRTA	10.07	53.75	69.07	45.01
ATS ohne MLRTA	24.83	101.06	117.22	58.41
SP-Tests mit MLRTA	1.25	3.71	2.17	23.51
SP-Tests ohne MLRTA	1.22	3.75	2.09	23.48

Tabelle 2: Rechenaufwand beim Rendern der Szenen aus Abbildung 9 mit und ohne *MLRTA* bei gemeinsamer Verwendung primärer und sekundärer Rays. (Abkürzungen analog zu Tabelle 1).

Umständen eine andere Anzahl Blätter ermittelt wird als es bei der naiven Implementierung der Fall ist. Für die Strahlengruppe musste zudem ein Durchschnittswert errechnet werden (= Anzahl der Schnittpunkttests pro Strahl), um Vergleichbarkeit zwischen den Ergebnissen zu gewährleisten.

Vergleicht man die in Tabelle 1 angegebenen Werte für die Anzahl der notwendigen Traversierungsschritte bei Implementierungen mit und ohne MLRTA, so fällt auf, dass sich sich das Verhältnis jeweils verändert hat: Bei Szene `erw6` waren ohne *MLRTA* 3.3 mal so viele Traversierungsschritte erforderlich, bei der Szene `conference` waren 2.4, bei der Szene `soda hall` 2.18 und bei der Szene `dragon` waren schließlich noch 1.29 mal so viele Traversierungsschritte erforderlich. Somit besteht möglicherweise ein Zusammenhang zwischen Komplexität, der Anzahl der verdeckten Elemente und Performance des MLRTA. In [5] wird hierauf jedoch nicht eingegangen. Ein Grund für die besonders guten Ergebnisse in der Szene mit den wenigsten Primitiven könnte darauf zurück zu führen sein, dass für Strahlengruppen, die große leeren Gebiete

		OpenRT 2.5 GHz P4 (1 thread)	MLRTA 2.4 GHz P4 (1 thread)	MLRTA 3.2 GHz P4 with HT (2 threads)
erw6	-shader	7.1	70.2	109.8
(0.8K)	+ shader	2.3	37.8	50.7
conference	-shader	4.55	11.2	19.5
(274K)	+shader	1.98	9.5	15.6
soda hall	-shader	4.12	21.1	35.5
(2195K)	+ shader	1.8	15.3	24.1

Tabelle 3: Vergleich der Frameraten beim Rendern dreier Szenen unter Verwendung von OpenRT und MLRTA. +shader bedeutet, dass auch Shadow Rays berechnet wurden, − shader bedeutet, dass kein Shader verwendet wurde.

schneiden, bereits bei der Einstiegsknotensuche nahezu der gesamte Suchbaum *abgeschnitten* werden konnte. Auch bei der Berechnung von *Shadow Rays* (siehe Tabelle 2) hat der *MLRTA* die Anzahl der notwendigen Traversierungsschritte deutlich herabgesetzt. Wir können festhalten, dass durch Verwendung des *MLRTA* in den betrachteten Testszenen die ATS gegenüber einer naiven Implementierung etwa um das $2,5$-fache verringert werden konnte.

3.3.2 Dauer für die Berechnung einer Szene

In diesem Abschnitt betrachten wir die für die Berechnung der in Abbildung 9 dargestellten Szenen benötigte Rechenzeit. Als Metrik wird die Framerate verwendet. Als Referenzwerte werden die beim OpenRT erzielten Werte aus [6] verwendet. Da der OpenRT nicht an der Szene `dragon` getestet wurde, können nur die Ergebnisse für die verbleibenden drei Szenen verglichen werden.

Anzumerken ist an dieser Stelle, dass die Tests auf handelsüblichen Notebooks [6] oder Desktop Rechnern [5] durchgeführt wurden.

Hinsichtlich der Framerate wurden die besten Ergebnisse wiederum bei Szene `erw6` erzielt. Da in [6] keine Angaben bezüglich der für die Organisation der Szene verwendeten Datenstruktur gemacht wurden, ist nicht ersichtlich, zu welchem Grad der KD-Baum des *MLRTAs* zu der Performanceverbesserung beigetragen hat. Auffällig ist jedoch, dass die Verringerung der erzielten Framerate beim *MLRTA* durch die Verwendung eines einfachen Shaders (Shattenberechnungen bei einer Lichtquelle) deutlich geringer ausfiel als im Referenzbeispiel. Während in [6] der Einsatz von Shadern noch als „bottleneck“ des *real-time Ray Tracings* dargestellt wurde, zeigen die Ergebnisse aus Tabelle 3, dass das Ausmaß dieses Effekts insbesondere von der Implementierung abhängt.

