



Masterthesis

Verfahren zur photorealistischen Bildsynthese in Echtzeit

Eingereicht von:

Axel Tetzlaff
Bussardweg 24
22527 Hamburg

E-Mail: axel.tetzlaff@gmx.de

Abgegeben am:
8. April 2008

Referent:

Prof. Dr. Christian-A. Bohn
Fachhochschule Wedel
Feldstraße 143
22880 Wedel
Tel: (04103) 804840
E-Mail: bo@fh-wedel.de

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation	1
1.2 Struktur der Arbeit	1
2 Das Raytracingverfahren	3
2.1 Rekursives Raytracing	3
2.2 Distributed Raytracing	4
2.3 Global Illumination	5
2.4 Problemstellungen	6
2.5 Naive Lösung	7
3 Datenstrukturen zur Beschleunigung	9
3.1 Disjunkte Raumaufteilungsverfahren	9
3.1.1 Gleichmäßige Gitter	10
3.1.2 Octrees und hierarchische Gitter	13
3.1.3 BSP- & Kd-Trees	15
3.2 Partitionierung der Objektmenge	24
3.3 Hybride Raumaufteilung	29
3.4 Paket- und Frustumtraversierung	30
3.4.1 Pakettraversierung für reguläre Gitter	34
3.4.2 kd-Trees	37
3.4.3 Bounding Volume Hierarchies	38
3.4.4 Sekundärstrahlen	39
4 Animation	40
4.1 Klassifizierung von Problemen dynamischer Szene	40
4.2 Konsequenzen für die Datenstrukturen	41
4.2.1 Trennung von statischen und dynamischen Inhalten	41
4.2.2 Aktualisierung von Beschleunigungsstrukturen	42
4.2.3 Schnelle Konstruktion	43
4.3 Zusammenfassung	52
5 Hardware	53
5.1 Cache	53
5.2 Richtlinien	56

5.2.1	Kompaktheit	56
5.2.2	Ausrichtung	57
5.2.3	Trennung von Daten	58
5.2.4	Befehlscache	58
5.2.5	Rekursion gegenüber Iteration	61
5.3	Effiziente Speicherauslegung	62
5.3.1	Allokation und Anordnung	62
5.3.2	Kompakte Knotendarstellung	62
5.4	Parallelisierung	67
5.4.1	SIMD	67
5.4.2	Multithreading	72
6	Praxis	75
6.1	Implementierung	75
6.2	Programmstruktur	76
7	Fazit	79
Anhang		81
7.1	Baryzentrische Koordinaten	81
Literaturverzeichnis		83
Abkürzungsverzeichnis		87
Eidesstattliche Erklärung		88

Abbildungsverzeichnis

2.1	Unterdrückung von Aliasing-Effekten durch Oversampling	5
2.2	Auswahl der Beleuchtungsfaktoren	5
2.3	Globale Beleuchtungsphänomene(Color bleeding, Caustics)	6
3.1	Fehler bei der Objektklassifizierung für regelmäßiges Gitter durch die AABBs	11
3.2	Effiziente Gittertraversierung	11
3.3	Bestimmung des nächsten Voxels bei Gittertraversierung	13
3.4	Teapot-in-Stadium bei regelmäßigen Gittern	14
3.5	Fallunterscheidung bei Traversierung eines BSP-Trees	16
3.6	Keine Terminierung bei Schnitt außerhalb des aktuellen Voxels . . .	18
3.7	Effizienter Dreieck/Subvoxel Schnitttest	21
3.8	Begrenzte Anzahl von sinnvollen Positionen für Trennebene	25
3.9	Fehlende Kandidaten bei ausschließlicher Betrachtung der AABBs .	25
3.10	Keine frühe Terminierung für BVHs	28
3.11	Begrenzung der Traversierung durch Verkürzen des Strahlsegments .	29
3.12	Beispiel für eine Raumaufteilung mit der BIH	32
3.13	Früher Ausschluss ganzer Strahlpakete	33
3.14	Verfolgung von Strahlpaketen am Beispiel kd-Tree	34
3.15	Probleme bei Verfolgung von Strahlpaketen in regulärem Gitter (2D)	35
3.16	Gemeinsame Hauptachse für scheibenbasierte Gittertraversierung .	35
3.17	Scheibenbasierte Gittertraversierung	36
3.18	Bestimmung der Traversierungsreihenfolge für Strahlpakete	38
3.19	Kohärenz bei Sekundärstrahlen	39
4.1	Qualitätsverminderung einer BHV	44
4.2	Lazy build für binäre Raumaufteilung	45
4.3	Inkrementelles Zählen der Objekte vor/hinter der Trennebene . . .	46
4.4	Zusammenfassung der Ereignislisten	49
4.5	Von der Trennebene geschnittenes Dreieck erzeugt neue Ereignisse .	50
5.1	Alignment unterschiedlicher Datenstrukturen	57
5.2	Vierstufiges Pipelining	61
5.3	Speicherauslegung für binären Baum	63
5.4	Konstruktion eines Binärbaums im einem begrenzten Speicherblock	66

5.5	SSE Beispiel: Addition	68
5.6	Inverse Frustum culling	71
5.7	Strahlpaket schneidet unterschiedliche Objekte	72
5.8	Typische Cachestruktur aktueller Prozessoren.	73
6.1	Diagramm der wichtigsten Klassen	78
7.1	Strahl-Dreieck Schnitttest mit baryzentrischen Koordinaten	81

Quelltextbeispiele

3.1	Beschreibung der Kd-Tree Datenstruktur in erweiterter Backus-Naur-Form	16
3.2	Traversierung eines Kd-Trees	18
3.3	Eine einfache Version eines Konstruktionsalgorithmus für kd-Trees .	19
3.4	Suche nach der besten Trennebene in $O(n^2)$	26
3.5	Beschreibung einer binären BVH Datenstruktur in erweiterter Backus-Naur-Form	27
3.6	Die Bounding Interval Hierarchy Inter Datenstruktur in erweiterter Backus-Naur-Form	29
3.7	Konstruktion einer BIH mit globaler Heuristik	31
3.8	Traversierung der Bounding Interval Hierarchy	32
4.1	Definition des Ereignisdatentyps für die Kandidatengenerierung . .	45
4.2	Algorithmus zur Bestimmung der besten Trennebene in $O(N \log^2 N)$.	47
5.1	Sinnvoller Einsatz von inlining	60
5.2	Kompakte Speicherauslegung eines Bounding Intervall Hierarchy Knotens	63
5.3	Kompakte Speicherauslegung eines kd-Tree Knotens	64
5.4	<i>Structure of arrays Anordnung</i> im Vergleich zum <i>Array of structures</i> .	68
5.5	Definition einer Datenstruktur für Strahlpakete im SoA-Format . .	70

Zusammenfassung

Raytracing ist ein grundlegengendes Verfahren der Computergrafik zur Synthese fotorealistischer Bilder. Die Simulation einzelner Lichtpfade ermöglicht es aus einer abstrakten Beschreibung einer dreidimensionalen Szene ein zweidimensionales Bild zur Darstellung am Computermonitor zu generieren. Die Verfolgung der Strahlen die den Lichtfluss simulieren ist ein sehr rechenaufwändiger Vorgang. Bis vor wenigen Jahren wurde deswegen nicht in Erwägung gezogen Raytracing für die Bildgenerierung in Echtzeit einzusetzen. Die steigende Rechenleistung der Prozessoren und die Entwicklung effizienter Algorithmen ermöglichen mittlerweile die Visualisierung virtueller Szenen mit dem Raytracingverfahren auf aktuellen Arbeitsplatzrechnern. Globale Beleuchtung und dynamische Szenen eröffnen eine neue Problemklasse die noch effizientere Ausnutzung von Kohärenzen der Szenendaten und der aktuellen Hardware fordert. Die langjährige Forschungsarbeit auf diesem Gebiet hat eine enorme Menge Literatur zu diesem Thema hervorgebracht. Diese Arbeit gibt eine Einführung in die wichtigsten Techniken die für die Implementierung eines Echtzeit Raytracers benötigt werden. Außerdem sollen die Grundlagen für das Verständnis aktueller Publikationen gelegt werden, welche die Kenntnis dieser Techniken voraussetzen. Im Zusammenhang dieser Arbeit wurde ein offenes Framework erstellt, in dem viele der vorgestellten Verfahren leicht verständlich implementiert wurden. Durch hochoptimierten Code verbergen existierende, frei verfügbare Implementierungen oft die Arbeitsweise der eigentlichen Algorithmen. Bei dem mit dieser Arbeit erstellten Quellcode wurde besonderer Wert auf eine klare Struktur und gute Nachvollziehbarkeit der implementierten Algorithmen gelegt, ohne dabei auf den Anspruch interaktiver Frame-raten zu verzichten.

1 Einleitung

1.1 Motivation

Der Großteil der interaktiven 3D-Anwendungen, die heutzutage auf dem Markt sind, werden mittels des Rasterisierungsverfahrens und der dafür angepassten Hardware umgesetzt. Dabei werden die einzelnen Dreiecke, aus denen die virtuelle Szene zusammengesetzt ist, unabhängig voneinander behandelt. Dadurch wird eine effiziente, parallele Implementierung des Algorithmus in speziell dafür gestalteter Hardware, die in nahezu allen Grafikkarten eingebaut ist, möglich. Die wachsenden Anforderungen an die Qualität der dargestellten Bilder erhöht jedoch zunehmend die Komplexität der Programme, die diese Technologie nutzen. Gerade bei Objekten deren Erscheinung von anderen Objekten abhängt (zum Beispiel bei Schattenwurf) wird die Simulation nicht vom Rasterisierungsverfahren abgedeckt. Komplexe Beleuchtungsszenarien werden deshalb oft in Software vorbereitet und mit in der Szenenbeschreibung abgespeichert. Das Raytracingverfahren bildet mit seinem physikalisch motivierten Ansatz eine alternative Vorgehensweise. Der Fluss des Lichts von den Lichtquellen zum Auge des Betrachters wird durch die Verfolgung von virtuellen Lichtstrahlen simuliert. Dieser allgemeine Ansatz ermöglicht die Darstellung vieler realistische Phänomene ohne sich dem Programmiermodell einer bestimmten Hardwarestruktur anpassen zu müssen. Durch seinen hohen Rechenaufwand wurde das Verfahren bis vor wenigen Jahren für Echtzeitdarstellungen nicht in Betracht gezogen. Die steigende Rechenleistung ermöglichte in den letzten Jahren erfolgreiche erste Implementierungen mit interaktiven Frameraten auf gewöhnlichen Arbeitsplatzrechnern.

1.2 Struktur der Arbeit

Die Arbeit gibt eine Einführung in die Themengebiete mit denen sich ein Softwareentwickler auseinandersetzen muss, um eine effiziente Implementierung eines Raytracers zu entwickeln. Zu diesem Zweck werden in Kapitel 2 zunächst die grundlegende Vorgehensweise des Raytracingverfahrens und die dabei zu lösenden Probleme aufgezeigt. Kapitel 3 zeigt mittels welcher Algorithmen das Kernproblem, die Bestimmung eines Schnitt-

punktes von einem Strahl und der Szenengeometrie, effizient gelöst werden kann. Die Implementierung dieser Algorithmen auf einem Computer bringt gewisse Implikationen mit sich. Eine naive Umsetzung kann die Effizienz der angewendeten Verfahren um Größenordnungen herabsetzen. Deswegen werden in Kapitel 5 verschiedene Eigenschaften aktueller Computer, und Richtlinien zur Entwicklung effizienter Anwendungen auf solchen, vorgestellt. Im letzten Kapitel 6 wird auf die praktische Entwicklung einer Raytracinganwendung eingegangen und ein Überblick über die im Zusammenhang mit dieser Arbeit entwickelte Software gegeben. Am Rand mancher Seiten befinden sich Anmerkungen mit den Symbolen . Diese verweisen auf die Stellen im Quellcode an denen ein zuvor beschriebener Algorithmus zu finden ist. Das Beispiel auf dieser Seite verweist auf die Methode `helloWorld` der Klasse `MyClass` im Verzeichnis `examples`.



2 Das Raytracingverfahren

Seit Beginn der Computergrafik ist eines der Hauptziele die Erzeugung fotorealistischer 2D-Bilder aus einer abstrakten 3D-Beschreibung einer (realen) Szene. Die erzeugten Bilder sollen von Photos nicht zu unterscheiden sein.

Um ein 2D-Bild einer 3D-Szene am Computer anzuzeigen muss die Bildebene in diskrete Bildelemente - die Pixel - aufgeteilt werden. Für jedes Pixel muss zunächst festgestellt werden, welches Objekt “durch” dieses Pixel sichtbar ist. Durch die Eigenschaften des Objekts kann darauf geschlossen werden, wieviel von dem beim Objekt eintreffenden Licht vom betrachteten Punkt auf dem Objekt, in Richtung des Betrachters reflektiert wird.

2.1 Rekursives Raytracing

Beim Raytracing werden zunächst so genannte *Primärstrahlen* generiert: Für jedes Pixel in der Bildebene wird ein Strahl generiert, der bei der Kameraposition entspringt und durch das jeweilige Pixel verläuft. Um den Farbwert des Pixels zu bestimmen wird als nächstes festgestellt welches Objekt der Szene der Strahl in seinem Verlauf als erstes schneidet. Dieser Teil des Raytracings heißt **Raycasting** und wurde bereits von [Appel \(1968\)](#) vorgestellt. Ein Strahl lässt sich mathematisch wie in Gleichung 2.1 beschreiben. Sind $t_{min} = 0$ und $t_{max} = \infty$ handelt es sich um einen unbegrenzten Strahl, andernfalls um ein Strahlsegment. Um bestimmen zu können wieviel Licht am Schnittpunkt in die entgegengesetzte Richtung des Strahls abgestrahlt wird, muss jedoch vorher bekannt sein wieviel Licht am Schnittpunkt eintrifft. Danach kann aus den Materialeigenschaften und der Beleuchtung am Schnittpunkt der entsprechende Beitrag zur Intensität des Pixels berechnet werden.

$$R = \vec{o}_R + t * \vec{d}_R, (t \in R, t_{min} \leq t < t_{max}) \quad (2.1)$$

Den Hauptteil zur Beleuchtung eines Punktes trägt meistens das Licht bei, welches direkt von einer Lichtquelle auf den Punkt abgegeben wird. Wieviel Licht einer Lichtquelle einen Punkt im Raum erreicht hängt zum einen von den Parametern der Lichtquelle ab,

aber ebenso von der umliegenden Geometrie, zum Beispiel ob sich ein anderes Objekt zwischen diesem Punkt und der Lichtquelle befindet. Um dies festzustellen werden so genannte *Schattenstrahlen*, die von dem Schnittpunkt zu den Lichtquellen führen, auf Schnitt mit den anderen Objekten der Szene getestet. Wird kein Schnitt gefunden trägt der entsprechende Anteil der Lichtquelle zur Beleuchtung des Schnittpunktes bei. Für transparente und spiegelnde Materialien beschreibt Whitted (1980) die Verfolgung weiterer, so genannter *Sekundärstrahlen*, da das menschliche Auge für diese Phänomene besonders empfindlich ist. Für spiegelnde Materialien wird der ideal reflektierte Strahl nach Gleichung 2.2 berechnet. Für transparente Materialien wird der ideal transmittierte Strahl unter Berücksichtigung des Snelliusschen Brechungsgesetzes(Gleichung 2.3) ermittelt.

Das abgestrahlte Licht an den Schnittpunkten dieser Strahlen trägt ebenfalls zur Beleuchtung des aktuellen Schnittpunkts bei. Hierfür muss zunächst bestimmt werden, welche Objekte die *Sekundärstrahlen* treffen und wieviel Licht in Richtung des Schnittpunktes ausgesendet wird. Da es sich hierbei um die selbe Fragestellung wie für den Primärstrahl handelt, lässt sich die Lösung besonders effizient durch Rekursion umsetzen, was dem Verfahren seinen Namen gab. Durch die Materialeigenschaften des Objekts, auf dem sich der Schnittpunkt befindet wird bestimmt, wieviel von den berechneten Anteilen der Beleuchtung zur Intensität des Pixels beitragen.



$$d_{reflect} = \vec{d}_r - 2 * (\vec{n}_s * \vec{d}_R) * \vec{n}_s, |\vec{d}_R| = 1, |\vec{n}_s| = 1 \quad (2.2)$$

$$d_{refract} = (d_R - (\vec{n}_s * (\vec{n}_s * \vec{d}_R)) * \frac{n_{out}}{n_{in}} - \vec{n}_s * \sqrt{(1 - (\frac{n_{out}}{n_{in}})^2) * (1 - (\vec{n}_s * \vec{d}_R)^2)}) \quad (2.3)$$

2.2 Distributed Raytracing

Bei der Ermittlung der Farbe eines Pixels durch einen Strahl nimmt man implizit an, dass die Fläche des Pixels unendlich klein ist. Diese Näherung kann zu unerwünschten Aliasingeffekten, wie den bekannten Treppeneffekten (siehe Abbildung 2.1 links), führen. Durch *Oversampling* (auch Supersampling) können diese Effekte verringert werden. Das heißt für die Fläche, die der Pixel in der Bildebene verdeckt, werden mehrere Strahlen verfolgt und die Ergebnisse gemittelt. Das Antialiasing muss sich hierbei nicht nur auf den Bildraum beziehen, sondern kann gleichzeitig auch Bewegungen oder ein virtuelles Kameraobjektiv abtasten. Dieses Verfahren bezeichnet man als *Distributed Raytracing*. Es wurde von Cook u. a. (1984) entwickelt und ermöglicht die Darstellung von Tiefenschärfe, Bewegungsunschärfe und unscharfen Reflexionen und Schatten. Die Steigerung

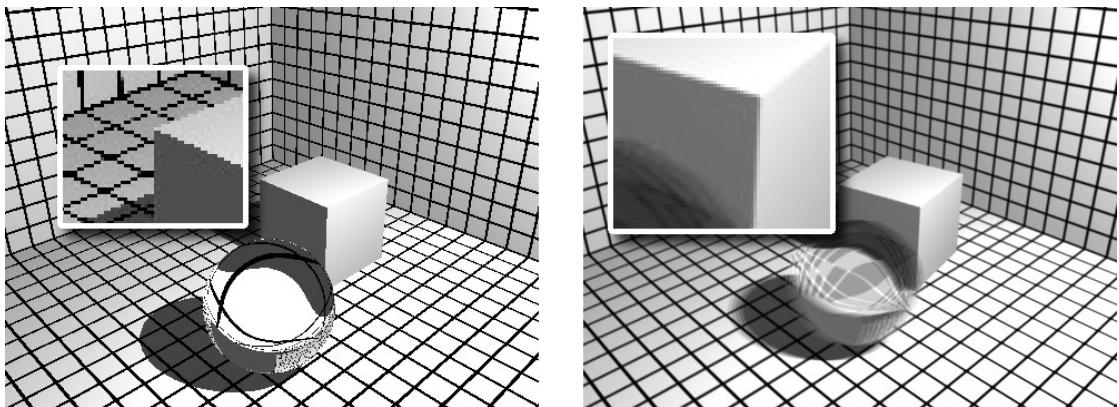


Abbildung 2.1: links: Aliasing beim Rendern ohne Oversampling, rechts: Antialiasing durch Oversampling führt zu glatteren Kanten und Motionblur bei sich bewegenden Objekten

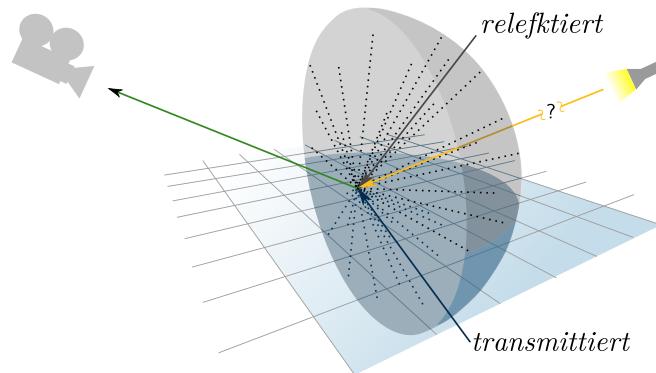


Abbildung 2.2: Beim Raytracing nach Whitted werden neben den direkten Beleuchtungsanteilen der Lichtquellen ausschließlich der am Schnittpunkt ideal reflektierte sowie der ideal transmittierte Strahl für die Beleuchtung berücksichtigt. Alle weiteren einfallenden Lichtanteile werden ignoriert.

der Bildqualität geht allerdings mit einem erhöhten Rechenaufwand für die zusätzlichen Strahlen einher.

2.3 Global Illumination

In der Realität empfängt ein Punkt im Raum aus unendlich vielen Richtungen Licht. Beim Raytracing, wie bisher beschrieben, werden zur Berechnung der Beleuchtung eines Punktes aber nur sehr wenige, ausgewählte Richtungen betrachtet, nämlich die ideal reflektierte, die ideal transmittierte und die Richtungen zu den Lichtquellen (siehe Abbildung 2.2). Beleuchtung aus anderen Richtungen wird ignoriert. Durch *Distributed Raytracing* können auch unscharfe Reflexionen und Transmissionen dargestellt werden.

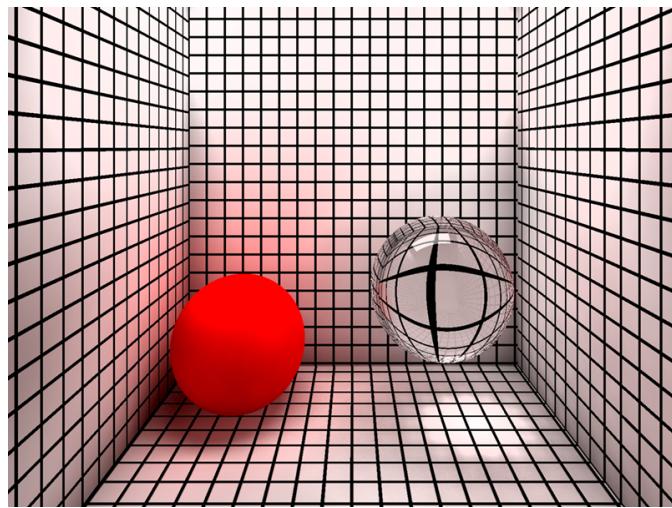


Abbildung 2.3: Die rote Kugel 'blutet' ihre Farbe aus. Dadurch das die Glaskugel das Licht bündelt erscheint die Fläche darunter heller als das Umfeld. (Erstellt mit dem GI-Raytracer von Mathias Leonhardt)

Der Einfluss von diffuser Reflexion und Bündelung von Licht durch Objekte in der Szene bleibt aber auch hier unbeachtet. Durch stochastische Emission weiterer Strahlen vom betrachteten Punkt aus, kann eine genauere Abschätzung der Beleuchtung dieses Punktes getroffen werden. Dadurch werden Phänomene wie *Color bleeding* (durch diffuse Reflexion) und *Caustics* (Lichtbündelungseffekte) sichtbar gemacht (siehe Abbildung 2.3). Auch hier resultiert die Steigerung der visuellen Qualität in erhöhtem Rechenaufwand für weitere Strahlberechnungen.