4 Diskussion

Die im letzten Abschnitt genannten Ergebnisse zeigen, dass sich die Performance von *Ray Tracing* Verfahren durch effiziente Algorithmen deutlich verbessert lässt, so dass dieses Verfahren tatsächlich für hochauflösende Echtzeitberechnungen auf handelsüblichen Desktop Rechnern attraktiv wird. In den betrachteten Beispielen wurde jeweils die gesamte Szene neu berechnet, in der Praxis ist dies jedoch selten notwendig, so dass sich auch in komplexeren Szenen höhere Frameraten erzielen lassen.

Dennoch fällt auf, dass sich die Ergebnisse mit zunehmender Szenenkomplexität und Anzahl verdeckter Elemente verschlechterten. Dies hängt damit zusammen, dass bei einer höheren Szenenkomplexität eine höhere Zahl an Traversierungsschritten für die Einstiegsknotensuche notwendig ist. Je mehr Objekte verdeckt sind, desto mehr gar nicht benötigte Zellen müssen bei der Einstiegsknotensuche traversiert werden. Aus diesem Grunde könnte man erwägen, den *MLRTA* mit *occlusion culling* Verfahren zu kombinieren.

Im Gegensatz zu den betrachteten Beispielen ist es kann es auch wünschenswert sein, komplexe Szenen hochauflösender darzustellen als weniger komplexe Szenen. Für derartige Prognosen kann der KD-Baum verwendet werden, indem man die Größe des von einem Knoten aufgespannten Teilbaums in Relation zur Größe des gesamten KD-Baums setzt. Damit könnte man also bei der Gruppierung der Strahlen auch ein inverses *Tile Splitting* durchführen: Man würde zunächst ein Level im Suchbaum bestimmen und für jeden Knoten dieses Levels die Komplexität des zugehörigen Unterraums ermitteln und gemäß diesem Wert die Strahlen auf die Unterräume verteilen. Dies hätte verschiedene Vorteile: In komplexen Szenen bräuchten weniger Teile des Suchbaums traversiert zu werden, da die Einstiegsknotensuche am entsprechenden Knoten (also bereits in einem möglichst tiefen Level) starten könnte. Da die Positionen der Strahlen bekannt sind, könnte man an sogar direkt mit dem Interval Traversal Algorithmus starten. Da sich die im letzten Kapitel präsentierten Ergebnisse dahingehend interpretieren lassen, dass der Hauptaufwand beim *Ray Tracing* im Traversieren des Suchbaums begründet ist, könnte auf diesem Wege *Ray Tracing* in komplexen Szenen deutlich effizienter gestaltet werden.

Insgesamt können wir also festhalten, dass in Zukunft mittels *Ray Tracing* durch noch intelligentere Algorithmen und bessere CPU und GPU Performance zunehmend höher auflösende Echtzeitberechnungen mit immer höheren Frameraten möglich werden. Es bleibt zu beobachten, in wie weit sich diese Fortschritte auch in neuen Produkten widerspiegeln, denn beispielsweise bei computergestützten Operationen in der Medizin könnten geometrisch korrekt arbeitende hochauflösende Visualisierungen von Vorteil sein.

Akronyme

MLRTA Multi Level Ray Tracing Algorithmus

LIFO Last In First Out

AABB Axis Aligned Bounding Box

MTS Minimal Tile Size

ITS Initial Tile Size

EP-Suche Einstiegsknotensuche

ATS Anzahl benötigter Traversierungsschritte

Literatur

[1] Paul Arthur, Navrátil William, and R. Mark. An analysis of ray tracing bandwidth consumption. Technical report, The University of Texas at Austin, Department of Computer Sciences, 2006.

[2] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5(1):9–22, 2000.

[3] Ian Parberr Fletcher Dunn. *3D math primer for graphics and game development*, chapter 15. Jones and Bartlett Publishers, 2002.

[4] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, 1990.

[5] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM.

[6] Ingo Wald, Timothy J. Purcell, Joerg Schmittler, Carsten Benthin, and Philipp Slusallek. Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics 2003 State of the Art Reports*. The Eurographics Association, 2003.