Ein weiteres Phänomen, welches nicht vom klassischen Raytracing dargestellt werden kann, entsteht durch Materialien, die Teile des Lichts weder reflektieren noch *ideal* transmittieren (und es nicht einfach absorbieren). Bei den besagten Materialien tritt das Licht in das Medium ein, wird unter der Oberfläche gestreut und tritt an einer anderen Stelle wieder aus. Marmor ist zum Beispiel ein solches Material, bei dem das so genannte *Subsurface scattering* auftritt.

2.4 Problemstellungen

Alle der genannten Verfahren müssen zur Ermittlung bestimmter Werte Informationen über Schnitte von Strahlen mit der Szene besitzen. Allgemein gilt: je realistischer die Darstellung sein soll, desto mehr Strahlen werden benötigt. Die beschränkt sich nicht nur auf Primärstrahlen. Doch selbst beim rekursiven Raytracing wird ohne Optimierun-

gen, wie bereits Whitted (1980) feststellte, über 90% der Laufzeit mit der Ermittlung von Strahl-Geometrie-Schnittpunkten verbracht. Die Optimierung der Lösung dieses Problems verspricht demnach den größten Leistungszuwachs für einen Raytracer.

Die Schnittpunktsuche lässt sich in drei Spezialfälle einteilen:

Sichtbarkeit eines Paares von Punkten Bei diesem Problem handelt es sich um ein Entscheidungsproblem. Es wird danach gefragt, ob ein Strahlsegment von *irgendeinem* Objekt der Szene geschnitten wird. Nähere Informationen über das geschnittene Objekt oder die Position des Schnittes sind dabei uninteressant. Diese Problem muss hauptsächlich für Schattenstrahlen gelöst werden.

Nächster Schnittpunkt zum Ursprung Für *Primärstrahlen* muss beantwortet werden, welches das dem Ursprung des Strahls am nächsten gelegene Objekt ist, das vom Strahl geschnitten wird. Da es sich hierbei um ein Suchproblem handelt, ist es algorithmisch härter als das zuvor beschriebene Entscheidungsproblem.

Alle Schnittpunkte Für einige Algorithmen zur Darstellung von globaler Beleuchtung ist es notwendig alle Schnittpunkte eines Strahlsegments mit der Szene zu kennen. Dementsprechend ist dies das härteste der drei genannten Probleme.

Im Folgenden wird hauptsächlich auf die effiziente Lösung des Suchproblems (nächster Schnittpunkt) eingegangen. Das Entscheidungsproblem kann in Abhängigkeit von der Anzahl und Ausprägung der Lichtquellen zwar sogar öfter auftreten, lässt sich aber auf das Suchproblem abbilden. Oft kann eine vereinfachte Version der Lösung des Suchproblems für das Entscheidungsproblem verwendet werden. Die effiziente Ermittlung aller Schnittpunkte wird zunächst nicht weiter diskutiert, da diese lediglich für wenige, ausgewählte Verfahren durchgeführt werden muss.

2.5 Naive Lösung

Das Problem des nächsten Schnittpunktes lässt sich sehr einfach mit dem folgenden Algorithmus lösen:

Teste für alle Objekte in der Szene:

- Schneidet der Strahl das Objekt ?
- Wenn ja: Falls der Schnittpunkt näher am Ursprung des Strahls liegt als der bisher beste gefundene Schnittpunkt: Speichere das aktuelle Objekt und den aktuellen Schnittpunkt als neuen besten Schnittpunkt.

acceleration
PrimitiveList

Dieser Ansatz ist für eine kleine Anzahl von Objekten durchaus einsetzbar. Wegen seiner linearen Komplexität und da die Anzahl der Objekte in realistischen Szenen von einigen tausend bis zu mehreren Millionen reichen kann, stellt dieser Ansatz jedoch keine wirkliche Option dar.

3 Datenstrukturen zur Beschleunigung

Um das Problem effizienter zu lösen, wurden verschiedene Verfahren entwickelt, von denen eine Auswahl im Folgenden näher vorgestellt wird. Alle der vorgestellten Verfahren nutzen den Effekt, dass sich die Kosten für die Konstruktion einer zusätzlichen Datenstruktur amortisieren. Die Datenstruktur kann die Beantwortung einer Strahl-Geometrie-Schnittpunktfrage signifikant beschleunigen, so dass der Leistungsgewinn bei einer hohen Anzahl von Strahlen höher ist als die Kosten, die für den Aufbau der Struktur entstehen. Das gemeinsame Ziel dieser Verfahren ist es, die Anzahl der nötigen Schnitttests des Strahls mit den Objekten der Szene auf ein Minimum zu reduzieren. Die Unterschiede der verschiedenen Ansätze liegen zum einen darin, ob der Raum oder die Menge der Objekte partitioniert werden. Zum Anderen wird teilweise eine gleichmäßige, bei anderen Verfahren eine ungleichmäßige Aufteilung gewählt. Außerdem hat die Entscheidung, ob eine Datenstruktur hierarchisch oder flach angelegt wird, großen Einfluss auf die Effizienz der Schnittpunktbestimmung.

Um Strahlen, welche die Szene außerhalb verfehlten schnell verwerfen zu können, wird in der Vorbereitungsphase eine Axis Aligned Bounding Box(AABB) für die Szene berechnet. Eine AABB beschreibt einen achsenparallelen Quader, der die enthaltenen Objekte eng umschließt. Repräsentiert wird er von zwei Vektoren $AAB\vec{B}_{min}, AAB\vec{B}_{max}$, die jeweils die minimalen/maximalen Werte enthalten, die eine Komponente eines Punktes im Raum annehmen darf, damit dieser innerhalb der AABB liegt.

Ein effizienter und robuster AABB-Strahltest wird von Williams u. a. (2005) beschrieben. Neben der Aussage ob ein Strahl eine AABB schneidet, werden zusätzlich zwei Parameter t_{min}, t_{max} ermittelt. Diese geben für den Fall, dass der Strahl die AABB schneidet, Auskunft darüber, welches Segment des Strahls innerhalb der AABB liegt.



3.1 Disjunkte Raumaufteilungsverfahren

Die disjunkten Raumauflösungsverfahren teilen *den Raum* in sich nicht überlappende Bereiche, im Folgenden *Voxel* genannt. Objekte werden den Voxeln, die sie überschneiden zugeordnet. Dies hat zur Folge, dass Objekte, die mehrere Voxel schneiden, auch

mehrmals referenziert werden. Da für jede Referenz Speicher benötigt wird, aber im voraus nicht bestimmt werden kann wieviele Voxel ein Objekt überlappt, kann für diese Art von Datenstruktur im Voraus keine genaue Abschätzung über die Größe des von ihr belegten Speichers gemacht werden.

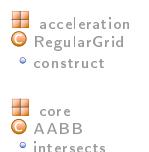
Von der disjunkten Partitionierung des Raums kann profitiert werden, indem nur diejenigen Objekte mit dem Strahl auf Schnitt getestet werden, die ganz oder teilweise in Voxeln liegen, die auch von dem Strahl geschnitten werden. Da ein Objekt in mehreren Voxeln referenziert werden kann, ist es möglich, dass ein Strahl zweimal mit dem gleichen Objekt auf Schnitt getestet wird. Um diese unnötigen Tests zu verhindern, kann man das so genannte *Mailboxing* (Benthin (2006), Glassner (1989)) anwenden. Hier wird für jedes Objekt gespeichert von welchem Strahl es als letztes geschnitten wurde.

3.1.1 Gleichmäßige Gitter

Ein gleichmäßiges Gitter teilt die AABB der Szene entlang der drei Hauptachsen in gleichmäßige Abschnitte auf. Dadurch entsteht ein dreidimensionales Feld aus gleichgroßen Voxeln. Im nächsten Schritt müssen die Objekte den Voxeln zugeordnet werden, die sie überlappen.

Konstruktion

Ein Schnitttest jeden Voxels mit allen Objekten ist nicht effizient. Daher wird meist zunächst über die AABB eines Objektes bestimmt, welche Voxel potentiell geschnitten werden können. Eine Klassifizierung ausschließlich auf der Basis der AABBs ist zwar schnell, führt aber dazu, dass Objekte Voxeln zugeordnet werden, welche sie gar nicht schneiden (siehe Abbildung 3.1). Dies kann zu einer unnötig hohen Anzahl von Objekt-Strahl-Schnitttests führen, weswegen es sich lohnt für die ermittelten Voxel einen genaueren Schnitttest mit dem Objekt durchzuführen. Ein effizienter Algorithmus, um zum Beispiel ein Dreieck auf Überschneidung mit einer AABB zu testen, wurde von Akenine-Möller (2001) entwickelt, wird hier aber nicht näher vorgestellt.



Effiziente Traversierung

Für die Traversierung sucht man nun die Voxel, die vom Strahl geschnitten werden. Danach werden lediglich diejenigen Objekte der Szene mit dem Strahl auf Schnitt getestet, die in den gefundenen Voxeln referenziert werden.

Durch die disjunkte und gleichmäßige Anordnung müssen unter Umständen nicht die Objekte *aller* gefundenen Voxel gegen den Strahl getestet werden. Ein Objekt B das

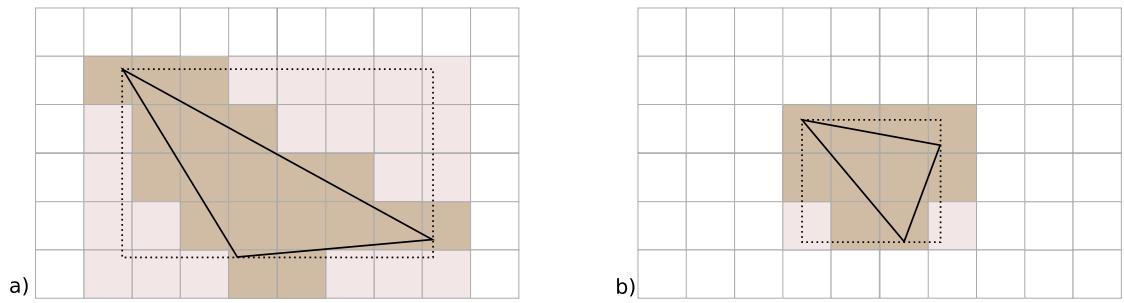


Abbildung 3.1: Fehler bei der Objektklassifizierung für regelmäßiges Gitter durch die AABBs a) Einordnung lediglich aufgrund der AABB für bei, relativ großen Objekten zu mehr falschen Zuordnungen als bei b) relativ kleinen Objekten

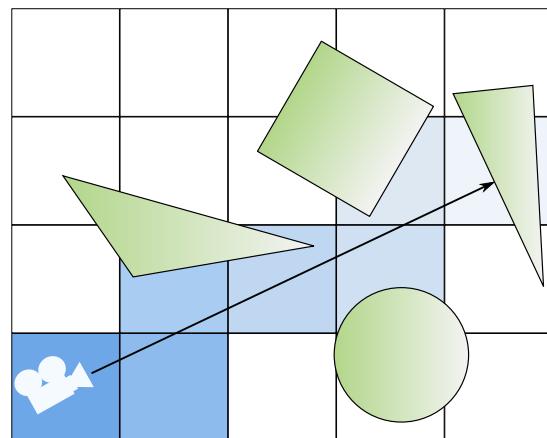


Abbildung 3.2: Effiziente Gittertraversierung

vom Strahlursprung aus gesehen hinter einem Objekt A liegt kann nicht einen näheren Schnittpunkt mit dem Strahl besitzen als Objekt A. Deswegen wird ein Algorithmus verwendet, der dem Bresenham-Algorithmus zum Zeichnen von Linien im 2D-Raum sehr ähnlich ist. Dieser von [Amanatides u. Woo \(1987\)](#) beschriebene Ansatz geht dabei wie folgt vor (siehe auch Abbildung 3.2):

1. Bestimme den Voxel in dem der Ursprung des aktuellen Strahlabschnitts liegt (oder durch den der Strahl in das Gitter eintritt, wenn der Strahlursprung außerhalb des Gitters liegt)
2. Teste den Strahl mit allen, in dem gefundenen Voxel referenzierten Objekten auf Schnitt
3. Wurde hier ein Schnitt gefunden, kann das Verfahren terminiert werden, da potentielle weitere Schnitte entlang des Strahls nur hinter dem gefundenen liegen können.
4. Bestimme denjenigen Nachbarvoxel in den der Strahl als nächstes eintritt, und nimm den Eintrittspunkt als neuen Ursprung für den Strahl an.
5. fahre mit 2. für den Nachbarvoxel fort

Für Strahlsegmente, die außerhalb der AABB der Szene ihren Ursprung haben, muss zunächst der Eintrittspunkt des Strahls in das Gitter berechnet werden. Da vorher jedoch bereits der in Abschnitt 3 erwähnte Test durchgeführt wurde, um festzustellen ob der Strahl die AABB der Szene überhaupt schneidet, kann der Eintrittspunkt durch Einsetzen von t_{min} als t in Gleichung 2.1 leicht gefunden werden.

Der Voxel, in welchem sich ein Punkt im Gitter befindet, lässt sich durch ganzzahlige Division der Komponenten durch die jeweilige Breite eines Voxels bestimmen. Um herauszufinden, in welchen Voxel der Strahl als nächstes eintritt, muss zunächst bestimmt werden, ob sich der Strahl in jeder der drei Dimensionen in positiver oder in negativer Richtung ausbreitet. Da sich die Richtung eines Strahls während der Ausbreitung nicht ändert, genügt es einmal zu Beginn der Traversierung jeden Strahls die Vorzeichen der entsprechenden Komponente des Richtungsvektors zu betrachten. Aus den Ausbreitungsrichtungen ergeben sich genau drei Voxel, in die der Strahl als nächstes eintreten kann. Das Wissen über den Index des aktuellen Voxels und den Ursprung des aktuellen Strahlabschnitts, erlaubt es die Entfernung zu den nächsten drei Voxeln wie folgt zu berechnen: $\Delta x = x_{next} - x_{current}$ (äquivalent für y und z). Durch Division der Entfernung durch die entsprechende Komponente des Richtungsvektors des Strahls, lässt sich

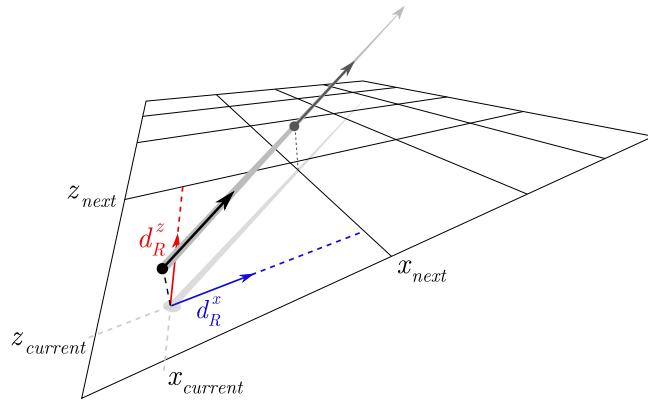


Abbildung 3.3: Bestimmung des nächsten Voxels in Strahlrichtung. Die Y-Achse wird hier vernachlässigt. Deswegen ergeben sich lediglich zwei potentielle nächste Voxel.

das kleinste t aller drei Achsen bestimmen, für das der Strahl in einen weiteren Voxel eintritt: $t_x = \frac{\Delta x}{d_R^x}$ (siehe Abbildung 3.3)

■ acceleration
● RegularGrid
○ getFirstIntersection

Nachteile

Durch die Verwendung von Gittern mit effizienter Traversierung können gegenüber dem naiven Ansatz zwar bereits viele unnötige Schnitttests vermieden werden, dennoch zeigt das Verfahren in gewissen Konfigurationen signifikante Schwächen. Gerade in kargen Szenen muss der Algorithmus viele leere Zellen traversieren und dafür die oben beschriebenen Rechenschritte ausführen. Wird, um dem entgegenzuwirken, die Auflösung des Gitters verringert steigt die Wahrscheinlichkeit, dass Anhäufungen von kleinen Objekten komplett in einzelne Voxel fallen. Damit müssen Strahlen, welche durch diesen Voxel verlaufen, gegen sehr viele Objekte auf Schnitt getestet werden, was ja gerade verhindert werden soll. Leider enthalten realistische Szenen oft viel leeren Raum, da die reale Welt vom Menschen meist so eingerichtet, dass er sich frei und bequem bewegen kann - was entsprechende freie Räume erfordert. Die genannten Phänomene werden auch das *Teapot-in-Stadium*-Problem genannt (Abbildung 3.4).

3.1.2 Octrees und hierarchische Gitter

Das Problem bei dem zuvor beschriebenen Ansatz ist, dass leerer Raum genauso *fein* aufgeteilt werden muss, wie Raumbereiche in denen sich viele kleine Objekte befinden. Um dem entgegenzuwirken kann der Raum zunächst in ein grobes Gitter aufgeteilt werden. Die Objekte der Szene werden, wie zuvor beschrieben, den zugehörigen Voxeln zugeord-

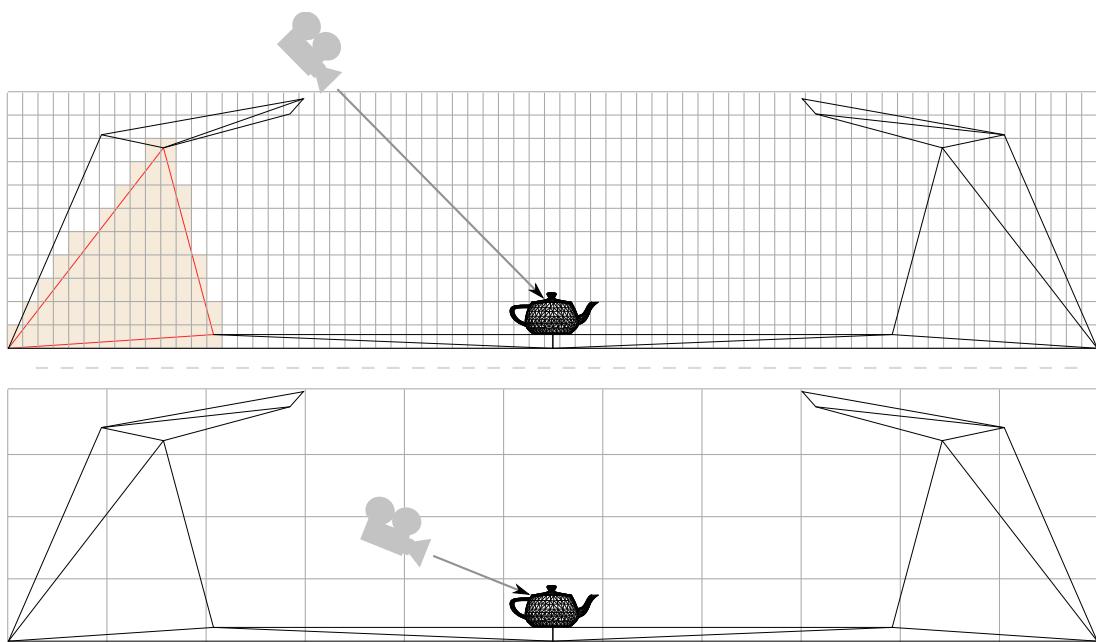


Abbildung 3.4: oben: Die feine Auflösung führt dazu, dass sehr viele leere Voxel traversiert werden müssen. Außerdem werden große Objekte von sehr vielen Voxeln referenziert, was zu enormen Speicherplatzbedarf führt. unten: Eine niedrige Auflösung führt dazu, dass viele Objekte in einzelnen Zellen liegen.

net. An leeren Voxeln wird der Raum nicht weiter unterteilt. In Voxeln denen Objekte zugeordnet wurden, werden erneut, feinere Gitter eingefügt. Der Zuordnungsvorgang wird für die Objekte in diesem Voxel und die neu entstandenen Voxel des Subgrids wiederholt. Dennoch muss die Frage nach der Wahl der Auflösung auch hier beantwortet werden. Dazu wurden in verschiedenen Arbeiten entsprechende Heuristiken entwickelt. Die Qualität dieser Verfahren hängt jedoch oft von der Szene und/oder von manuell zu wählenden Parametern ab, die mit guten *Erfahrungswerten* belegt werden müssen.

Eine automatische Partitionierung der Szene wird durch die Verwendung von Octrees erreicht. Diese werden in [Glassner \(1988\)](#) erstmals für die Verwendung mit Raytracing vorgeschlagen. Im Prinzip sind die Octrees eine Untermenge der rekursiven Gitter. Jedes Gitter teilt den Raum in jeder Dimension in zwei Teile. Das heißt die AABB der Szene wird in ein Gitter aus acht Würfeln geteilt. Dann wird wie bei den rekursiven Gittern fortgefahren, indem jede nicht leere Gitterzelle in weitere acht Kuben unterteilt wird.

Die Traversierung verläuft ähnlich wie bei den nicht-hierarchischen Gittern. Jedoch müssen im Gegensatz zu den nicht-hierarchischen Gittern zunächst alle relevanten Voxel der Kindgitter getestet werden, bevor zu einem Nachbarvoxel fortgeschritten werden darf. Ein Vorteil der gleichmäßigen Unterteilung der Octrees ist, dass in einem beliebigen Traversalsschritt aus der Rekursionsstufe direkt auf die Größe eines Voxels geschlossen werden kann.

3.1.3 BSP- & Kd-Trees

Eine weitere Verfeinerung der Raumaufteilung kann durch die Verwendung von binären Bäumen erreicht werden. Der Raum wird durch jeden der inneren Knoten des Baumes in lediglich zwei Voxel geteilt. Diese müssen, im Gegensatz zu Gittern, nicht die selbe Größe besitzen. Binary Space Partitioning(BSP)-Trees werden das erste Mal für die Computergrafik in [Fuchs u. a. \(1980\)](#) verwendet. Die dort ausschließlich für Polygone entwickelte Version lässt sich aber einfach für beliebige Primitive erweitern. Durch die Tatsache, dass die Trennebene beliebig gewählt werden kann, sind die BSP-Trees in der Lage sich besser an die Geometrie anzupassen als gleichmäßige Verfahren.

Für das Raytracing hat sich ein Spezialfall der BSP-Trees als besonders effizient herausgestellt: die so genannten Kd-Trees (auch axis-aligned oder rectilinear BSP-Trees). Sie grenzen sich von den allgemeinen BSP-Trees durch die Bedingung ab, dass die trennende Ebene in einem inneren Knoten immer orthogonal zu einer der drei Hauptachsen sein muss. Neben einfacherer Konstruktion und Traversierung lässt sich der Baum auch kompakter als ein beliebiger BSP-Tree speichern. Für einen inneren Knoten müssen, ne-

Quelltext 3.1: Beschreibung der Kd-Tree Datenstruktur in erweiterter Backus-Naur-Form

```

KdTree   = InnerNode | Leaf
InnerNode = Axis Position KdTree KdTree
Axis     = 'x' | 'y' | 'z'
Position  = float
Leaf      = { Primitiv }

```

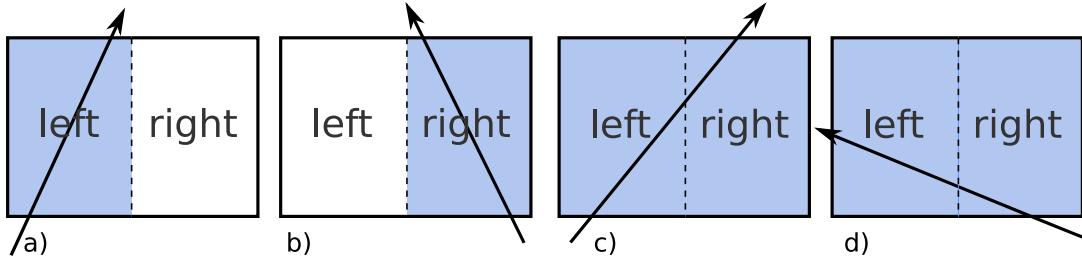


Abbildung 3.5: a) nur der linke Kindknoten wird geschnitten und muss besucht werden, b) nur der rechte Kindknoten wird geschnitten, c) beide Knoten müssen besucht werden, der linke wird zuerst geschnitten und muss daher zuerst besucht werden, d) beide Knoten werden geschnitten, der rechte Knoten muss als erstes besucht werden.

ben den Referenzen auf die Kindknoten, lediglich die Achse x, y, z und ein Skalar - die Position der Ebene auf der Achse - gespeichert werden müssen. Die Blätter des Baumes referenzieren jeweils die Liste der Objekte die den Voxel überlappen, der sich aus der Raumteilung der Vorgängerknoten ergibt. Ein Kd-Tree lässt sich wie in Quelltext 3.1 definieren.

Traversierung

Ein Grund warum binäre Bäume sich so großer Beliebtheit erfreuen ist, dass die Traversierung einfacher ist als bei mehrstelligen Bäumen oder hierarchischen Gittern, was eine effiziente Implementierung begünstigt. Im Gegensatz zu Octrees, wo bis zu vier der acht Kindvoxel (in verschiedenen Reihenfolgen) von einem Strahl geschnitten werden können, tritt beim Kd-Tree lediglich einer von vier möglichen Fällen pro Knoten auf (siehe Abbildung 3.5):

1. Es wird nur der vordere Voxel geschnitten
2. Es wird nur der hintere Voxel geschnitten
3. Es wird zuerst der hintere und dann der vordere Voxel geschnitten

4. Es wird zuerst der vordere und dann der hintere Voxel geschnitten

Zur Beantwortung der Strahlanfragen müssen die Knoten des Baums, von der Wurzel an beginnend, rekursiv besucht werden. Erreicht der Algorithmus ein Blatt, wird der Strahl mit allen im Blatt enthaltenen Objekten auf Schnitt getestet. Für einen inneren Knoten werden lediglich die Kindknoten besucht, deren Voxel vom Strahl geschnitten wird. Für den Fall, dass beide Knoten vom Strahl geschnitten werden, ergibt sich die Traversierungsreihenfolge der Kindknoten aus der Reihenfolge in der der Strahl die Voxel der Kindknoten schneidet. Bei Beachtung der Traversierungsreihenfolge kann der Algorithmus bereits nach dem Besuch des ersten Knotens terminiert werden, wenn ein Schnittpunkt mit einem Objekt aus diesem Teilbaum gefunden wurde. Durch die disjunkte räumliche Anordnung der Voxel ist garantiert, dass jeder weitere potentielle Schnitt, der im Voxel des *anderen* Kindknotens liegt, hinter dem bereits gefundenen liegen muss.

Für die Bestimmung der Traversierungsreihenfolge genügt es das Vorzeichen einer Komponente des Richtungsvektors zu betrachten. Die interessante Komponente ist die, entlang welcher der Voxel des aktuellen Knotens geteilt wird. Der Teilbaum, der als erstes traversiert werden muss, wird dabei als *nah* und der zweite als *ferner* Teilbaum bezeichnet. Ob der nahe Teilbaum überhaupt besucht werden muss, lässt sich über die relative Position des Strahlursprungs zur Trennebene bestimmen: Liegt er in Ausbreitungsrichtung hinter der Trennebene, kann der nahe Teilbaum übersprungen werden. Der ferne Teilbaum muss nur dann nicht traversiert werden, wenn der Strahl diesen außerhalb des Strahlsegments schneidet. Dafür wird die parametrische Distanz des Ursprungs zur Trennebene mit dem Parameter t_{max} des Strahlsegments verglichen. Quellcode 3.2 zeigt das beschriebenen Vorgehen bei der Traversierung in Pseudocode.

Die frühe Terminierung der Traversierung darf allerdings nur dann vorgenommen werden, wenn der gefundenen Schnittpunkt innerhalb des Voxels des aktuell traversierten Teilbaums liegt. Befindet sich der Schnittpunkt außerhalb des aktuellen Voxels, können anderenfalls potentielle, nähere Schnittpunkte nicht gefunden werden (siehe Abbildung 3.6). Kritisch sind dabei die Fälle, in denen der bei der Traversierung des *vorderen* Teilbaumes gefundene Schnitt, tatsächlich im Voxel des *hinteren* Teilbaums liegt. Dieses lässt sich leicht feststellen, indem man den Abstand des Schnittpunktes zum Strahlursprung mit dem Abstand zur Trennungsebene vergleicht.

Wie bereits in Abschnitt 3.1 angedeutet führt das dazu, dass ein Objekt mehrmals mit dem gleichen Strahl auf Schnitt getestet werden kann, was gerade bei komplexen Objekten vermieden werden sollte. Beim Mailboxing, wie in Benthin (2006) beschrieben, wird jedem Strahl eine eindeutige Identifikationsnummer(ID) zugewiesen. Bei dem Schnitttest mit

 acceleration
 KdTreeSimple
 getFirstIntersection

Quelltext 3.2: Traversierung eines Kd-Trees

```

Intersection getIntersection (KdTree node, Ray ray){
    if ( node.isLeaf() ) {
        return node.primitives.intersect (ray);
    } else {
        KdTree near, far;
        if ( r. direction [node.axis] > 0.0 )
            near = left ; far = right;
        else
            near = right; far = left ;
    }
    d = (node.splitPos - ray.origin .[ node.axis]) / ray. direction [ node.axis];
    Intersection result ; // wird leer initialisiert
    if ( d > 0 ) // der 'nahe' Knoten muss besucht werden
        result = getIntersection(near, ray);
    if ( ! result.isEmpty() ) // ggf. fruehe Terminierung
        return result;
    if ( d < ray.tmax ) // nur falls 'ferner' Voxel geschnitten wird
        return getIntersection( far, ray);
    return result;
}

```

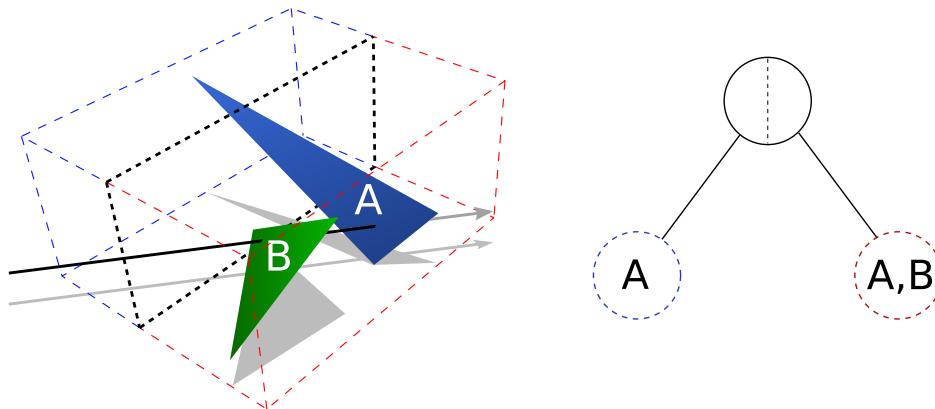


Abbildung 3.6: links: Der blaue Voxel wird als erstes getestet, da er als erstes vom Strahl geschnitten wird. Es wird zwar ein Schnittpunkt mit Dreieck A gefunden, das Verfahren darf aber nicht terminiert werden, da der Schnittpunkt nicht innerhalb des blauen Voxels liegt - denn der rote Voxel enthält noch einen Schnittpunkt mit B der näher am Ursprung liegt. rechts: Der Baum zu der Geometrie besteht aus der Wurzel die die gesamte AABB in den blauen Voxel (der lediglich A enthält) und den roten Voxel (der A und B) teilt.

Quelltext 3.3: Eine einfache Version eines Konstruktionsalgorithmus für kd-Trees

```
KdTree buildKdTree( List<Primitive> primitives, AABB voxel ) {
    if ( terminationCriterionMet() ) {
        return Leaf( primitives );
    } else {
        ( splitPosition , splitAxis ) = heuristic . getSplit ( primitives , voxel );
        ( Vleft, Vright ) = voxel.split( splitPosition, splitAxis );
        List<Primitive> Pleft, Pright;
        foreach ( primitive in primitives ) {
            if ( primitive.overlaps( Vleft ) )
                Pleft.add( object );
            if ( primitive.overlaps( Vright ) )
                Pright.add( object );
        }
        return InnerNode ( splitAxis, splitPosition ,
                           buildKdTree(left, Vleft),
                           buildKdTree(right, Vright) )
    }
}
```

einem Objekt, speichert das Objekt die Strahl-ID. So kann bei einem wiederholten Test festgestellt werden, dass der Schnittpunkt bereits berechnet wurde, und muss so nicht wiederholt werden.

Konstruktion

Bei der top-down Konstruktion eines kd-Trees wird die AABB der Szene durch eine achsenparallele Trennebene in zwei Subvoxel unterteilt. Die Objekte werden dann dem linken, rechten oder beiden Subvoxeln zugeordnet, je nachdem welchen der Subvoxel ein Objekt überlappt. Ein Abbruchkriterium ist eine minimale Anzahl an Objekten für die ein Voxel nicht weiter unterteilt werden soll. Um den Speicherbedarf zu begrenzen wird meistens zusätzlich eine maximale Rekursionstiefe als Abbruchbedingung festgelegt.

Quelltext 3.3 zeigt einen rekursiven Algorithmus zur Konstruktion eines Kd-Trees.

Die Wahl der Achse und Position der Schnittebene wird an eine *Heuristik* delegiert. Eine der einfachsten Heuristiken für diesen Zweck wählt immer den Mittelpunkt der längsten Achse der AABB. Das führt zu einer ähnlichen Raumaufteilung wie bei den Octrees. Diese Raumaufteilung nutzt nicht die Fähigkeit der BSP-Trees sich an die Szenegeometrie anzupassen und unterliegt deswegen Heuristiken, die dies ausnutzen. Diese Beobachtung wird durch die statistische Untersuchungen in [Havran \(2000\)](#) bestätigt.

Um die Objekte bei der Konstruktion in den jeweils linken oder rechten Teilbaum



einzuordnen, muss bestimmt werden welche Voxel der Kindknoten von dem Objekt geschnitten werden. Man kann sich hierbei auf eine Betrachtung der AABBs der Objekte beschränken, um einen aufwändigen Objekt-AABB-Schnitttest zu vermeiden. Dies führt aber, wie bereits bei den Gittern in 3.1.1, dazu, dass Objekte auch in Voxel eingesortiert werden, die zwar die AABB des Objekts, aber nicht vom Objekt selbst geschnitten werden. Neben dem erhöhten Speicherplatzbedarf führt dies auch zu unnötigen Strahl-Objekt-Schnitteests in der Traversierungsphase.

Für Dreiecke und AABBs gibt es zahlreiche Schnitttestalgorithmen. Für die Tests die zur Klassifizierung bei der Konstruktion eines Kd-Trees benötigt werden ist es jedoch möglich einen weniger allgemeinen, aber dafür effizienteren Algorithmus anzuwenden wie er von Wächter (2008) beschrieben wird. Dabei werden folgende Annahmen ausgenutzt:

- Ein Dreieck überlappt immer den aktuellen Voxel (zumindest teilweise)
- Ein Dreieck, welches den aktuellen Voxel überlappt muss auch mindestens einen der beiden Subvoxel überlappen.
- Wenn die AABB eines Dreiecks komplett auf einer Seite der Trennebene liegt, kann es nur den auf dieser Seite liegenden Voxel überlappen.
- Wenn das Dreieck genau in (parallel zu, und an der Stelle) der Trennebene liegt muss eine Heuristik entscheiden in welche der beiden Voxel das Dreieck eingesortiert wird.

Der Algorithmus geht dann wie folgt vor. Es werden zunächst zwei Trivialfälle getestet

1. Prüfe ob das Dreieck in der Trennebene liegt
2. Prüfe ob die AABB des Dreiecks vollständig vor oder hinter der Trennebene liegt.

Falls einer der beiden Fälle zutrifft, muss das Dreieck direkt in einen der Teilbäume eingesortiert werden. Andernfalls muss festgestellt werden, ob das Dreieck den Querschnitt des Voxels, an der Stelle der Trennebene (A_{split}), schneidet. Dazu werden die folgenden Schritte ausgeführt:

1. Bestimme den Vertex des Dreiecks der sich *alleine* auf einer Seite der Trennebene befindet.
2. Über die Entfernung d zur Trennebene (siehe Abbildung 3.7) lassen sich die beiden Schnittpunkte S_1, S_2 des Dreiecks mit der (unbegrenzten) Trennebene bestimmen.

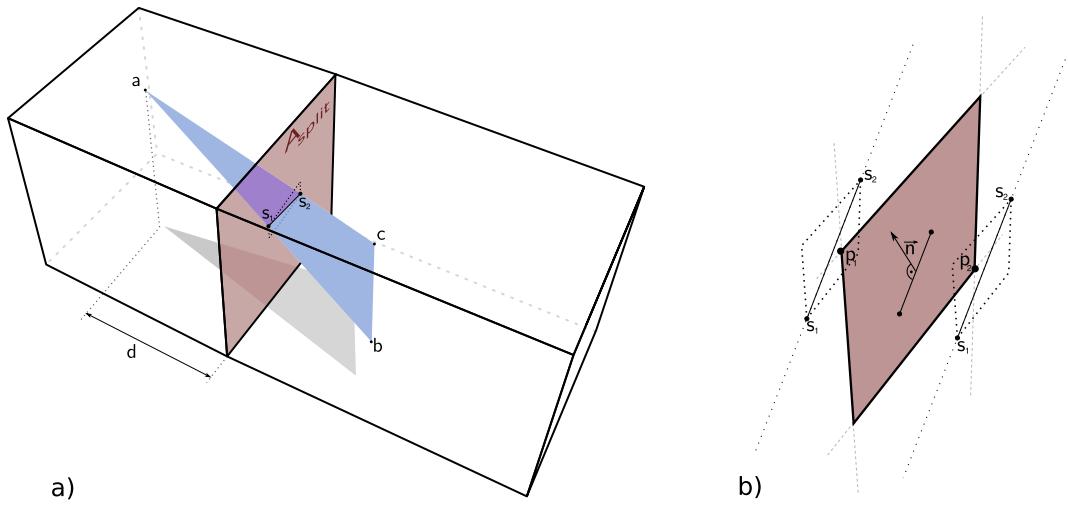


Abbildung 3.7: a) Das Dreieck muss beiden Voxeln zugeordnet werden, da A_{split} von $S_1 - S_2$ geschnitten wird. b) Für die äußeren beiden Fälle schneidet die Gerade das Rechteck nicht, somit kann auch $S_1 - S_2$ das Rechteck nicht schneiden.

3. Falls das achsenparallele Viereck dessen Diagonale die Strecke $S_1 - S_2$ darstellt, den Querschnitt des Voxels (A_{split}) nicht schneidet, kann das Dreieck sofort *einem* der beiden Voxel zugeordnet werden.
4. Andernfalls wird zunächst eine Senkrechte \vec{n} zu $S_1 - S_2$ bestimmt. Über die Komponenten von n lässt sich die Ecke p_1 des Rechtecks A_{split} bestimmen, welche am weitesten von $S_1 - S_2$ entfernt ist. Falls die Gerade auf der $S_1 - S_2$ liegt zwischen dieser und der gegenüberliegenden Ecke liegt **muss** $S_1 - S_2$ das Rechteck schneiden und muss in beide Voxel eingesortiert werden. Andernfalls wird das Dreieck wieder nur in einen Voxel eingesortiert. Zur Bestimmung ob die beiden Ecken auf der selben Seite der Geraden liegen genügt es die Vorzeichen der folgenden Produkte: $(S_1 - p_1) * \vec{n}$ und $(S_1 - p_2) * \vec{n}$.

Surface Area Heuristic

Die Effizienz der Strahlstrahlenfragen an einen Kd-Tree hängt maßgeblich von der Qualität der Raumaufteilung ab, die durch den Baum vorgenommen wird. Globale, regelmäßige Aufteilungen führen zu den bei Gittern erwähnten Problemen. Auch eine Wahl der Schnittebenen bei denen sich ein möglichst balancierter Binärbaum ergibt - also die Platzierung der Trennungsebene am Objektmedian - führt nicht zu einem idealen Baum für Strahlstrahlenfragen. Diese Konstruktion ist zwar für allgemeine Suchbäume effizient, geht

aber davon aus, dass alle Knoten gleichhäufig besucht werden. Da für große Voxel die Wahrscheinlichkeit, dass sie von einem Strahl getroffen werden aber höher ist, ist diese Annahme so nicht gültig. Weiter wird im Gegensatz zu Suchen auf allgemeinen binären Suchbäumen auch nicht genau ein Pfad von der Wurzel zu dem gesuchten Blatt besucht. Vielmehr müssen unter Umständen an mehreren inneren Knoten beide Teilbäume interpretiert werden, wenn im ersten Teilbaum kein Schnittpunkte gefunden wurde. (Wald, 2004)

Die Heuristiken welche die beste Raumaufteilung liefern, berechnen die Güte einer potentiellen Trennungsebene mit Hilfe einer Kostenfunktion (Havran (2000)). Die bekannteste Heuristik, die eine solche Kostenfunktion zur Verfügung stellt, ist die Surface Area Heuristic(SAH). Ein guter Baum ermittelt den gesuchten Schnittpunkt beziehungsweise, dass es keinen Schnittpunkt gibt, für einen beliebigen Strahl mit möglichst wenig Traversierungsschritten und möglichst wenig Schnitttests. Für die SAH wird angenommen, dass Strahlen sich gleichverteilt im Raum befinden und in ihrem Verlauf nicht durch Objekte unterbrochen werden. Aus dieser Vereinfachung lässt sich eine Beziehung zwischen der Oberfläche eines Voxels und der Wahrscheinlichkeit, dass dieser von einem Strahl getroffen wird, herstellen: Je größer die Oberfläche eines Voxels ist, desto wahrscheinlicher ist es, dass dieser von einem Strahl getroffen wird. Ein guter Baum enthält möglichst viele, möglichst große, leere Zellen - und diese möglichst nah an der Wurzel.

Um die Kostenfunktion für die Positionierung der Trennebene eines Voxels aufstellen zu können trifft die SAH folgende Annahmen:

- Strahlen sind gleichmäßig im Raum verteilt
- Strahlen sind Geraden(ohne Start- und Endpunkt) und werden nicht unterbrochen
- Die Kosten für den Schnitt mit einem Objekt C_i und die Kosten für einen Traversierungsschritt C_t sind bekannt

Durch diese Annahmen lässt sich aus der Oberfläche $A_{V_{sub}}$ eines Teilvoxels $V_{sub} \subset V$ auf die Wahrscheinlichkeit P schließen, dass von ein Strahl der V trifft auch V_{sub} trifft 3.1

$$P(V, V_{sub}) = \frac{A_{V_{sub}}}{A_V} \quad (3.1)$$

Für die Kostenfunktion wird zunächst der Trivialfall betrachtet: Wie hoch sind die Kosten für einen Strahl der die AABB der Szene trifft wenn der Wurzelknoten eines Kd-Trees direkt ein Blatt ist, und demnach alle Primitive enthält? Dies entspricht der naiven Lösung aus 2.5, denn jedes Objekt muss gegen den Strahl getestet werden. Die Kosten sind demnach $C_{leaf} = N * C_i$, wobei N der Anzahl der Objekte entspricht.

Nach Bestimmung der Position der Trennebene für einen Voxels werden die Objekte den entstehenden zwei Subvoxeln zugeordnet. Die Kosten die sich für die Schnitttests ergeben werden mit den jeweiligen Trefferwahrscheinlichkeiten der Subvoxel gewichtet. Das heißt eine Trennung, bei der ein Voxel mit sehr wenigen Objekten, mit einer hohen Wahrscheinlichkeit getroffen wird, ergibt einen sehr *günstigen* Knoten. Zusätzlich kommen dann die Kosten für den Traversierungsschritt hinzu, die im Verhältnis zu einem Objekt-Strahlschnittest aber sehr niedrig sind. Die Kosten C für die Trennung eines Voxels V durch die Ebene q lassen sich dann wie folgt in Gleichung 3.2 bestimmen:

$$C_V(q) = C_t + P(V, V_{left}) * C_{left} * C_i + P(V, V_{right}) * C_{right} * C_i \quad (3.2)$$

Für eine gute Raumaufteilung muss ein möglichst günstiger Baum gefunden werden. Um die Kosten für einen kompletten Baum zu berechnen müssen die Positionen aller Trennebenen bekannt sein. Denn nur so lässt sich P , und damit die Kosten, für die tiefer liegenden Voxel bestimmen. Dies steht jedoch im Widerspruch dazu, dass die Kostenfunktion bereits in den oberen Ebenen des Baumes genutzt werden soll um eben die beste Position für die Trennebenen zu finden.

Um dennoch eine Abschätzung vornehmen zu können, nimmt man für eine potentielle Trennebene an, dass die entstehenden Subvoxel nicht weiter unterteilt werden. Diese, sehr grobe, Abschätzung überschätzt die tatsächlich für die Subvoxel entstehenden Kosten, da diese mit hoher Wahrscheinlichkeit weiter unterteilt werden - und damit günstiger werden. Die Vorgehensweise hat sich in der Praxis aber dennoch als wirksames Indiz für gute Trennebenen erwiesen.

Beim Vergleich zweier potentieller Trennebenen muss mehrmals Gleichung 3.1 für die Kostenbestimmung ausgewertet werden. Da alle potentiellen Ebenen eines Knotens den gleichen Voxel teilen, und lediglich das relative Verhältnis der Kosten zueinander interessant ist, kann auf die Division durch A_V verzichtet werden. Eine weitere Vereinfachung der Berechnung der Kosten lässt sich erreichen wenn es nur einen Typ von Objekten (zum Beispiel Dreiecke) gibt. In diesem Fall kann C_i durch 1 ersetzt werden. So ergibt sich zur Berechnung der relativen Kosten in einem Knoten Gleichung 3.3.

$$C_{Vrel}(q) = C_t + V_{left} * N_{left} + V_{right} * N_{right} \quad (3.3)$$

Für plane Objekte kann zusätzlich der Sonderfall eintreten, dass sie *in* der Trennebene liegen. Sie müssen aber trotzdem einen der beiden Teilbäume zugeordnet werden. Eine Zuordnung in beide Teilbäume ist zwar nicht falsch, führt aber zu weniger effizienten Strahlanfragen. (Wald u. Havran, 2006) empfehlen diesen Fall extra zu verfolgen und

anschließend zu untersuchen ob einer der Teilbäume keine Objekte enthält. Plane Objekte die in der Trennebene liegen werden dann dem nicht-leeren Teilbaum zugeordnet.

Zusätzliches Terminierungskriterium

Neben der Positionierung der Trennebene lässt sich die SAH zusätzlich zur Einführung einer besseren Abbruchbedingung für die rekursive Konstruktion einsetzen. Durch Gleichung 3.2 lassen sich die Kosten für den Fall berechnen, dass der aktuelle Voxel nicht weiter unterteilt wird. Sind die Kosten, die durch die Trennung mit der günstigsten Trennebene entstehen, höher als den Knoten nicht zu teilen, lohnt es sich nicht die Unterteilung fortzusetzen. Da die lokale Bestimmung dazu tendiert die Kosten für die entstehenden Subvoxel zu überschätzen kann es erforderlich sein bei dem Vergleich einen entsprechenden Faktor einzubeziehen um ein vorzeitiges Terminieren zu verhindern.

Kandidatengenerierung für Trennebenen

Da es auf einer Achse unendlich viele mögliche Positionen für eine Trennebene gibt, muss für die Kandidaten eine sinnvolle Untermenge gewählt werden. Für diese kann dann die Kostenfunktion der SAH ausgewertet und der günstigste Kandidat verwendet werden. Bei Betrachtung der Komponenten der Kostenfunktion (siehe Abbildung 3.8) stellt sich heraus, dass das gesuchte Minimum nur an bestimmten Stellen liegen kann. Diese befinden sich jeweils an den Minima beziehungsweise Maxima der AABBs der Objekte (für Objekte die teilweise außerhalb des Voxels liegen trifft dies auch für die Schnittpunkte der Objektkanten mit dem Voxel zu: siehe Abbildung 3.9). Zwischen zwei benachbarten Stellen kann kein Minimum der Kostenfunktion liegen (Wald u. Havran, 2006).

Um die beste Position für die Trennebene zu finden, müssen also zunächst die Grenzen der AABBs der Objekte entlang jeder Achse gesammelt werden. Danach muss für jede der gefundenen Stellen die Kostenfunktion der SAH ausgewertet werden. Dabei wird verfolgt welche Position die bisher niedrigsten Kosten verursacht hat (siehe auch Quelltext 3.4).



3.2 Partitionierung der Objektmenge

Grundlegend anders als die disjunkten Raumaufteilungsverfahren wird bei der Anwendung von Bounding Volume Hierarchies(BVH) vorgegangen. Hier wird anstatt die AABB der Szene, die Menge der Objekte partitioniert.

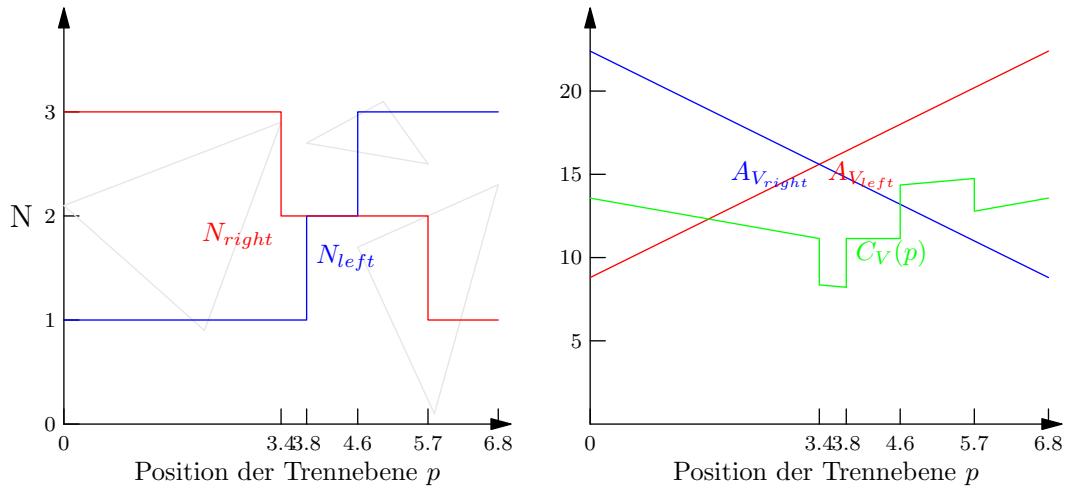


Abbildung 3.8: links: Anzahl der Objekte im rechten bzw. linken Subvoxel bei entsprechender Positionierung der Schnittebene. rechts: Die Oberflächen (bzw. hier Umfänge, da 2D) der Subvoxel bei entspr. Trennung. In grün die lokale Kostenfunktion

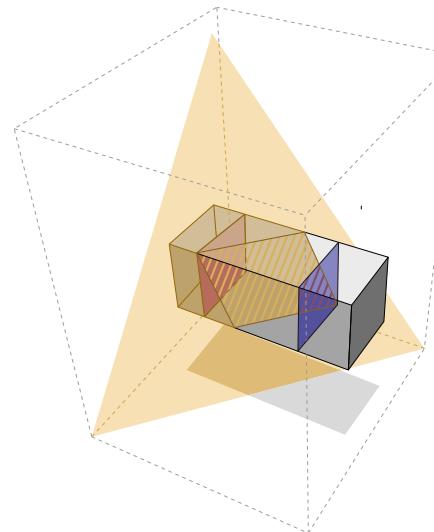


Abbildung 3.9: Die Minima/Maxima der Objekte als Trennkandidaten zu verwenden reicht nicht aus. Die rote und blaue Position für Schnittebenen sind wichtige Kandidaten die an den Minima/Maxima der Schnittkanten des Objektes mit dem Voxel zu finden sind. Da der Voxel komplett in der AABB des Dreiecks liegt würde in diesem Fall sonst gar kein Kandidat gefunden.

Quelltext 3.4: Suche nach der besten Trennebene in $O(n^2)$

```
(float, Axis) searchBestSplitPos(AABB voxel, List<Primitive> primitives) {
    List<float> splitCandidates[{x,y,z}];
    foreach (primitive in primitives) {
        AABB clippedBox = primitive.clipTo(voxel);
        foreach (axis in {x,y,z}) {
            splitCandidates[axis].add(clippedBox.min[axis]);
            splitCandidates[axis].add(clippedBox.max[axis]);
        }
    }

    float bestCost = infinity;
    float bestPos;
    Axis bestAxis;
    foreach (axis in {x,y,z}) {
        foreach (candidatePosition in splitCandidates) {
            List<Primitive> N_left, N_right;
            foreach (primitive in primitives) {
                if ( primitive.AABB.min[axis] < candidatePosition )
                    ++N_left;
                if ( primitive.AABB.max[axis] >= candidatePosition )
                    ++N_right;
            }
            (V_left, V_right) = voxel.split( axis, candidatePosition );
            float currentCost = C_t + V_left.surface() * N_left + V_right.surface() * N_right;
            if ( currentCost < bestCost ) {
                bestCost = currentCost;
                bestPos = candidatePosition;
                bestAxis = axis;
            }
        }
    }
    return (bestPos, bestAxis);
}
```

Quelltext 3.5: Beschreibung einer binären BVH Datenstruktur in erweiterter Backus-Naur-Form

```

BVH      = InnerNode | Leaf
InnerNode = AABB BVH BVH
Leaf     = AABB { Primitiv }
AABB   = Minima Maxima
Minima   = Vector3
Maxima   = Vector3
Vector3  = float float float

```

Bei der top-down Konstruktion wird die Menge der Objekte in zwei (für binäre Bäume, sonst auch mehr) disjunkte Teilmengen zerlegt. Für jede Teilmenge wird ein neuer Knoten erstellt. Dieser Vorgang wird für jeden neuen Knoten wiederholt bis nur noch ein Objekt in den entstehenden Teilmengen vorhanden ist. Für dieses wird dann ein Blattknoten angelegt. Nach der Erzeugung der Blätter wird für jeden Knoten die gemeinsame AABB der Kindknoten berechnet. Die AABB eines Blattknotens entspricht der AABB des enthaltenen Objekts. Genauso wie bei den kd-Trees kann zugunsten des Speicherplatzbedarfs auch eine höhere Anzahl Objekte gewählt werden, für die ein Blattknoten erzeugt wird.

Eine Eigenschaft durch die sich BVHs positiv von den bisher beschriebenen Strukturen abgrenzen ist, dass jedes Objekt genau einmal referenziert wird. Dadurch kann der Speicherbedarf für die BVH im voraus bestimmt werden. Ein binärer Baum mit N Blättern hat immer genau $2 * N - 1$ Knoten insgesamt. Für eine BVH mit N Objekten für welche jeweils ein Blattknoten erzeugt wird, ergibt sich demnach genau dieser Speicherbedarf. Da jedes Objekt nur einmal referenziert wird, ist auch kein Mailboxing nötig. Jeder Strahl wird mit jedem Objekt maximal einmal auf Schnitt getestet.

Im Gegensatz zu den kd-Trees arbeitet die Traversierung einer BVH nach dem Ausschlussverfahren. Für einen kd-Tree gilt:

Wird ein Voxel von einem Strahl getroffen, muss auch mindestens einer der beiden Subvoxel getroffen werden.

Für die BVH lautet der Schluss hingegen:

Wird die AABB des aktuellen Knotens *nicht* vom Strahl geschnitten, kann der Strahl auch keine AABB eines Kindknotens schneiden.

Die Traversierung wird für also für jeden Knoten fortgesetzt, dessen AABB von dem Strahlsegment geschnitten wird. Die Partitionierung der Objektmenge führt dazu, dass



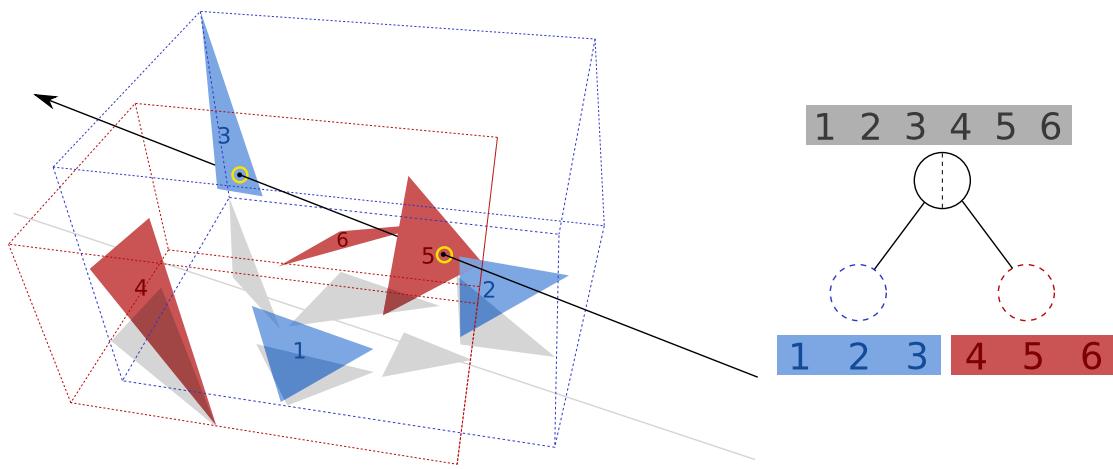


Abbildung 3.10: Wird nach Fund des Schnittpunkts im blauen Teilbaum, der rote Teilbaum nicht mehr besucht, wird der nächste, gesuchte Schnittpunkt überhaupt nicht gefunden.

sich die AABBs zweier Kindknoten überlappen können. Damit ist eine frühe Terminierung bei Fund eines Schnittpunkts, wie beim kd-Tree, nicht möglich. Dies würde in einem Fall, wie er in Abbildung 3.10 dargestellt ist, dazu führen, dass der gesuchte Schnittpunkt nicht gefunden wird.

Wenn in dem Teilbaum dessen AABB zuerst vom Strahl geschnitten wird, ein Schnittpunkt gefunden wurde, muss deshalb auch der andere Teilbaum durchsucht werden. Wird ein Schnittpunkt gefunden kann das Strahlsegment bis zum Schnittpunkt *gekürzt* werden, da potentielle weitere Schnitte die hinter dem bereits gefundenen Schnittpunkt liegen, uninteressant sind. Dadurch muss ein Knoten dessen AABB hinter dem verkürzten Segment des Strahls beginnt, nicht mehr traversiert werden (siehe Abbildung 3.11)

■ acceleration/bhv
● BvhSimple
○ getFirstIntersection

In einem ersten systematischen Vergleich verschiedene Beschleunigungsstrukturen (Havran (2000)) haben die kd-Trees wesentlich besser abgeschnitten als die BVHs. Dies ist zum Teil dadurch zu erklären, dass die frühzeitige Terminierung für BVHs nicht so strikt vorgenommen kann wie bei Verfahren, die den Raum disjunkt aufteilen. Ursache ist also die Überlappung der AABBs benachbarter Knoten. Durch die fehlende Berücksichtigung der räumlichen Verteilung der Objekte bei der zuvor beschriebenen Konstruktion, kann eine beliebig große Überschneidung entstehen. Havran (2000) wendet einen Konstruktionsalgorithmus von Goldsmith u. Salmon (1987) zur Bottom-Up Konstruktion der BVH an. Interessanterweise basiert die Entwicklung der SAH Kostenfunktionen auf Überlegungen aus Goldsmith u. Salmon (1987). Inzwischen haben Wald u. a. (2007) gezeigt, dass mit BVHs, durch die Konstruktion mit der SAH durchaus vergleichbare Ergebnisse zu kd-Tree basierten Systemen erreicht werden können.

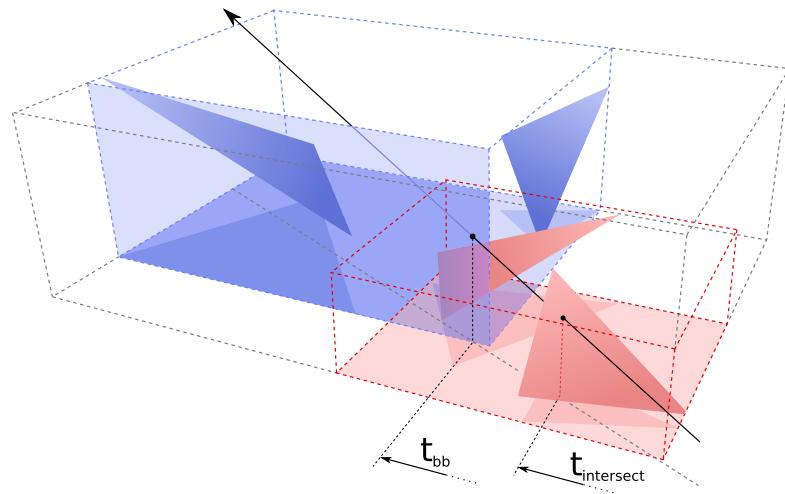


Abbildung 3.11: Da t_{bb} größer ist als $t_{intersect}$ muss der Knoten zu dem die blaue AABB gehört nicht mehr traversiert werden. Der Strahl schneidet zwar die blaue AABB, das relevante Strahlsegment kann aber auf $t_{intersect}$ verkürzt werden.

Quelltext 3.6: Die Bounding Interval Hierarchy Inter Datenstruktur in erweiterter Backus-Naur-Form

```

BIH      = InnerNode | Leaf
InnerNode = Axis float float BIH BIH
Leaf     = { Primitiv }
    
```

3.3 Hybride Raumaufteilung

Um von den Stärken beider Vorgehensweisen zu profitieren wurden in letzter Zeit, unabhängig voneinander, mehrere ähnliche Verfahren entwickelt (unter anderem [Zuniga u. Uhlmann \(2006\)](#), [Havran u. a. \(2006\)](#)) die versuchen die Vorteile von kd-Trees und BV-Hs zu kombinieren. Stellvertretend soll hier die Bounding Intervall Hierarchy(BIH) von [Wächter u. Keller \(2006\)](#) vorgestellt werden.

Anstatt eine komplette Bounding Box in jedem Knoten zu speichern, enthält ein innerer Knoten lediglich zwei, zu einer Hauptachse orthogonalen, Ebenen sowie Referenzen zu zwei Kindknoten.

Wächter schlägt zur Konstruktion eine neue, globale Heuristik vor. Dabei wird die AABB Szene für jeden Rekursionsschritt entlang der längsten Achse am räumlichen Median geteilt. Die Objekte werden dann jeweils danach, welchen der Subvoxel sie am meisten überlappen, in den rechten oder linken Teilbaum eingesortiert. Für Dreiecke wird meistens lediglich geprüft ob der Schwerpunkt vor oder hinter der Trennebene liegt. Die

beiden Ebenen werden anschließend auf die maximale beziehungsweise minimale Ausdehnung der Objekte im linken, beziehungsweise rechten Teilbaum, entlang der aktuellen Achse gesetzt. Der Vorgang wird für die Teilmengen wiederholt bis lediglich ein Objekt in der Liste verbleibt, für welches dann ein Blattknoten angelegt wird. Wichtig ist, dass der Voxel dessen Mittelpunkt für die Bestimmung der Trennebene verwendet wird, nicht an die tatsächliche AABB der enthaltenen Objekte angepasst wird. Quelltext 3.7 veranschaulicht die Vorgehensweise.

Da die Objekte nur in einen der beiden Kindknoten eingesortiert werden, teilt sich die BIH mit der BVH die Vorteile, die durch die einmalige Referenzierung eines Objekts entstehen. Im Gegenzug kann es passieren, dass das Maximum der Objekte die vor der Trennebene liegen, größer ist als das Minimum der Objekte die hinter der Trennebene liegen. Damit teilt die BIH auch den Nachteil der BVH, dass bei Fund eines Schnittpunkts, der Teil des Baums der noch nicht untersucht wurde, zumindest für den Bereich der Überlappung, noch traversiert werden muss. Durch die Quicksort-ähnliche Sortierung bei der Konstruktion wird jedoch sichergestellt, dass die Überlappung so klein wie möglich gehalten wird. Da wie bei der BVH Bereiche hinter dem gefundenen Schnittpunkt von der Suche ausgeschlossen werden können, werden die Fälle in denen beide Kindknoten traversiert müssen, auf ein Minimum reduziert. Ein Knoten der BIH lässt sich fast so kompakt darstellen wie der eines kd-Trees, da im Gegensatz zur BVH nicht ganze AABBs gespeichert werden. Auch die Tests bei der Traversierung fallen ähnlich einfach wie bei den kd-Trees aus, da lediglich entlang einer Achse getestet werden muss. Dies ermöglicht auch die eindeutige Bestimmung der Traversierungsreihenfolge, wie sie bei kd-Trees vorgenommen wird. Quelltext 3.8 zeigt wie einfach die Suche nach einem Schnittpunkt durchzuführen ist.



3.4 Paket- und Frustumtraversierung

Eine weitere Klasse von Verfahren, die sich mit den bisher beschriebenen Datenstrukturen kombinieren lässt, nutzt die räumliche Kohärenz von Strahlen aus. Darunter versteht man praktisch, Strahlen deren Ursprünge nah beieinander liegen und die sich in ähnliche Richtungen ausbreiten. Diese Form der Kohärenz tritt besonders für *Primärstrahlen* auf, lässt sich bei geschickter Anordnung aber auch bei Sekundärstrahlen begrenzt ausnutzen.

Das Verfahren profitiert von Fällen in denen viele Operationen, für alle Strahlen einer Gruppe das gleiche Ergebnis liefern. Genutzt werden hierfür konservative Tests die für ein Stellvertreter der Gruppe beantwortet werden, und durch dessen Eigenschaften die Ergebnisse der Tests auf alle Strahlen der Gruppe verallgemeinert werden können.

Quelltext 3.7: Konstruktion einer BIH mit der von Wächter vorgeschlagenen globalen Heuristik

```

BihNode subdivide(List<Primitive> primitives, AABB aabbnode, int depth) {
    if ( ( primitives.size() == 1 ) || ( depth == 0 ) )
        return new Leaf(primitives[0]); // create leaf

    splitaxis = aabbnode.getLongestAxis();
    splitpos = aabbnode.Minaxis +  $\frac{aabb_{node}.width_{axis}}{2}$ ;
    left = 0;
    right = primitives.size() - 1;
    do { // quicksortlike in-place partitioning
        while ( left < right && primitives[left].centeraxis <= splitpos )
            ++left;
        while ( right > left && primitives[right].centeraxis > splitpos )
            --right;
        if ( left < right )
            primitives.swap(left, right);
    } while ( left < right );

    if ( primitives[right].centeraxis < primitives[left].centeraxis )
        primitives.swap(left, right);
    List<Primitive> leftPrimitives = primitives.range(start, left - 1);
    List<Primitive> rightPrimitives = primitives.range(left, end);

    AABB leftBox = nodeAABB.split(axis, splitPos, left);
    AABB rightBox = nodeAABB.split(axis, splitPos, right);
    if ( left == end + 1 ) // all objects left
        return subdivide( start, end, leftBox, depth );
    else if ( left == start ) // all objects right
        return subdivide( start, end, rightBox, depth );
    else {
        leftMax = nodeAABB.Minaxis;
        rightMin = nodeAABB.Maxaxis;
        foreach ( primitive in leftPrimitives )
            if ( primitive.aabb.Maxaxis > leftMax )
                leftMax = primitive.aabb.Maxaxis;
        foreach ( primitive in rightPrimitives )
            if ( primitive.aabb.Minaxis < rightMin )
                leftMax = primitive.aabb.Minaxis;
        return new Innernode( subdivide(leftPrimitives, leftBox, depth - 1), // left node
                             subdivide(rightPrimitives, rightBox, depth - 1), // right node
                             leftMax, rightMin, axis );
    }
}

```

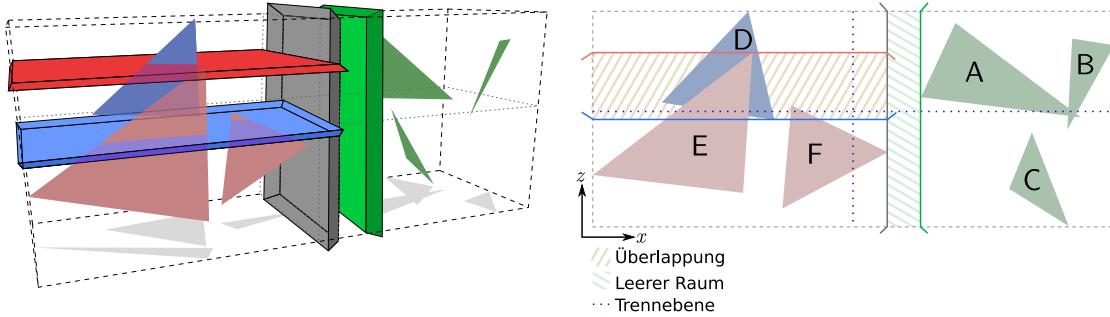


Abbildung 3.12: Beispiel für eine Raumaufteilung mit der BIH. Durch die erste (vertikale) Trennebene werden die Objekte A,B,C von den restlichen getrennt. Dabei entsteht keine Überlappung. In der nächsten Stufe wird das Dreieck D von den beiden anderen Dreiecken E und F in diesem Knoten getrennt. Dabei ergibt sich die eingezeichnete Überlappung, da das Maximum des Knotens der E und F größer ist, als das Minimum des Knotens der D enthält.

Quelltext 3.8: Traversierung der BIH. Der `+ =` Operator weißt nur dann zu wenn der rechte Operand einen kleineres `t` aufweist als der linke.

```

Intersection searchClosestIntersection( BIH node, Ray ray, float t_min, float t_max ) {
    Intersection closest(); // initialisiert mit dem 'leeren' Schnittpunkt
    if ( node.isLeaf() ) {
        foreach ( primitive in node.primitives )
            closest += primitive.getIntersection( r );
        return closest;
    }
    if ( r.direction[node.axis] > 0.0 ) { // aus der Strahlrichtung kann geschlossen werden
        ( near, far ) = ( left, right );
    } else
        ( near, far ) = ( right, left );
    t_near = ( node.plane_near - ray.origin[node.axis] ) / r.direction[node.axis];
    t_far = ( node.plane_far - ray.origin[node.axis] ) / r.direction[node.axis];
    if ( t_min > t_near ) { // naher voxel wird nicht vom Strahl geschnitten
        if ( t_far < t_max ) { // entfernter Voxel wird geschnitten
            closest += searchClosestIntersection(node.child_far, ray, max( t_min, t_far ), t_max);
        } // else: Strahl verlaeuft zwischen Ebenen und schneidet keinen der Kindvoxel
    } else if ( t_far > t_max ) { // ausschliesslich der nahe Voxel muss durchsucht werden
        closest += searchClosestIntersection( node.child_near, ray, t_min, min( t_max, t_near ) );
    } else { // beide Knoten muessen durchsucht werden
        closest += searchClosestIntersection( node.child_near, ray, t_min, min(t_max, t_near ) );
        closest +=
            searchClosestIntersection( node.child_far, ray, max(t_min, t_far), min(closest.t , t_max));
    }
    return closest;
}
    
```

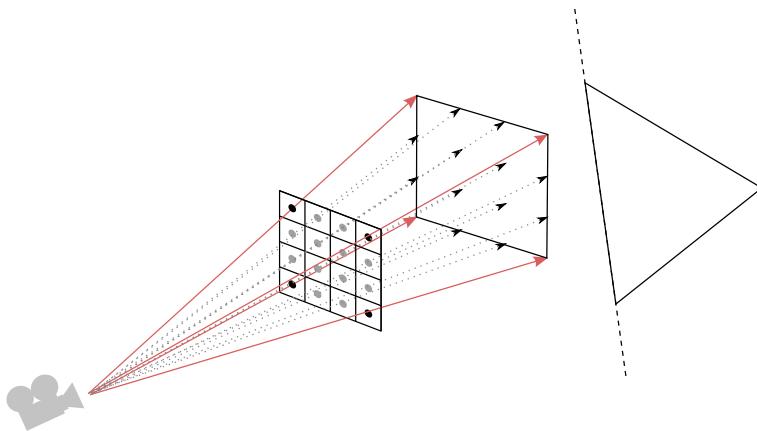


Abbildung 3.13: Die Schnittpunkte der vier Eckstrahlen liegen außerhalb der gleichen Seite des Dreiecks (eine baryzentrische Koordinate muss außerhalb von $[0, 1]$ liegen siehe Anhang 7.1). Durch die Konvexität des Vierecks kann geschlossen werden, dass auch die “inneren” Strahlen keinen gemeinsamen Punkt mit dem Dreieck besitzen.

Dmitriev u.a. (2004) wenden diese Idee das erste Mal für den Schnitttest von Strahlpaketen mit Dreiecken an. Anstatt jeden Strahl einzeln mit dem Dreieck zu testen, werden zunächst die vier “Eckstrahlen” jeweils den folgenden fünf Tests unterzogen:

1. liegt die Ebene des Dreiecks vor Beginn des Strahlsegments ($t_{split} < t_{min}$)
2. liegt die Ebene des Dreiecks nach Ende des Strahlsegments ($t_{split} > t_{max}$)
3. für jede der **drei** baryzentrischen Koordinaten des Schnittpunktes von Strahl und Ebene des Dreiecks wird getestet ob diese negativ ist (zu baryzentrischen Koordinaten siehe 7.1).

Schlägt einer dieser Tests für alle vier “Eckstrahlen” gleichzeitig fehl, verfehlt das gesamte Paket das Dreieck. Durch die Konvexität des Pyramidenstumpfes, der durch die vier Eckstrahlen gebildet wird, kann direkt geschlossen werden, dass keiner der Strahlen in dem Paket das Dreieck schneidet (siehe Abbildung 3.13). Für die Fälle, in denen keiner der Tests für alle vier Strahlen gleichzeitig fehlschlägt, muss der Test jedoch für alle Strahlen des Pakets einzeln durchgeführt werden.

Die Kohärenz lässt sich nicht nur bei den Schnitttests von Strahlen und Dreiecken anwenden, sondern kann bereits bei der Traversierung der Beschleunigungsdatenstruktur ausgenutzt werden. Für kohärente Strahlen werden während der Traversierung ähnliche Teile der Beschleunigungsdatenstruktur besucht. Die Operationen die dabei durchgeführt werden müssen, können dabei für die Strahlen im Paket zusammengefasst und somit effizienter ausgeführt werden.

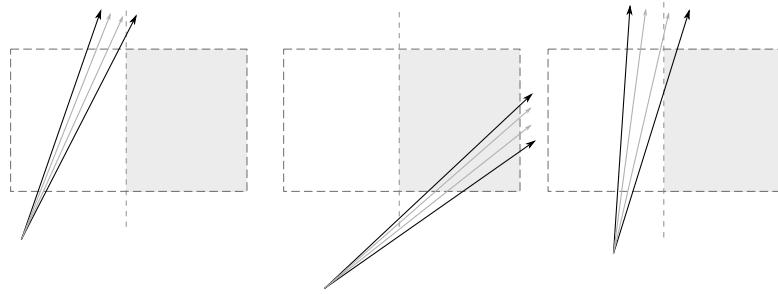


Abbildung 3.14: Verfolgung von Strahlpaketen am Beispiel kd-Tree. links: alle Strahlen schneiden lediglich den linken Voxel, die Traversierung wird lediglich für den linken Teilbaum fortgesetzt. mittig: die Traversierung muss lediglich für den rechten Teilbaum fortgesetzt werden. rechts: sobald auch nur ein Strahl des Pakets einen Voxel schneidet muss dieser Voxel untersucht werden. Deswegen müssen hier beide Teilbäume traversiert werden.

Die Traversierungsroutine der Raumaufteilungsverfahren muss allgemein wie folgt angepasst werden um die Verarbeitung von Strahlpaketen zu ermöglichen. Bei den beschriebenen Verfahren wurde bisher für einen einzelnen Strahl festgestellt welche Voxel von diesem Strahl geschnitten werden. Für Pakete von Strahlen können für die im Paket enthaltenen Strahlen unterschiedliche Voxel geschnitten werden (siehe Abbildung 3.14). Die Untersuchung muss deshalb für **jeden** der Voxel die von *irgendeinem* Strahl des Pakets geschnitten werden, für das gesamte Paket fortgesetzt werden.

Dadurch, dass Knoten auch besucht werden, wenn sie lediglich von wenigen Strahlen des Pakets geschnitten werden, entsteht ein entsprechender Overhead. Dieser amortisiert sich allerdings für kohärente Pakete mit dem Leistungsgewinn, der durch die gemeinsame Traversierung entsteht. Der Leistungsgewinn wird erzielt indem nicht für jeden Strahl einzeln ermittelt wird welcher Voxel geschnitten wird. Vielmehr wird wieder über einen konvexen Stellvertreter für das gesamte Paket ermittelt welche Voxel potentiell geschnitten werden. So lassen sich schnell ganze Teilbäume verwerfen. Dieser Ansatz ist für jedes Raumauflösungsverfahren einsetzbar. Die effiziente Implementierung unterscheidet sich aber für die verschiedenen Verfahren stark und wird deshalb für die in Kapitel 3 vorgestellten Strukturen nachfolgend kurz dargestellt.

3.4.1 Paketraversierung für reguläre Gitter

Wald u. a. (2006) haben, bereits mit dem Anspruch animierte Szenen darzustellen, einen effizienten Algorithmus zu Verfolgung kohärenter Strahlpakete durch gleichmäßige Gitter präsentiert. Um der in Abbildung 3.15 dargestellten Problematik aus dem Weg zu gehen,

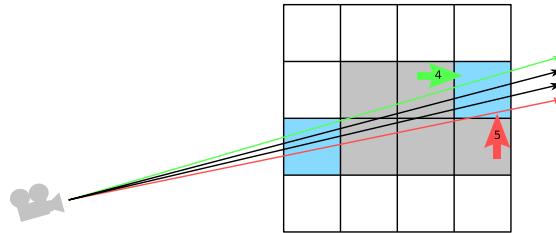


Abbildung 3.15: Probleme bei Verfolgung von Strahlpaketen in regulärem Gitter (2D). Durch die gleichmäßig „feine“ Aufteilung divergieren die Strahlen schneller als bei hierarchischen Strukturen. Das Strahlpaket könnte zwar aufgeteilt werden, jedoch werden unter Umständen wieder die gleichen Zellen besucht. Man könnte die Pakete dann natürlich wieder zusammenfügen. Zusätzlich kann die Zelle jedoch nach unterschiedlich vielen Schritten von den verschiedenen Paketen besucht werden, was dieses Vorgehen insgesamt unattraktiv macht.

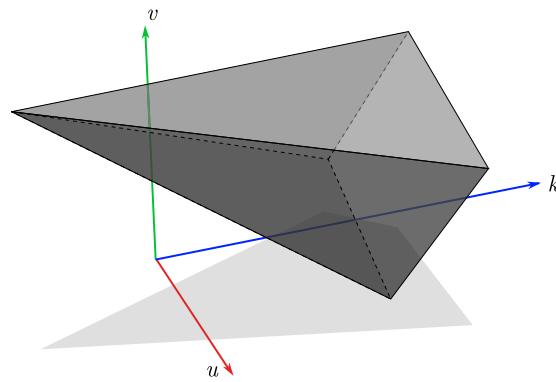


Abbildung 3.16: Für die scheibenbasierte Gittertraversierung wird eine Hauptachse k bestimmt entlang derer sich alle Strahlen eines Paketes in gleicher Richtung ausbreiten (positiver bzw. negativer).

wird, anstatt das in 3.1.1 vorgestellte Verfahren zu erweitern wird, ein komplett neuer Ansatz gewählt.

Zunächst muss für das Paket eine der Achsen des Weltkoordinatensystems als *Hauptachse* k gewählt werden. Entlang dieser müssen sich von den Strahlen im Paket entweder alle in positiver oder alle in negativer Richtung ausbreiten. Dies stellt keine Einschränkung für *kohärente* Pakete dar, da für den Fall das dies nicht gegeben wäre, zwischen zwei Strahlen ein Winkel größer als 180° aufgespannt würde - was nicht mehr dem Kohärenzkriterium entspräche. Die beiden verbleibenden Achsen werden mit u und v bezeichnet.

Wald u. a. (2006) schlagen nun vor das Gitter in Scheiben zu analysieren. Eine Scheibe des Gitters bilden dabei alle Gittervoxel für die der Gitterindex entlang k den gleichen Wert hat. Das Verfahren beginnt mit der ersten Scheibe, die von dem Strahlpaket geschnitten wird. Für die Scheibe wird ermittelt welche der enthaltenen Voxel von mindestens einem Strahl geschnitten werden. Die Objekte dieser Voxel werden dann mit

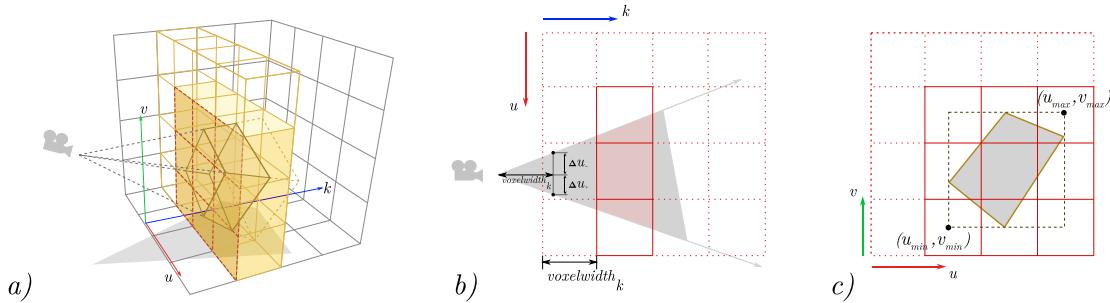


Abbildung 3.17: a) Für die Traversierung sind lediglich die Voxel einer Scheibe interessant, die von dem Pyramidenstumpf geschnitten werden b) Durch die Seiten des Pyramidenstumpfs lässt sich bestimmen um wieviel sich der Ausschnitt einer Scheibe der die relevanten Voxel enthält pro Schritt in Richtung k ausdehnt. c) Ausschnitt der aktuellen Scheibe in der Frontalansicht

jedem Strahl auf Schnitt getestet. Danach wird mit der nächsten Scheibe in Richtung k ebenso fortgefahrene. Der Vorteil im Vergleich zur der Vorgehensweise nach Abschnitt 3.1.1 ist, dass das Strahlpaket nun immer als Ganzes behandelt werden kann. Andererseits führt es aber auch dazu, dass Strahlen mit Objekten geschnitten werden, deren Voxel sie nicht schneiden. Die zusätzlichen Kosten amortisieren sich jedoch für kohärente Pakete über das gesamte Verfahren. Mit dieser Grundlage kann man nun dazu übergehen, die Überlappung der Strahlen und einer Gitterscheibe nicht mehr per Strahl vorzunehmen. Anstatt dessen wird die konvexe Hülle des Strahlpakets in Form eines Pyramidenstumpfs ermittelt. Im einfachsten Fall - für Primärstrahlen eines Rechtecks im Bildbereich - sind dies einfach die vier Eck-Strahlen eines Bündels. Die Überlappung des Pyramidenstumpfs wird vereinfacht als achsenparalleles Rechteck dargestellt (siehe Abbildung 3.17 c)). Die aktuelle Ausbreitung dieses Rechtecks pro Scheibe, kann iterativ durch die Addition der zuvor ermittelten Steigungen ($\Delta u_+, \Delta u_-, \Delta v_+, \Delta v_-$) errechnet werden (siehe Abbildung 3.17 b) $\Delta u_+, \Delta u_-$). Die Umwandlung der so errechneten Werte in Gitterindizes erhält man durch Subtraktion der Werte vom Gitterursprung und anschließende Division durch die Breite eines Voxels.

Wie bereits in der Einführung zu disjunkten Raumaufteilungsverfahren erwähnt kann es bei regulären Gittern besonders für große Objekte dazu kommen, dass diese in mehreren Voxeln referenziert werden. Da durch die Regelmäßigkeit die Voxelgrenzen nicht an die Objekte angepasst werden, trifft dies auch für kleinere Objekte häufig zu und führt dazu, dass die meisten Objekte mehrfach referenziert werden. Durch die Traversierung von Paketen ergeben sich zusätzliche redundante Strahl-Objekt-Schnitte, da alle Strahlen immer gegen alle Objekte aller vom Paket geschnittenen Voxel einer Scheibe getestet werden. Deswegen profitiert die Paketraversierung umso mehr von dem Ein-

satz von *Mailboxing*. Der Overhead, der dabei pro Strahl entsteht, ist jedoch wesentlich geringer als bei der Traversierung einzelner Strahlen, da das *Mailboxing* auf das ganze Paket angewendet werden kann. Konkret heißt das, dass die Objekte beim Schnittest mit einer *Paket-ID* anstatt einer Strahl-ID gekennzeichnet werden. Trifft die Traversierung auf ein Objekt das bereits mit dem aktuellen Paket markiert wurde, kann der Test für alle Strahlen des Pakets übersprungen werden. ([Wald u. a., 2006](#))

3.4.2 kd-Trees

Der Traversierungsalgorithmus für kd-Trees kann recht intuitiv für die Verarbeitung von Strahlpaketen angepasst werden. Die Entscheidung, ob an einem inneren Knoten mit dem linken, dem rechten oder beiden Teilbäumen fortgefahrene wird, hängt nun lediglich davon ab, ob es *irgendeinen* Strahl im Paket gibt, der den entsprechenden Subvoxel schneidet. An den Blattknoten werden alle Strahlen des Pakets mit den enthaltenen Objekten auf Schnitt getestet. Einer der Gründe warum die kd-Trees die anderen Verfahren in vielen Fällen übertreffen ist die Möglichkeit die Objekte, die dem Strahlursprung näher gelegen, sind vor Objekten zu testen, die vom Ursprung weiter entfernt liegen. Deswegen kann die Suche entsprechend früher terminiert werden als bei anderen Verfahren. Hierfür ist aber die Befolgung der richtigen Traversierungsreihenfolge notwendig. [Wald \(2004\)](#) und [Benthin \(2006\)](#) zeigen, dass eine eindeutige Bestimmung in welcher Reihenfolge die Kinder eines Knotens traversiert werden müssen für zwei allgemeine Fälle für Strahlpakete möglich ist. Abbildung 3.18 veranschaulicht, dass für den ersten Fall alle Strahlen den gleichen Ursprung besitzen müssen. Für den zweiten Fall müssen die Komponenten der Richtungsvektoren der verschiedenen Strahlen das gleiche Vorzeichen besitzen. Die eindeutige Reihenfolge bedeutet *nicht*, dass jeder der Subvoxel von jedem tatsächlich Strahl getroffen wird. Es bedeutet lediglich, dass die Anwendung der frühen Terminierung in dieser Reihenfolge kein falsches Ergebnis liefert. Würde die Reihenfolge für die in Abbildung 3.18 schwarz eingezeichneten Strahlen auf die roten Strahlen angewandt und im als *nah* markierten Knoten ein Schnittpunkt gefunden, würde bei der frühen Terminierung die Suche nach dem Besuch des ersten Kindknotens abgebrochen. Ein Schnittpunkt der in dem, für den roten Strahl fälschlicherweise als *fern* markierten Knoten liegen würde, könnte nicht mehr gefunden werden.

[Wald \(2004\)](#) wählt als Kriterium die Gleichheit der Vorzeichen der Komponenten der Richtungsvektoren. Pakete die dieses Kriterium nicht erfüllen müssen gesondert behandelt werden. Am einfachsten lässt sich dies durch einzelne Verfolgung der Strahlen des Pakets umsetzen. Für kohärente Pakete tritt dieser Fall jedoch sehr selten ein. Durch die

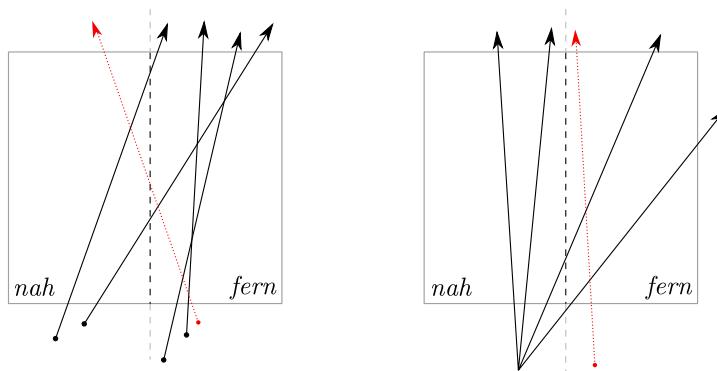


Abbildung 3.18: Bestimmung der Traversierungsreihenfolge. Der rote Fall bricht das jeweilige Kriterium, um zu zeigen, dass dies die Eindeutigkeit zerstört. links: Die Ausbreitungsrichtung ist für alle Strahlen entlang der Trennachse gleich. rechts: Der gleiche Ursprung ermöglicht die eindeutige Bestimmung der Reihenfolge.

Sicherheit, dass die Komponenten unterschiedlicher Richtungsvektoren das selbe Vorzeichen haben, lässt sich ein effizienterer Algorithmus zur Bestimmung der Reihenfolge finden als für den Fall, dass lediglich der Ursprung der Strahlen übereinstimmt. Da dieser Teil des Programms weitaus häufiger ausgeführt wird als eine anfängliche Überprüfung der Vorzeichen, ist die Vorzeichengleichheit als Kriterium dem des gleichen Ursprungs vorzuziehen.

3.4.3 Bounding Volume Hierarchies

Wald u. a. (2007) haben unter anderem ein effizientes Vorgehen zur Traversierung von BVHs entwickelt. Dabei werden verschiedene konservative Tests angewendet, um einfache Fälle während der Traversierung besonders schnell entscheiden zu können.

Der erste Test profitiert davon, dass ein Teilbaum durchsucht werden muss sobald einer der Strahlen im Paket die entsprechende AABB schneidet. Während der Traversierung wird für jeden Knoten verfolgt, welches der erste Strahl im Paket ist, der die AABB des Knotens schneidet. Dieser Strahl wird dann auch als erstes mit den Kindknoten auf Schnitt getestet. So kann in vielen Fällen schnell entschieden werden, dass ein Kindknoten besucht werden muss. Trifft dieser Strahl nicht, werden nicht die restlichen Strahlen getestet sondern, zunächst der folgende Test ausgeführt.

Der zweite, so genannte “early miss”-Test, prüft anhand von Intervallarithmetik, ob das Paket die AABB des Teilbaums überlappen kann. Ist dies ausgeschlossen, kann der Teilbaum ohne weitere Tests verworfen werden.

Hat keiner der beiden Tests ein positives Ergebnis geliefert, werden die restlichen Strah-

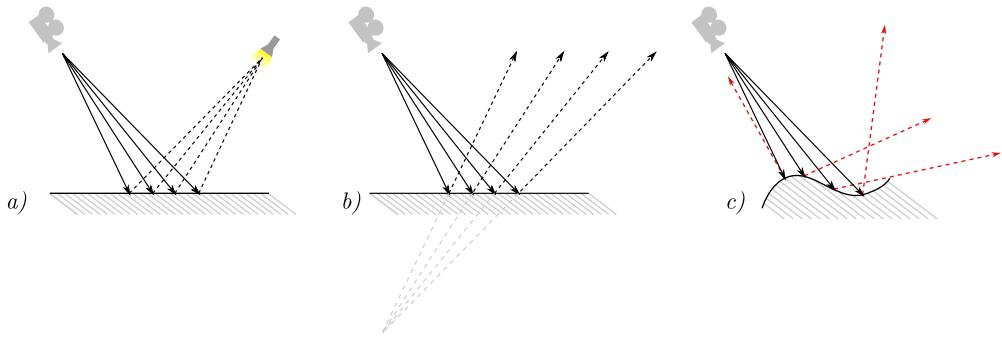


Abbildung 3.19: a) Erzeugung eines kohärenten Schattenstrahlpaketes durch Verlagerung des Ursprungs zur Position der Lichtquelle. b) Bei Reflexion an einer ebenen Fläche bleibt die Kohärenz auch für die reflektierten Strahlen erhalten. c) Durch Reflexion an einer gekrümmten Fläche wird die Kohärenz der reflektierten Strahlen zerstört.

len im Paket auf Schnitt mit der AABB des Teilbaums getestet bis entweder ein Strahl gefunden wird der sie schneidet oder für alle Strahlen festgestellt wird, dass sie die AABB verfehlten.

3.4.4 Sekundärstrahlen

Für die Schattenstrahlen der Schnittpunkte eines Strahlpaketes lässt sich einfach ein kohärentes Strahlpaket finden indem die Schattenstrahlen umgedreht werden. Für den gemeinsamen Ursprung der Strahlen wird die Position der Lichtquelle gewählt. Die Richtungen der Schattenstrahlen zeigen auf die gefundenen Schnittpunkte. Die Kohärenz bleibt auch erhalten, wenn die Primärstrahlen nicht das selbe Objekt treffen, kann aber zerstört werden, wenn die Objekte weit voneinander entfernt liegen. Für reflektierte Strahlen ist die Situation komplizierter. Kohärente Pakete von reflektierten Strahlen lassen sich effizient nur für solche Pakete finden, deren Strahlen an der gleichen Ebene reflektiert werden. Abbildung 3.19 zeigt die entsprechenden Fälle. ([Dmitriev u. a., 2004](#))

4 Animation

Kamerafahrten durch statische Modelle sind per Raytracing schon seit ein paar Jahren, auch auf Arbeitsplatzrechnern, möglich. Für animierte, beziehungsweise allgemein dynamische Szenen, müssen die verwendeten Algorithmen jedoch unter einem weiteren Aspekt betrachtet werden. Da sich die Geometrie hier von Frame zu Frame verändert, kann die für die vorherige Ausprägung der Geometrie erstellte Beschleunigungsdatenstruktur nicht mehr verwendet werden. Veränderte Positionen oder neue Objekte machen die Aussagen, die eine *veraltete* Beschleunigungsstruktur trifft, ungültig.

4.1 Klassifizierung von Problemen dynamischer Szene

Für die Darstellung dynamischer Szenen müssen die bisher beschriebenen Datenstrukturen neu bewertet werden. [Lext u. a. \(2000\)](#) haben verschiedene Problemklassen für das Raytracing definiert:

1. Hierarchische Animation
2. Unstrukturierte Bewegung von Objekten
3. “Teapot-in-Stadium”-Problem
4. Niedrige Kohärenz zwischen zwei Bildern
5. Überlappende Bounding Volumen
6. Variierende Objektverteilung

Für dynamische Szenen besonders interessant sind die Punkte 1, 2 und 4. Die anderen Punkte treten auch für statische Szenen auf, durch die lediglich die Kamera bewegt wird. Deswegen werden nun diese drei Punkte näher erläutert.

Hierarchische Animation Bei der Animation, beziehungsweise Simulation von komplexen Bewegungen werden einzelne Teile hierarchischer Objekte häufig in einem lokalen Koordinatensystem modelliert. Anschließend wird dieses relativ zu dem Vorgängerknoten in einem Szenengraphen positioniert. Die einzelnen Teile ändern ihre

Geometrie häufig über die gesamte Dauer der Animation nicht. Diese Art der Kohärenz kann ausgenutzt werden, um dynamische Szenen effizienter darzustellen. Allerdings werden Systeme, die hauptsächliche auf dieser Annahme aufbauen in Szenen mit einem kleinen Anteil hierarchischer Animation dementsprechend schwache Leistung zeigen

Unstrukturierte Bewegung Bei der unstrukturierten Bewegung gibt es keinerlei Zusammenhang zwischen den Bewegungen einzelner Objekte. Da keine Kohärenzen zwischen den Objekten ausgenutzt werden können stellt dies ein komplexes Problem für jeden Raytracer dar.

Niedrige Kohärenz zwischen zwei Frames Ähneln sich zwei aufeinanderfolgende Bilder, können unter Umständen Informationen aus dem letzten Bild verwendet werden, um das aktuelle schneller zu berechnen. Dies gilt sowohl für Algorithmen, die dies im Bildbereich ausnutzen als auch für die Position und Orientierung der 3D-Objekte im Raum. Das Fehlen einer solchen Kohärenz kann entsprechende Verfahren verlangsamen.

4.2 Konsequenzen für die Datenstrukturen

Der einfachste Ansatz Veränderungen der Szenengeometrie zu berücksichtigen ist der komplette Neuaufbau der Beschleunigungsdatenstruktur für jedes Frame. Die Konstruktionszeiten variieren sehr stark. Sie können je nach Datenstruktur, Heuristik und Anzahl der Objekte von ein paar Millisekunden bis zu einigen Minuten dauern. Natürlich ist es wünschenswert, Strukturen die für das letzte Frame erstellt wurden soweit wie möglich weiterzuverwenden. Dies ist jedoch nicht für alle der beschriebenen Datenstrukturen uneingeschränkt möglich.

4.2.1 Trennung von statischen und dynamischen Inhalten

Für Anwendungen, bei denen große Teile der Geometrie statisch sind, kann es sich auszahlen dynamische und statische Inhalte getrennt zu verarbeiten. Unter Umständen kann es sich sogar lohnen weiter zwischen hierarchischer und unstrukturierter Bewegung zu unterscheiden. Dabei stellt sich die Frage nach welchen Kriterien diese Unterscheidung vorgenommen wird. Wald (2004) überlässt diese Entscheidung der Anwendung, in der dynamischen Inhalte gesondert spezifiziert werden müssen. Ein ähnliches Vorgehen wird auch für Verfahren, welche die Rendering-Pipeline der Grafikkarte verwenden genutzt. So

genannte *Vertex Buffer Objects* werden verwendet, um Geometriedaten direkt in den Grafikkartenspeicher zu schreiben. Dies kann genutzt werden, um die Darstellung von Inhalten die sich gar nicht oder nur sehr langsam ändern effizienter zu gestalten [NVIDIA \(2003\)](#).

In [Günther u. a. \(2006\)](#) wird die Möglichkeit aufgezeigt, durch Analyse von animierten Modellen in verschiedenen *Posen*, Inhalte, die ausschließlich affiner Transformation unterzogen werden, automatisch zu identifizieren.

Für große Teile einer Szene, die lediglich als ganzes, affin transformiert werden, lohnt es sich die Beschleunigungsdatenstruktur getrennt aufzubauen. Pro Bild muss dann lediglich eine Datenstruktur aufgebaut werden, in der die Teilmodelle als ganze Objekte behandelt werden. Anstatt jedes Element der Teilmodelle in das Weltkoordinatensystem zu transformieren, wird zur Laufzeit lediglich der Strahl in das lokale Koordinatensystem des entsprechenden Modellteils transformiert. So kann ein großer Teil der Konstruktionszeit gespart werden.

Der Ansatz den Strahl in das Koordinatensystem von Modellteilen zu transformieren hat den positiven Nebeneffekt, dass sich die Teilgeometrien leicht *instanziieren* lassen. Das heißt, die für ein Teilmodell aufgebaute Beschleunigungshierarchie kann an mehreren Knoten der “*Meta*”-Struktur in Kombination mit weiteren Transformationen referenziert werden. Dies ist besonders effizient für Modelle in denen bestimmte Teilmodelle sehr häufig verwendet werden, da die Beschleunigungshierarchie für diese Teile nur einmal konstruiert und gespeichert werden muss. Wird der Strahl durch die Kaskadierung von Modellen jedoch häufig transformiert kann es durch die eingeschränkte Genauigkeit der Gleitkommaarithmetik zu numerischen bedingten Fehlern kommen.

4.2.2 Aktualisierung von Beschleunigungsstrukturen

Dennoch gibt es Fälle in denen keine oder zu wenig Informationen über die Dynamik der dargestellten Objekte zur Verfügung steht. Im Gegensatz zu den kd-Trees kann eine BVH aktualisiert werden ohne dass die Topologie verändert werden muss. Da die Topologie der BVH keine Aussage über die räumliche Verteilung der Objekte trifft, müssen nach einer Bewegung der Szenenobjekte lediglich die in den Knoten gespeicherten AABBs den neuen Ausdehnungen angepasst werden. In den meisten Fällen führt dies mittelfristig jedoch zu einer Verschlechterung der Qualität der BVH. Die Strahlanfragen werden immer langsamer, je weiter die Animation fortschreitet. Entfernen sich Objekte, die einen gemeinsamen Vorgängerknoten besitzen, voneinander, vergrößern sie dessen AABB-Volumen und erhöhen somit die Anzahl der Strahlen für die dieser Knoten durchsucht werden muss (siehe

Abbildung 4.1). Wie stark die Qualität der BVH nachlässt kann von vielen Faktoren abhängen (Ausprägung der Geometrie zu Konstruktionszeit, Heuristik, Ausprägung der Dynamik). Um eine für die *Deformierung* geeignete Anfangspartitionierung zu finden, beschreiben Wald u. a. (2007) die Analyse der Animation in verschiedenen Posen (zum Beispiel für Menschmodell). Dadurch lässt sich feststellen welche Objekte für die Dauer der Animation eine räumliche Nähe beibehalten. Dieses Vorgehen erfordert aber auch wieder entsprechendes Vorwissen über die Animation.

Völlig ohne Vorwissen über die Struktur und Bewegung der zugrundeliegenden Geometrie kommt das Verfahren von Lauterbach u. a. (2006) aus. Anhand einer Heuristik wird nach jeder Aktualisierung die Verschlechterung der Qualität der BVH gemessen. Überschreitet die Verschlechterung einen festgelegten Schwellwert wird die BVH neu aufgebaut. Die Heuristik misst die Größenänderung der Oberfläche der AABB eines Knotens im Vergleich zur Größenänderung der Oberflächen der AABBs der Kindknoten. So kann festgestellt wann sich die Oberfläche eines Knotens lediglich aufgrund der Verteilung der Objekte ändert. Fälle wie in Abbildung 4.1 können so frühzeitig festgestellt werden. Die BVH muss dann für die veränderte Objektverteilung neu konstruiert werden. Für Animationen in denen nur wenig Bewegung stattfindet wird unter Umständen kein Neuaufbau für die komplette Dauer nötig. Das Verfahren wird leider nur für Szenen beschrieben, bei denen für die Dauer der Animation keine Objekte hinzugefügt oder entfernt werden. Eine entsprechende Anpassung kann jedoch nicht teurer sein als ein Neuaufbau der BVH und sollte dementsprechend einfach durchzuführen sein.

Die Aussagen und Verfahren, die hier über die Aktualisierung von BVHs vorgestellt wurden, sind auf alle anderen Verfahren, die auch die Objektliste partitionieren, wie zum Beispiel die BIH, direkt übertragbar.

4.2.3 Schnelle Konstruktion

Trotz ausgereifter Verfahren zur Aktualisierung von BVHs stellt sich die Frage wie sich *gute* Beschleunigungsdatenstrukturen möglichst effizient konstruieren lassen. Ein Ansatz die Konstruktionszeit für hierarchische Datenstrukturen zu verringern ist es, die Datenstruktur vor der ersten Strahlanfrage gar nicht oder nur teilweise aufzubauen. Bei diesem *lazy building* soll vermieden werden, dass Teile der Datenstruktur aufgebaut werden, die für das aktuelle Bild von keinem Strahl durchsucht werden müssen. Dies bietet sich besonders an, wenn zusätzlich die von der Anwendung bereitgestellten Daten vor der Konstruktion weiteren Vorverarbeitungsschritten unterzogen werden müssen (zum Beispiel die Tesselierung von Flächen Djeu u. a. (2007)). Die inneren Knoten der Daten-

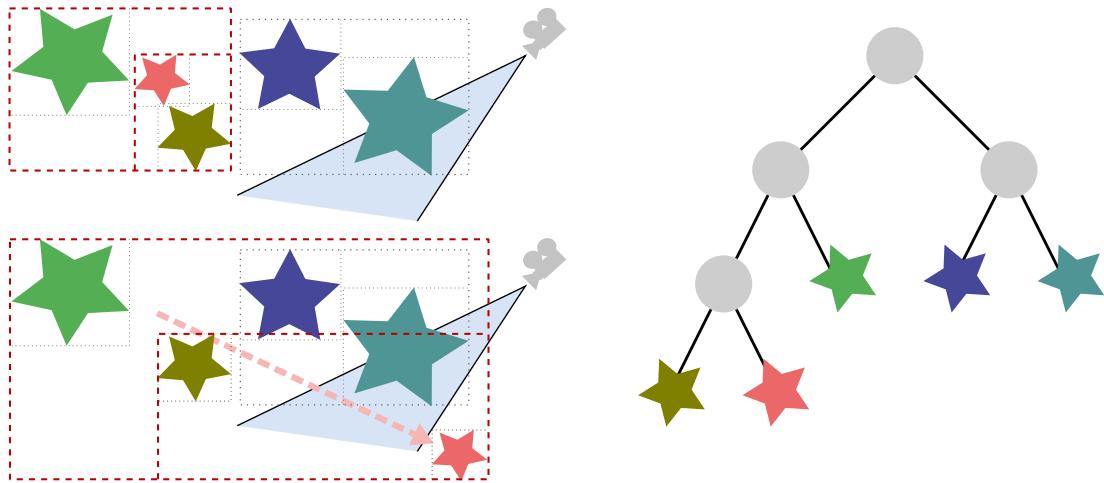


Abbildung 4.1: Qualitätsverminderung einer BVH. Bei Bewegung von Objekten kann die Topologie der BVH beibehalten werden, lediglich die BBs müssen angepasst werden. Doch durch ungünstige Bewegung, wie hier des roten Objekts, müssen für Strahlen des gezeigten Frustums alle inneren Knoten der BVH getestet werden. Ein Baum bei dem das rote Objekt im rechten Teilbaum der Wurzel eingesortiert würde, wäre effizienter.

struktur müssen dafür um ein Datenfeld erweitert werden, welches darüber Auskunft gibt, ob die Teilbäume dieses Knotens für eine Strahlanfrage noch konstruiert werden müssen. Bei der Traversierung eines so markierten Knotens muss dann die Konstruktion der Teilbäume nachgeholt werden. Das verschiebt natürlich lediglich einen Teil des Konstruktionsaufwands in die Traversierungsphase. Teile der Hierarchie welche Objekte enthalten, die von keinem Strahl getroffen werden, müssen aber überhaupt nicht konstruiert werden. Abbildung 4.2 verdeutlicht wie dies für eine einfache Beispieldaten aussehen könnte. Für Verfahren, die ebenfalls Effekte globaler Beleuchtung darstellen, verringert sich der effektive Gewinn dieser Vorgehensweise unter Umständen jedoch drastisch. Da hier viele Teile der Szene untersucht werden müssen, obwohl sie von der Kameraposition direkt sichtbar sind, kann es passieren, dass doch nahezu der gesamte Baum der Beschleunigungsdatenstruktur aufgebaut werden muss.

Die Laufzeit des Konstruktionsalgorithmus hängt hauptsächlich von der Komplexität der Heuristik ab, die für die Partitionierung des Raums beziehungsweise der Objektmenge gewählt wird. Bei der Wahl der Heuristik geht es deshalb hauptsächlich darum, die höheren Kosten einer aufwändigen Heuristik in der Konstruktionsphase gegen die Beschleunigung der Strahlanfragen in der Traversierungsphase abzuwegen.

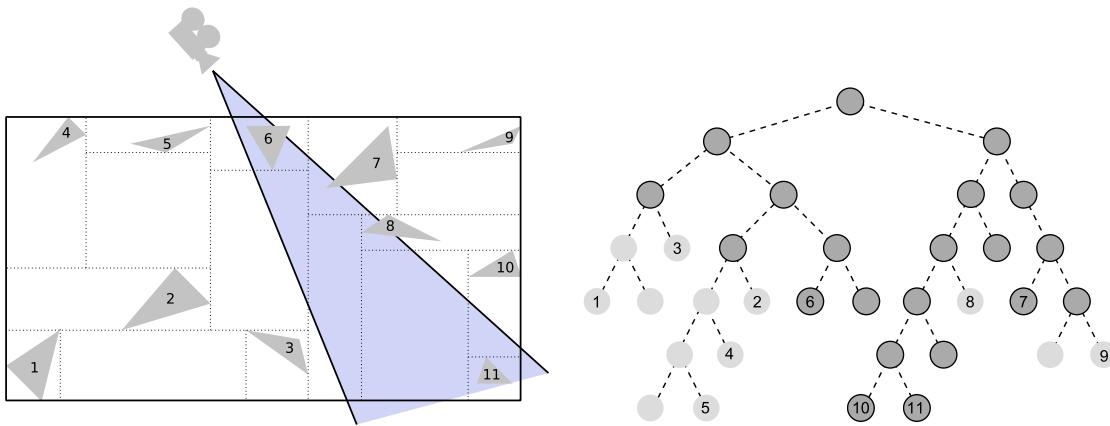


Abbildung 4.2: Binäre Raumaufteilung für eine sehr einfache Szene. Die hellen Knoten im Baum verursachen beim lazy build keine Konstruktionskosten.

Quelltext 4.1: Ereignisse bestehen aus einer Position t und einer Kennzeichnung ob bei dieser Position ein Objekt beginnt oder endet. Für plane Objekte kann noch der Spezialfall eintreten, dass das Objekt komplett in eine Schnittebene an dieser Stelle fallen würde.

```
Event      = float EventType
EventType = END | INPLANE | START
```

SAH in $O(N \log^2 N)$

Die *Surface Area Heuristic* nimmt momentan die beste Raumaufteilung vor. Der im letzten Kapitel beschriebene Ansatz zur Verwendung der SAH (3.4) besitzt eine quadratische Komplexität in Abhängigkeit von den verwendeten Szenenobjekten und ist damit, außer für sehr kleine Szenen, nicht praktikabel. Unter anderem in Pharr u. Humphreys (2004) wird vorgestellt wie man, ohne Beschränkung der Allgemeinheit, das Problem auch in $O(N \log^2 N)$ lösen kann. Die Ursache für die quadratische Komplexität bei dem naiven Ansatz ist, dass die Anzahl der Objekte die sich auf der linken und rechten Seite einer potentiellen Schnittebene befinden (N_{left}, N_{right}) jedesmal durch Iteration über alle Objekte des Knotens ermittelt werden muss - und das für jeden Kandidaten der Trennebene in jedem inneren Knoten.

Um N_{left}, N_{right} entlang einer Achse k effizienter zu bestimmen, lässt sich die Tatsache nutzen, dass die Orte an denen sich diese Werte ändern mit den Positionen der potentiellen Schnittebenen zusammenfallen (nämlich die Minima/Maxima der AABBs der Objekte auf diese Achse). Die Stelle an der die AABB eines Objekts auf eine Achse beginnt oder endet wird auch "Ereignis" genannt.

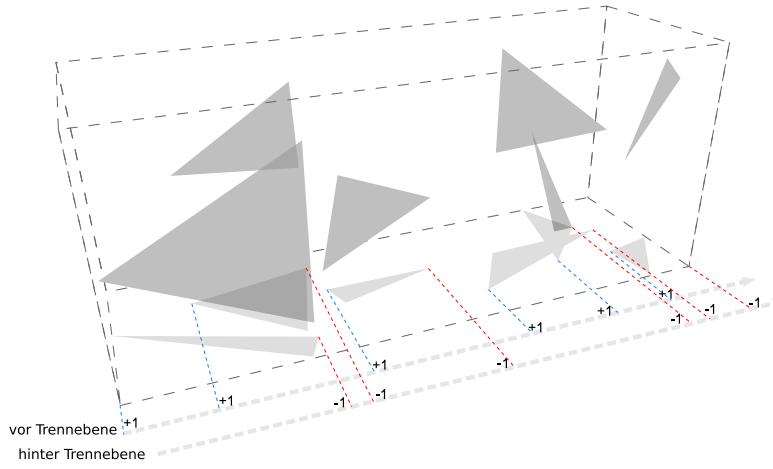


Abbildung 4.3: Durch zählen der bisher aufgetretenen Ereignisse lassen sich während der Iteration über die Liste die Objekte vor und nach dem aktuellen Ereignis bestimmen. Die Anzahl der Objekte vor der Ebene wird mit Null initialisiert und bei jedem “Anfangs”-Ereignis(blau) inkrementiert. Die Anzahl der Objekte hinter der Ebene wird mit der Anzahl der Objekte in diesem Voxel initialisiert und bei jedem “Ende”-Ereignis(rot) dekrementiert.

Erstellt man eine sortierte Liste aller Ereignisse entlang einer Achse, lassen sich N_{left} , N_{right} während einer Iteration über diese Liste durch einfaches “Mitzählen” bestimmen. Bei der Konstruktion der Ereignisliste wird zwischen Ereignissen bei denen eine AABB beginnt und denen, an denen sie endet unterschieden. Für plane Objekte muss zusätzlich der Fall berücksichtigt werden, dass ein Objekt komplett *in* der Trennebene liegen kann. Abbildung 4.3 veranschaulicht diesen Vorgang. Um die Liste der Ereignisse sortieren zu können, muss auf den Ereignissen eine Ordnung definiert sein. Diese wird wie folgt definiert: $(e_1 < e_2) = true$ wenn $e_1.t < e_2.t \vee ((e_1.t = e_2.t) \wedge (e_1.type < e_2.type))$, wobei t die Position des Ereignisses auf der aktuellen Achse darstellt. Für EventType gilt wiederum: *END* < *INPLANE* < *START*.

Für die Evaluierung der Kandidaten werden alle Ereignisse, die an einer Stelle auftreten, “aufgesammelt”. Durch die Ereignisse, die vor einer Stelle aufgetreten sind, kann auf die Anzahl der Objekte im linken, beziehungsweise rechten Subvoxel geschlossen werden. Plane Objekte, die parallel zur Trennebene liegen, benötigen eine gesonderte Behandlung. Liegt ein solches Objekt an der Stelle der Trennebene, und ist einer der beiden Subvoxel leer, ergibt sich ein besserer Baum, wenn diese Objekte in den nicht-leeren Teilbaum sortiert werden. Der komplette Algorithmus wird noch einmal in Quelltext 4.2 in Pseudocode wiedergegeben.

acceleration
 KdTreeSahNlog2N
 construct

Quelltext 4.2: Algorithmus zur Bestimmung der besten Trennebene in $O(N \log^2 N)$

```

1  (float, Axis) searchBestSplitPos(aabb voxel, List<Primitive> primitives) {
2      axisbest = none
3      candidatebest = 0
4      costbest =  $\infty$ 
5      foreach ( axis in {x,y,z} ) {
6          List<Events> events;
7          foreach (primitive in primitives)
8              if ( primitive.aabb.min[axis] == aabb.max[axis] )
9                  events.add( Event(primitive.aabb.min[axis], INPLANE) );
10             else {
11                 events.add( Event(max(primitive.aabb.min[axis], voxel.min[axis]), START) )
12                 events.add( Event(min(primitive.aabb.max[axis], voxel.max[axis]), END) )
13             }
14             events.sort()
15             Nleft = 0, Nright = event.length, NinPlane = 0
16             event = events.first
17             while( event in events ) {
18                 candidatecurrent = event.t
19                 pstart = pend = pinPlane = 0
20                 while( (event in events) && (event.t == candidatecurrent)
21                     && (event.t == END) ) {
22                     event = event.next
23                     ++pend
24                 }
25                 while( (event in events) && (event.t == candidatecurrent)
26                     && (event.t == INPLANE) ) {
27                     event = event.next
28                     ++pinPlane
29                 }
30                 while( (event in events) && (event.t == candidatecurrent)
31                     && (event.t == START) ) {
32                     event = event.next
33                     ++pstart
34                 }
35                 Nright -= (pend + pinPlane)
36                 NinPlane = pinPlane
37                 costcurrent = SAH(v, candidatecurrent, axis, Nleft, Nright, NinPlane);
38                 if ( costcurrent < costbest ) {
39                     costbest = costcurrent
40                     candidatebest = candidatecurrent
41                     axisbest = axis
42                 }
43                 Nleft += (pstart + pinPlane)
44                 NinPlane = 0
45             }
46         }
47     return (candidatebest, axisbest);
48 }
```

SAH in $O(N \log N)$

Eine weitere Reduzierung der algorithmischen Komplexität der Konstruktion anhand der SAH gelang [Wald u. Havran \(2006\)](#). Ihr Verfahren baut auf dem zuvor beschriebenen Prinzip auf. Sie vermeiden jedoch die erneute Sortierung der Ereignisliste. Anstatt dessen wird zu Beginn der Konstruktionsphase einmalig eine Liste aufgebaut, welche die Ereignisse entlang aller Achsen enthält. Jedes Ereignis muss nun zusätzlich um die Information erweitert werden auf welcher Achse es aufgetreten ist (siehe Abbildung 4.4). Außerdem wird für jedes Ereignis verfolgt, welches Objekt es ausgelöst hat. Damit weiterhin eine Sortierung der Ereignisse möglich ist, muss auch die Ordnung auf den Ereignissen angepasst werden. Primäres Kriterium bleibt die Stelle an der das Ereignis auf einer der Achsen auftritt. Da aber für eine Stelle auf *einer* Achse alle Ereignisse auf einmal ausgewertet werden sollen, wird als zweites Kriterium die Achse eingesetzt. Der Ereignistyp bleibt als letztes Kriterium. Sodas gilt $e_1 < e_2$ falls einer der folgenden Bedingungen erfüllt ist:

- $e_1.t < e_2.t$
- $e_1.t = e_2.t \wedge e_1.axis < e_2.axis$
- $e_1.t = e_2.t \wedge e_1.axis = e_2.axis \wedge e_1.type < e_2.type$

Für eine effizientere Suche muss zunächst der Algorithmus 4.2 so angepasst werden, dass er auf einer, nach den angepassten Kriterien sortierten Liste operiert. Damit fallen als erstes die Zeilen sechs bis vierzehn heraus, da diese Liste als Eingabeparameter erwartet wird und gerade nicht jedesmal erzeugt werden soll. Die Schleife über alle Achsen (Zeile fünf) entfällt ebenfalls, da die Liste bereits die Ereignisse aller Achsen enthält. Damit bei der Auswertung eines Kandidaten für eine Schnittebene immer nur die Ereignisse entlang einer Achse gleichzeitig betrachtet werden, müssen abschließend die Bedingungen der inneren `while`-Schleifen um die Bedingung $event.axis == candidate_{axis}$ erweitert werden.

Der so angepasste Algorithmus funktioniert aber nur, wenn er als Eingabe eine bereits (aufsteigend) sortierte Liste aller Ereignisse für den aktuellen Voxel erhält. Damit die Liste für zwei neue Subvoxel nicht aus den Objekten neu erstellt und sortieren werden muss, wird die Ereignisliste des aktuellen Voxels als Grundlage verwendet. Nachdem die Trennposition *split* gefunden wurde, muss zunächst für jedes Objekt festgestellt werden, welchen der Subvoxel es überlappt (gegebenenfalls beide). Dazu wird zunächst für alle Objekte konservativ angenommen, dass sie beide Voxel schneiden. Durch Iteration über die Ereignisliste lassen sich die Objekte leicht klassifizieren. Für “START”-Ereignisse die

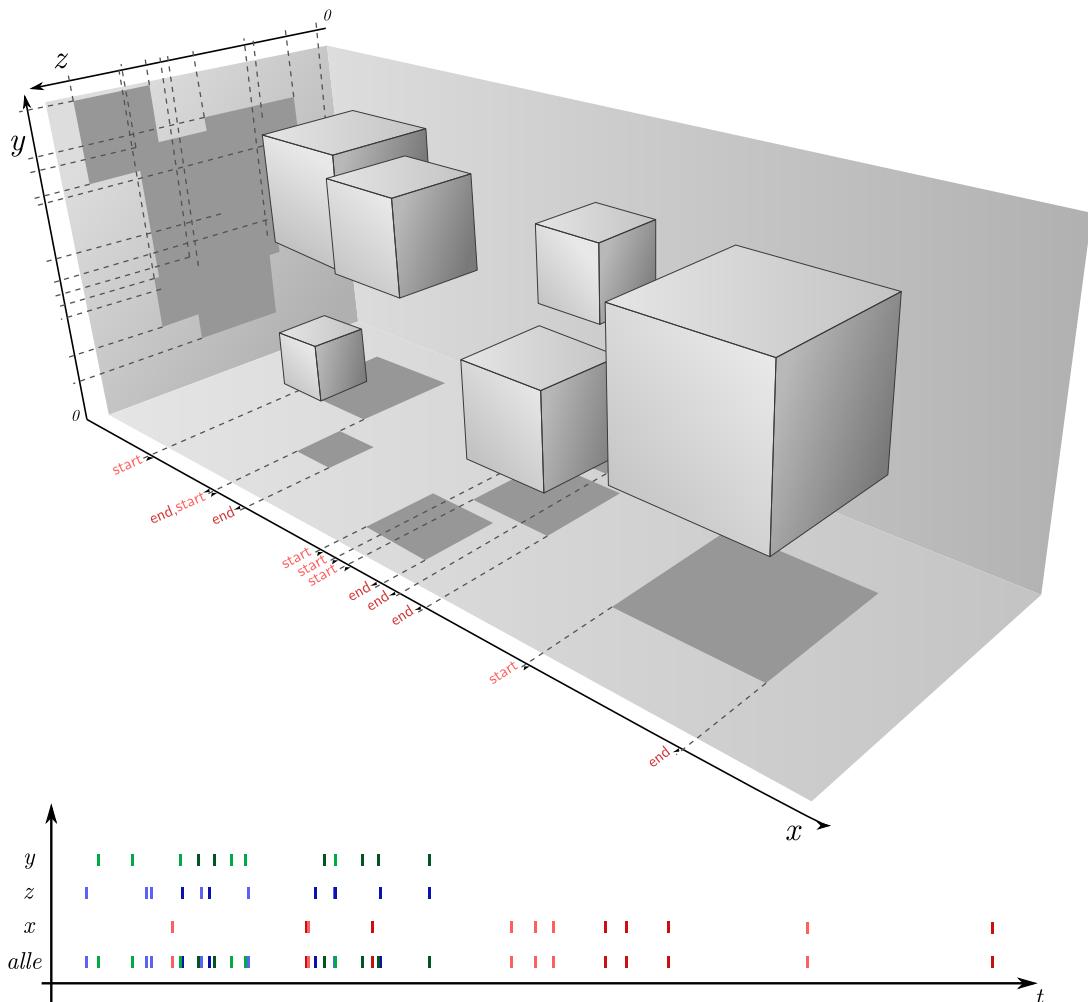


Abbildung 4.4: Das Minimum einer AABB wird zu einem START-Ereignis, das Maximum zu einem ENDE-Ereignis. Die Ereignisse aller Achsen werden gemeinsam geschachtelt in einer Liste gespeichert.

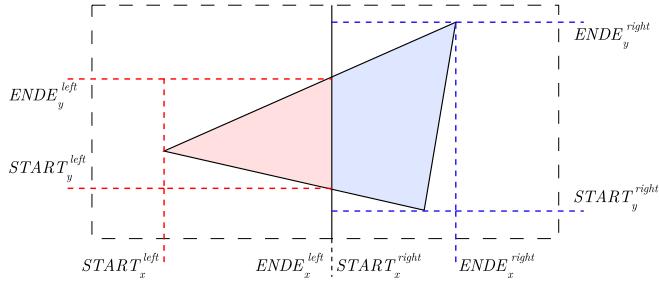


Abbildung 4.5: Ein Dreieck welches von der Trennebene geschnitten wird erzeugt Ereignisse in beiden Subvoxeln.

nach oder in der Trennungsebene liegen kann das zugehörige Objekt als ausschließlich im rechten Voxel liegend markiert werden. Für ‘‘Ende’’-Ereignisse, die vor oder in der Trennungsebene liegen, wird das entsprechende Objekt als ausschließlich im linken Voxel liegend markiert. Objekte planarer Ereignisse werden ebenfalls, je nachdem ob sie vor oder hinter der Trennungsebene liegen, ausschließlich für den linken oder rechten Teilbaum markiert. Für plane Objekte deren Ereignis entlang der Trennachse in der Trennungsebene liegt, wird das bereits vorgestellte Kriterium, ob einer der beiden Voxel gar keine Objekte enthält angewandt. In einer weiteren Iteration über die Ereignisse werden nun zwei Teillisten erstellt - eine für den linken und einen für den rechten Voxel. Dabei werden zunächst lediglich die Ereignisse aus der Gesamtliste in die entsprechende Teilliste übernommen, deren Objekte als vollständig im linken oder rechten Voxel liegend markiert wurden. Objekte, die von der Trennungsebene geschnitten werden, tragen unterschiedliche Ereignisse zu den beiden Listen bei. Es werden nicht nur die bestehenden Ereignisse dieser Objekte aufgeteilt sondern auch neue Ereignisse erzeugt. Dies ist notwendig, da das Objekt in beiden Voxeln ‘‘beginnen’’ und ‘‘enden’’ muss. Die überlappenden Objekte werden hierfür an den Subvoxeln geklippt. Die Minima und Maxima der geklippten Teilobjekte erzeugen die neuen Ereignisse für die jeweilige Teilliste (siehe Abbildung 4.5). Die neu erzeugten Ereignisse werden für den linken und rechten Teilbaum in einer eigenen Liste gespeichert. Diese verhältnismäßig kurzen Listen lassen sich schnell sortieren. Wald u. Havran (2006) gehen davon aus, dass lediglich $O(\sqrt{N})$ Objekte die Trennungsebene überlappen. Die Ereignisse hierfür lassen sich demnach in $O(N)$ sortieren. Danach müssen für beide Teilbäume die Liste der ‘‘alten’’ mit der Liste der ‘‘neuen’’ Ereignisse zusammengeführt werden. Da beide Listen sortiert sind, lässt sich auch dies in $O(N)$ umsetzen.

Der von Wald u. Havran (2006) entwickelte Algorithmus besitzt die algorithmisch niedrigste Komplexität aller bekannten Algorithmen zur Konstruktion kompletter SAH

basierter Hierarchien. Er eignet sich allerdings nicht für “lazy builds”, da der Sortiervorgang am Anfang alle Objekte berücksichtigt.

Sampling SAH

Um die Kosten für die Konstruktion weiter zu senken, kann die SAH Auswertung an weniger Stellen vorgenommen werden. Hunt u. a. (2006) schlagen vor die Anzahl der Primitive zur Linken und zur Rechten der Trennebene an q Positionen auszuwerten. Weitere q Samples werden in den Bereichen ausgewertet in denen die Differenz dieser Werte am größten ist. Durch lineare Interpolation der Objektanzahlen ergibt sich eine stückweise quadratische Kostenfunktion. Das Minimum dieser Funktion wird dann als Position für die Schnittebene gewählt.

Alternativ präsentieren Popov u. a. (2006) ein Verfahren, das die Kostenfunktion linear, dafür aber mit 1024 Samples abtastet. Durch die hohe Anzahl der Samples wird eine Verschlechterung der Qualität des konstruierten Baumes weitestgehend vermieden. Das Hauptziel des Verfahrens ist es jedoch, den zufällig verteilten Speicherzugriff anderer Konstruktionssverfahren zu vermeiden. Mehr zur Optimierung von Speicherzugriffen wird in Kapitel 5 erläutert.

Globale Heuristiken

Unter globalen Heuristiken versteht man solche, die zur Partitionierung die Szenengeometrie nicht analysieren. Die klassischen Vertreter dieser Kategorie sind der Median der Objektliste und der Median der Szenen-AABB entlang der längsten Achse. Da hier keine Operationen auf den in einem Knoten enthaltenen Objekten ausgeführt werden müssen, lassen sich hiermit Objekthierarchien schneller als mit jedem SAH Verfahren konstruieren. Der Nachteil dieser Heuristiken ist, dass die resultierenden Hierarchien sich nicht so gut an die lokale Ausprägung der Geometrie anpassen. Strahlaufragen können deswegen meist nicht so effizient beantwortet werden, wie von Datenstrukturen, die mit der SAH konstruiert wurden. Für Anwendungen in denen die Beschleunigungsdatenstruktur jedes Frame neu konstruiert werden muss, kann die Kombination einer “schnelleren, globalen Konstruktion” mit einer “langsameren” Traversierung unter Umständen den effizienteren Kompromiss darstellen. Gerade bei dynamischen Szenen mit extrem vielen Objekten profitiert die Gesamtleistung von einer sehr schnellen Konstruktion (Wächter u. Keller (2006)). Im Gegenzug kann sich eine langsamere Konstruktion in einer Situation in der extrem viele Strahlen ausgesendet werden (z.B. durch hohe Auflösung) durch eine schnelle Traversierungsphase amortisieren.

4.3 Zusammenfassung

Die Anforderung, dynamische Szenen darzustellen, macht die Beantwortung der Frage nach dem *besten* Verfahren nahezu unbeantwortbar. Die bereits totgesagten Verfahren der Partitionierung der Objektmenge erfreuen sich neuer Beliebtheit, dank ihrer Fähigkeit sich veränderten Objektpositionen anzupassen ohne die Topologie neu aufbauen zu müssen. Auch die Wahl der Heuristik zur Konstruktion ist nicht allgemein zu beantworten. Zu viele Faktoren beeinflussen die Gesamtleistung eines Raytracers. Szenengröße, Auflösung des zu rendernden Bildes, relative Objektgröße und nicht zuletzt die Art in der die Szene spezifiziert wird (z.B. Szenegraph oder "Suppe" von Dreiecken) haben großen Einfluss auf die Effizienz der beschriebenen Verfahren. Aus diesem Grund analysieren viele Verfahren nur den Fall indem die Szene durch eine ungeordnete Menge von Dreiecken - die so genannte *Triangle Soup* - jedes Frame neu spezifiziert wird. Solche allgemeinen Lösungen werden jedoch immer von, für einen Spezialfall angepassten Lösungen, übertraffen werden. So nutzten beispielsweise [Djeu u. a. \(2007\)](#) einen durch die Anwendung zu spezifizierenden Szenengraphen um schnell effiziente kd-Trees zu konstruieren.

5 Hardware

Als vor über 25 Jahren die Grundlagen für das Raytracingverfahren entwickelt wurden, unterschied sich die Hardware auf der diese implementiert wurden stark von den Rechnern die heute zu Verfügung stehen.

Damals war ein Zugriff auf den Hauptspeicher ungefähr so schnell wie die Ausführung einer einfachen Operation. In den letzten Jahrzehnten haben die Hersteller die Frequenz mit der ein Prozessor Operationen ausführen kann immer weiter erhöht. Die Geschwindigkeit mit der Daten aus dem Arbeitsspeicher transportiert werden können hat sich allerdings nicht proportional mitentwickelt.

Dies liegt maßgeblich daran, dass für den Hauptspeicher Speicherbausteine verwendet werden, welche die Daten nicht in ausreichend hoher Geschwindigkeit zur Verfügung stellen können. Die Verwendung schnellerer Bausteine ist prinzipiell möglich, jedoch nicht ökonomisch, da diese Lösungen um Größenordnungen teurer wären, als die eingesetzten. ([Drepper, 2007](#))

Eine Konsequenz dieser Architektur ist, dass Probleme wie das Raytracing nun im Gegensatz zu früher nicht mehr durch die Rechenleistung begrenzt werden, sondern durch die Datenmenge, die in kurzer Zeit zum Prozessor übertragen werden kann.

Natürlich sollte Softwareentwicklung so unabhängig wie möglich von der konkreten Hardware sein. Für interaktives Raytracing auf aktuellen PCs ist es jedoch unumgänglich gewisse Strukturen der Hardware zu kennen und bei der Programmierung zu berücksichtigen. In diesem Kapitel wird erläutert welche Maßnahmen Hardwarehersteller treffen um das Verhältnis zwischen Prozessor- und Speicherzugriffsgeschwindigkeit zu verbessern und wie dies vom Softwareentwickler unterstützt werden kann. Es ist anzunehmen, dass die vorgestellten Konzepte, wie Caches, mindestens mittelfristig auch weiter bei Prozessoren von Arbeitsplatzrechnern Anwendung finden werden.

5.1 Cache

Das wichtigste Konzept zur Verbesserung der oben beschriebenen Situationen sind Caches. Ein Cache ist ein Datenspeicher der aus Bauteilen besteht die Daten in höherer

Geschwindigkeit zur Verfügung stellen können als der Arbeitsspeicher.

Welchen Unterschied die theoretisch optimale Nutzung des Caches macht, zeigt ein kleines Rechenbeispiel aus Drepper (2007):

Angenommen, dass der Zugriff auf den Hauptspeicher 200 Zyklen benötigt und der Zugriff auf den Cache 15. Ein Programm, das auf 100 Datenelemente 100mal zugreift, benötigt 2.000.000 Zyklen für Speicherzugriffe und lediglich 168.500 Zyklen wenn alle Daten gecacht werden können. Dies entspricht einer Verbesserung von 91.5%.

Da diese Bausteine wesentlich teurer sind als die Hauptspeicherbausteine, fällt der Cache meist mehrere Größenordnungen kleiner aus als der Hauptspeicher (2GB Arbeitsspeicher - 2MB Cache). Da die Datensätze, die heutzutage während des Programmverlaufs verarbeitet werden müssen, viel größer sind als der zur Verfügung stehenden Caches, bedarf es guter Strategien um immer die Daten in den Cache zu laden, die als nächstes benötigt werden. Dadurch, dass parallel zu anderen Operationen des Prozessors Daten aus dem Hauptspeicher in den Cache geladen werden können, ist es möglich die langsame Hauptspeicher-Zugriffzeiten zu verstecken.

Die im Cache gespeicherten Datensätze besitzen eine gröbere Granularität als die des Arbeitsspeichers. Das liegt zum einen daran, dass die Hauptspeicher-Bausteine effizienter arbeiten wenn mehrere Datenwörter am Stück ausgelesen werden. Zum anderen muss der Cache neben den Daten selbst, auch noch die Adresse der Daten im Arbeitsspeicher halten. Die Adresse jeden adressierbaren Datenwortes, welches im Cache abgelegt wird, zu speichern, wäre ineffizient. Ein unnötig großer Teil des teuren Cachespeichers wäre mit Adressen belegt. Deswegen ist der Cache in 'Lines' organisiert, welche größer sind als das kleinste durch den Prozessor adressierbare Datenwort. Um wieviel größer hängt vom konkreten Prozessortyp ab. Bei der Umsetzung von Cache-Lines wird zusätzlich angenommen, dass Daten auf die kurz hintereinander zugegriffen wird ein *räumlichen Nähe* im Speicher besitzen. Diese Annahme ist durchaus berechtigt, denn für die folgenden zwei, häufig auftretenden Fälle trifft genau das zu:

Zusammengesetzte Datentypen Oft werden einfache Datentypen zu komplexen, strukturierten Typen zusammengefasst. Diese bilden dann die "kleinsten" Einheiten die von vielen Teilen eines Programms verarbeitet werden und damit auch zwischen Hauptspeicher und Prozessor ausgetauscht werden. Durch die Kombination mehrerer Felder kann ein Exemplar schnell auf die Größe einer Cacheline anwachsen. Wird auf die verschiedenen Felder zugegriffen profitiert dies vom Caching, da das ganze Objekt in der entsprechenden Cacheline bereits vorliegt.

Sequenzen kleiner Datensätze Ein einzelner Buchstabe einer Zeichenkette belegt nur den Bruchteil einer Cacheline. Für solche kleinen Datensätze kommt es jedoch häufig vor, dass sie in großen, sequentiell gespeicherten Blöcken vorliegen. Zum Beispiel profitiert die sequentielle Ausgabe einer Zeichenkette davon, dass selbst wenn diese nicht im Cache liegt, nach Einlesen des ersten Zeichens, die folgenden Zeichen bereits im Cache liegen werden.

Die zweite Annahme, nach der Caches gestaltet sind, ist die der *zeitlichen Nähe*. Das heißt, es wird davon ausgegangen, dass Daten auf die vor kurzem zugegriffen wurde, in kurzer Zeit wieder gelesen oder geschrieben werden. Dies trifft insbesondere für Programmcode zu. So kann es zum Beispiel passieren, dass innerhalb einer Schleife durch einen Funktionsaufruf immer wieder zu einer bestimmten, entfernten Stelle im Speicher gesprungen wird, um das dort liegende Stück Code auszuführen. Durch die von-Neumann'sche Rechnerarchitektur, nach der die meisten der Arbeitsplatzrechner entworfen sind, liegen der Programmcode und die Daten gemeinsam im Hauptspeicher. Da die Verarbeitung jedoch verhältnismäßig unabhängig voneinander geschieht, hat es sich als vorteilhaft erwiesen getrennte Caches für Programmcode und Programmdaten einzuführen. Dies ermöglicht zusätzlich die Dekodierphase für einen Prozessorbefehl im Cache zu überspringen, indem nach der initialen Ausführung der dekodierte Befehl in den Cache geschrieben wird. (Drepper, 2007)

Sowohl das Cachen von Programmdaten, als auch das Pipelining (siehe unten Abschnitt 5.2.4) setzen voraus, dass präzise und möglichst weit im Voraus vorhergesagt werden kann, welche Daten, beziehungsweise Befehle, geladen werden müssen. Kann dieses nicht gewährleistet werden, kommt es zu so genannten *Pipeline stalls*, das heißt der Prozessor muss unter Umständen mehr als 100 Zyklen auf Daten warten und führt während dieser Zeit keine Befehle aus.

Um dies zu vermeiden wird viel Energie in die Optimierung der Vorhersage, welcher Code als nächstes ausgeführt wird, investiert. Die entwickelten Verfahren liefern jedoch nur den vollen Effekt, wenn bei der Softwareentwicklung gewisse Richtlinien beachtet werden, auf die im Folgenden näher eingegangen wird. Allgemein lässt sich die bestmögliche Erfüllung der Annahme der *räumlichen* und *zeitlichen* Nähe als Richtlinie formulieren. Da die Erfüllung, beziehungsweise Verletzung dieser Annahmen jedoch häufig nicht offensichtlich ist, werden im Folgenden ein paar konkretere Richtlinien aufgezeigt.

5.2 Richtlinien

Die folgenden Richtlinien sollen dabei helfen, effizienten Code für aktuelle Arbeitsplatzrechner zu erstellen. Leider stehen diese Richtlinien teilweise im Gegensatz zu denen des allgemeinen Softwaredesigns. Umso wichtiger ist es diese Optimierungen an den relevanten Stellen einzusetzen. Die Frage, die beim Raytracing am häufigsten beantwortet werden muss, ist die nach dem Schnittpunkt eines Strahls mit der Szenengeometrie. Der Programmcode, der dieses Problem behandelt, wird am häufigsten ausgeführt und stellt deswegen den Fokus für die folgenden Maßnahmen dar. Neben einer kurzen Erläuterung einer Richtlinie wird auch die Anwendung dieser Richtlinie für ein Raytracingprogramm dargestellt. Für die meisten Richtlinien wird eine Entwicklung in C, beziehungsweise C++ angenommen. Programmiersprachen, die weiter von der Hardware abstrahieren, lassen viele der vorgestellten Maßnahmen nicht zu. Die Implementierung eines so komplexen Systems, wie das eines Raytracers, in Assembler, würde wiederum zu einer unverantwortlichen Codebasis führen.

5.2.1 Kompaktheit

Die Datenmenge, auf die im Verlauf eines Programms zugegriffen wird, auch als “working set” bezeichnet. Passt das *working set* komplett in den Cache, muss zur eigentlichen Laufzeit nicht auf den Hauptspeicher zugegriffen werden. Damit würden auch keine Verzögerungen durch die hohen Zugriffszeiten entstehen. Üblicherweise, und besonders für das Raytracing, übersteigt der Umfang des *working sets* das Fassungsvermögen des Caches jedoch um mehrere Größenordnungen.

Wird für die Datenstrukturen, die vom Programm verwendet werden, darauf geachtet diese möglichst kompakt zu gestalten, können mehr Datensätze im Cache gehalten werden. Damit wird die Wahrscheinlichkeit eines *Cache-hits* erhöht. Darunter versteht man den Fall, dass ein benötigter Datensatz bereits im Cache liegt und nicht aus dem Hauptspeicher geladen werden muss (im Gegensatz zu einem *Cache-miss*).

Bei der Schnittpunktbestimmung sind die am häufigsten benötigten Datensätze die Knoten einer Beschleunigungsdatenstruktur. Zur kompakteren Darstellung lassen sich verschiedene Techniken anwenden. Oft kann man mehrere Flags (zum Beispiel ob ein Knoten ein innerer Knoten oder ein Blatt ist) gemeinsam in einem Byte speichern, anstatt für jedes Flag ein einzelnes Feld in der Struktur anzulegen. Beispiele werden in Kombination mit der Anwendung der in Abschnitt 5.3 erläuterten Richtlinien vorgestellt.

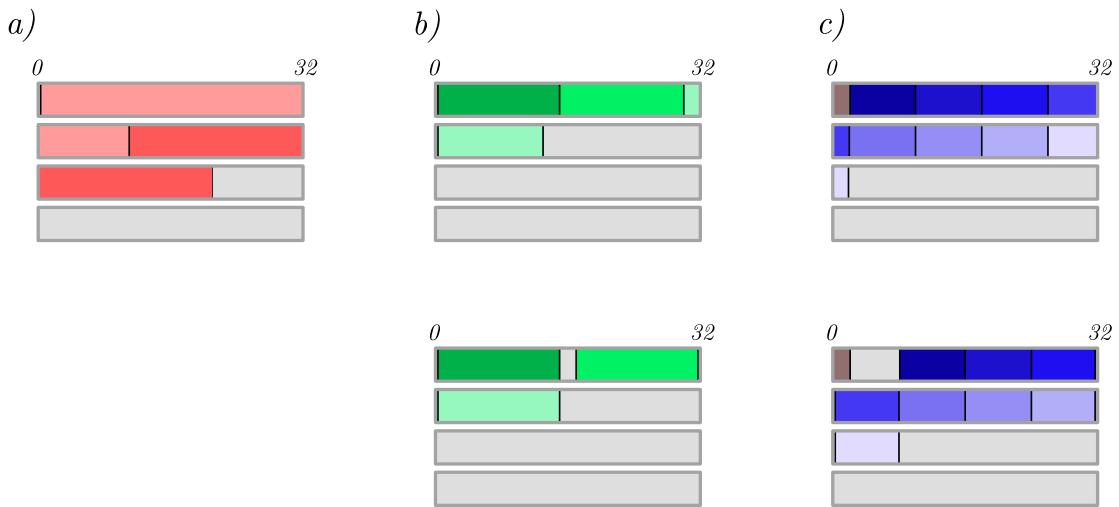


Abbildung 5.1: Alignment unterschiedlicher Datenstrukturen. a) Ein Exemplar ist größer als eine Cacheline. Es kann nicht verhindert werden, dass für ein Element min. zwei Cachelines geladen werden müssen. b) oben: Ein Exemplar belegt 15 Bytes. Durch lückenlose Speicherung beginnt das dritte Element in der ersten und ragt in die zweite Cacheline. unten: Durch ein 16Byte Alignment werden kann die Überlappung verhindert werden, erzeugt aber insgesamt einen wachsenden Speicherbedarf linear zur Menge der gespeicherten Elemente. c) oben: Durch das fehlende Alignment wird hier jedes vierte Element zwei Cachelines überlappt. unten: Durch das 8Byte-Alignment wird nach Daten am Anfang eine 6Byte Lücke erzwungen. Danach können die Datensätze ohne Überlapp und weitere Lücken gespeichert werden.

5.2.2 Ausrichtung

Durch den Compiler wird die Möglichkeit gegeben Exemplare einer Datenstruktur nur an Adressen ganzzahliger Vielfacher eines wählbaren Wertes zuzulassen. Dies entspricht der Ausrichtung einer Datenstruktur auf ein bestimmtes Raster (4,8,12,16 Bytes etc). Durch eine geschickte Wahl dieses so genannten *Alignments* kann man verhindern, dass ein Datenwort, welches kleiner als eine Cacheline ist, zwei Cachelines überlappt. Dies ist zum Einen bei einem Cache-Miss vorteilhaft, denn beim Laden der Daten muss lediglich eine Cacheline geladen werden. Gleichzeitig wird lediglich eine andere Cacheline überschrieben - und somit implizit die Wahrscheinlichkeit für einen Cache-hit des in der nicht überschriebenen Cacheline enthaltenen Datensatzes, erhöht. Zum Anderen haben durch die Ausrichtung die niedrigen Bits eine Adresse eines solchen Datensatzes dann immer den Wert 0. In diesen Bits können wiederum kleinere Werte, wie Flags, gespeichert werden. Auf der anderen Seite führt ein Alignment, das größer ist als die betreffende Datenstruktur, zu entsprechend viel ungenutztem Speicher. Abbildung 5.1 zeigt die Konsequenzen unterschiedlicher Ausrichtungen verschiedener Datenstrukturen.

5.2.3 Trennung von Daten

Um die räumliche Nähe von Daten, auf welche zur gleichen Zeit zugegriffen wird, beizubehalten, müssen manchmal Datenstrukturen aufgetrennt werden. Gibt es Datenfelder, die semantisch zu einem Typ gehören, wovon eines aber besonders häufig gelesen wird, kann es sich lohnen die beiden Datenfelder unabhängig zu speichern. So werden die nicht so häufig gelesenen Felder nicht automatisch mit in den Cache geladen, und es steht mehr Platz für die Daten, auf die öfter zugegriffen wird, zur Verfügung.

Insbesondere gilt dies für Fälle in denen auf ein Feld eines strukturierten Datentypen lediglich lesend zugegriffen wird, während auf einem zweiten Feld häufig schreibende Zugriffe erfolgen. Auch schreibende Zugriffe erfolgen in ganzen Cachelines. Wenn kleine Teile des Inhalts einer Cacheline verändert werden, muss trotzdem die gesamte Cacheline zurück in den Hauptspeicher geschrieben werden. Damit würde die Bandbreite beim Schreiben durch die nicht veränderten Daten unnötig belastet.

Da diese Vorgehen die Komplexität des Programmcodes erheblich erhöhen kann, sei darauf hingewiesen, dass diese Optimierung sich nur in vereinzelten Fällen lohnt. Sie sollte ausschließlich angewendet werden wenn die Größe des *working sets* die des Caches übersteigt, die Daten sequentiell gespeichert werden und auf diese in kritischen Programmabschnitten häufig zugegriffen wird. In keinem Fall sollte dieses Vorgehen zum Regelfall bei dem Entwurf von Datenstrukturen werden.

Ein Beispiel für die korrekte Anwendung dieser Optimierung ist das Mailboxing für disjunkte Raumaufteilungsverfahren. Innerhalb eines Frames ändern sich die Geometriedaten nicht. Die Mailboxen einiger Objekte werden unter Umständen jedoch für jeden Strahl verändert. Obwohl die Mailbox semantisch zu dem Objekt gehört, lohnt es sich die Mailboxen gesondert zu speichern. Dadurch kann verhindert werden, dass die Objektdaten, die sich in der selben Cacheline wie die Mailboxingdaten befinden, jedes mal mit in den Hauptspeicher geschrieben werden.

5.2.4 Befehlscache

Wie bereits erwähnt besitzen Prozessoren seit mehreren Jahren getrennte Caches für Programmbefehle und Daten. Die optimale Nutzung des Befehlscaches ist sogar noch wichtiger als die des Datencaches. Wenn der erwartete Programmteil nicht im Cache liegt muss der Prozessor im schlimmsten Fall die Latenz für einen Hauptspeicherzugriff abwarten bis wieder Befehle ausgeführt werden. Eventuelle Latenzen, die sich aus Datencache-misses der enthaltenen Befehle ergeben, addieren sich entsprechend auf. Da ein Compiler für die letztendliche Codegenerierung zuständig ist, hat der Programmierer

meist keinen direkten Einfluss auf die Nutzung des Befehlscaches. Durch die richtige Verwendung des Compilers lässt sich aber indirekt eine bessere Nutzung erreichen. Generell gilt die Richtlinie der Kompaktheit genauso für den Programmcode wie für die verwendeten Daten. Die häufigste Quelle für erhöhte Wartezeiten sind Sprünge deren Ziel nicht statisch bestimmbar ist oder deren Ziel nicht im Befehlscache liegt.

Inlining

Daher sollten, besonders bedingte, Sprünge vermieden werden. Um den Sprung in eine entfernte Funktion zu vermeiden bietet der Compiler die Möglichkeit des *Inlining*. Wird eine Methode mit dem Schlüsselwort `inline` gekennzeichnet, ist dies ein Hinweis für den Compiler keine neue Funktion im klassischen Sinne anzulegen, sondern den enthaltene Code direkt an die Stelle des Aufrufs einzufügen. Ob der Hinweis befolgt wird hängt vom jeweiligen Compiler ab, da es sich wie gesagt nur um einen Hinweis handelt.

Die Verwendung von `inline` ist jedoch ein zweischneidiges Schwert. Einerseits spart man einen Funktionsaufruf und die damit verbundenen Kosten. Auf der anderen Seite kann die Menge des Programmcodes stark ansteigen was im Widerspruch zur Richtlinie der Kompaktheit steht. Verschiedene Autoren ([Drepper \(2007\)](#), [Meyers \(2006\)](#)) sind sich einig, dass sich Inlining hauptsächlich in zwei Fällen lohnt:

- Für sehr kurze Methoden (zum Beispiel Getter und Setter)
- Für Methoden die nur einmal aufgerufen werden.

Für alle anderen Fälle sollte `inline` nur im Zusammenhang mit genauem Profiling verwendet werden, weil der Code unübersichtlicher und das Debuggen erschwert wird. Im schlechtesten Fall kann die Verwendung von `inline` sogar Leistungseinbußen mit sich bringen.

Da für die meisten Compiler das Inlining in der Kompilierungsphase geschieht, muss die Funktionsdefinition auch in der Headerdatei vorgenommen werden, was wie bereits angedeutet, die Lesbarkeit herabsetzen kann. Um die Lesbarkeit so wenig wie möglich zu verschlechtern sollten die mit `inline` markierten Methoden außerhalb der Klassendeklaration implementiert werden (siehe Quelltext [5.1](#)).

Dynamische Bindung

In C++ gibt es die Möglichkeit Methoden mit dem Schlüsselwort `virtual` zu markieren. Dadurch wird dem Entwickler ermöglicht eine polymorphe Klassenhierarchie zu implementieren. Da die Sprungadresse virtueller Methoden aber erst zur Laufzeit berechnet

Quelltext 5.1: links: Definition aller Inlinemethoden mindert Lesbarkeit. rechts: Getrennte Definition der Methoden macht Schnittstelle der Klasse klar erkennbar

<pre>class BadStyle { public: void methodA(); inline void methodB() { doSomething(); } void methodC(); }</pre>	<pre>class GoodStyle { public: void methodA(); void methodB(); void methodC(); } // Ende der Klassen Deklaration inline void GoodStyle::methodB() { doSomething(); }</pre>
---	---

wird, sollte auf die Verwendung in kritischen Programmteilen verzichtet werden. Für das Raytracing heißt das konkret, dass in der Phase der Schnittpunktbestimmung keine virtuellen Methoden aufgerufen werden sollten. [Wald \(2004\)](#) berichtet, dass der Aufruf einer einzigen virtuellen Methode einen Leistungseinbruch von 5% zur Folge hatte.

Da die Verwendung von virtuellen Methoden maßgeblich zu einer guten Programmstruktur beiträgt, sei auch hier nochmal darauf hingewiesen, dass solche Maßnahmen ausschließlich in leistungskritischen Programmabschnitten anzuwenden sind.

Pipeline stalls

Da die Prozessoren aber selbst schneller als die Cachezugriffe sind, wird zur Befehlsverarbeitung das so genannte Pipelining angewendet. Hierbei wird die Ausführung eines Befehls in Teilphasen zerlegt. Diese bestehen zum Beispiel aus einer Phase zum Laden des Befehls und einer zum Dekodieren. Dazu kommen weitere Phasen zum Laden der Operatoren, die Verarbeitung und das Speichern der Ergebnisse ([Rechenberg u. Pomberger \(2002\)](#)). Durch parallele Ausführung der einzelnen Phasen unterschiedlicher Befehle lassen sich die Latenzen für das Laden einzelner Befehle verstecken (siehe Abbildung 5.2).

Als Beispiel nehmen wir eine vierstufige Pipeline an, von der jede Stufe zur Ausführung einen Takt benötigt. Für die Ausführung wird also jeder Befehl in vier Phasen zerlegt, die jeweils einen Takt dauern. Ein Befehl hat somit eine *Latenz* von vier Takten. Die sequentielle Ausführung von vier Befehlen würde 16 Takte dauern. Jede Phase wird von einem eigenständigen Modul ausgeführt. Dadurch wird es möglich, unterschiedliche Phasen aufeinander folgender Befehle parallel auszuführen. Der *Durchsatz* von Befehlen kann so deutlich erhöht werden. Für das Beispiel der vierstufigen Pipeline heißt das, dass die vier Befehle, anstatt in 16, in lediglich 7 Takten ausgeführt werden können.

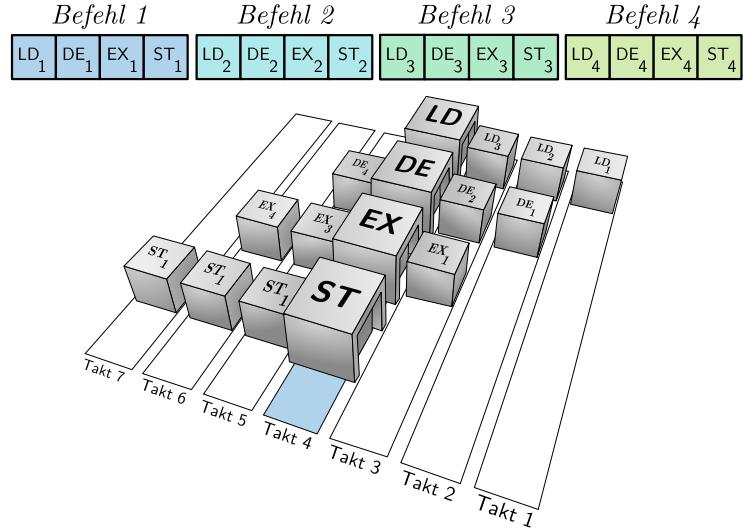


Abbildung 5.2: oben: Jeder Befehl wird in vier Phasen zerlegt. unten: Durch parallele Ausführung verschiedener Phasen unterschiedlicher Befehle (LD_4, DE_3, EX_2, ST_1) lässt sich der Durchsatz erhöhen ($LD = \text{Befehl laden}, DE = \text{dekodieren}, EX = \text{Ausführen}, ST = \text{Speichern}$)

Das Pipelining liefert aber nur den vollen Durchsatz, wenn die Pipeline die ganze Zeit gefüllt bleibt. Nach einem bedingten Sprung können zwei verschiedene Folgen von Befehlen ausgeführt werden. Die Befehlspipeline kann aber lediglich eine Folge aufnehmen. Eine falsche Vorhersage führt dazu, dass die bereits in der Pipeline vorhandenen Befehle ungültig sind. Für diesen Fall muss die Pipeline komplett geleert werden. Der volle Durchsatz kann erst wieder erreicht werden, wenn der erste Befehl alle Stufen der Pipeline durchlaufen hat. Diese so genannte *misprediction penalty* kann aufgrund der unterschiedlichen Länge der Pipeline bei verschiedenen Prozessoren von 12 bis über 50 Takte umfassen [Fog \(2008b\)](#). Dies ist zwar nicht so verheerend wie eine schlechte Ausnutzung des Caches, aber trotzdem signifikant an den kritischen Stellen eines Programms. Die Hardwarehersteller verwenden dementsprechend viel Kapazitäten, um eine gute Sprungvorhersage zu erreichen. Solange die Verzweigungen regelmäßigen Mustern folgen funktioniert diese recht gut. Trotzdem ist die einfachste Methode falsche Vorhersagen zu vermeiden, in kritischen Abschnitten so wenig Verzweigungen wie möglich zu verwenden.

5.2.5 Rekursion gegenüber Iteration

Die iterative Formulierung eines Algorithmus arbeitet meist effizienter als die rekursive Variante, da letztere zusätzliche Funktionsaufrufe benötigt ([Lantzman \(2007\)](#)). Deswegen

bietet es sich an, kritischen, rekursiven Code, wie zum Beispiel die Traversierung der Raumaufteilungshierarchie, dementsprechend umzuformulieren ([Wald \(2004\)](#)).

5.3 Effiziente Speicherauslegung

Besonders wichtig für eine effiziente Berechnung der Schnittpunkte der Strahlen mit der Szenengeometrie ist eine höchst effiziente Implementierung der Beschleunigungsdatenstruktur. Mindestens genauso wichtig wie die eigentlichen Berechnungen ist dabei die Auslegung der Struktur im Speicher. Für hierarchische Strukturen besteht diese zum Einen daraus, wie ein einzelner Konten ausgelegt ist, und zum Anderen wie die verschiedene Knoten relativ zueinander im Speicher angeordnet sind. Im Folgenden werden Optimierungen für *binäre* Bäume erläutert.

5.3.1 Allokation und Anordnung

Die Speicherung von Daten in einem Baum ist zwar *logisch* nicht linear, wird dies physikalisch aber spätestens beim Speichern im *linearen* Arbeitsspeicher. Durch die verschiedenen Möglichkeiten die Hierarchie im linearen Speicher anzugeordnen ergeben sich mehr oder weniger geeignete Strukturen. Besonders was die effiziente Cachenutzung betrifft kann es hier zu signifikanten Leistungsunterschieden kommen.

Bei der klassischen Top-down Konstruktion wird der Speicher während der Konstruktion für jeden Knoten einzeln angefordert. Der Elternknoten speichert dann zwei Referenzen auf die ihm folgenden Teilbäume (Abbildung 5.3 a)). Eine erste Optimierung für binäre Bäume ist das Speichern der zwei Kindknoten direkt hintereinander. Da von der Adresse des ersten Kindknotens, durch einfache Inkrementierung auf die Adresse des zweiten Knotens geschlossen werden kann, erübrigt sich eine explizite Speicherung der zweiten Adresse (Abbildung 5.3 b)). Alternativ kann auch einer der Kindknoten direkt nach dessen Elternknoten gespeichert werden. Auch hierbei kann auf die explizite Adressierung dieses Kindknotens verzichtet werden. Befürworter der zweiten Strategie argumentieren, dass es hier bei der Traversierung eine höhere Wahrscheinlichkeit gibt, dass der nächste zu traversierende Kindknoten bereits mit dem Elternknoten zusammen in den Cache geladen wurden (Abbildung 5.3 c)).

5.3.2 Kompakte Knotendarstellung

Für eine cacheeffiziente Implementierung müssen die Knoten besonders kompakt gespeichert werden. In [Benthin \(2006\)](#) wird ein effizientes Layout für einen kd-tree Knoten

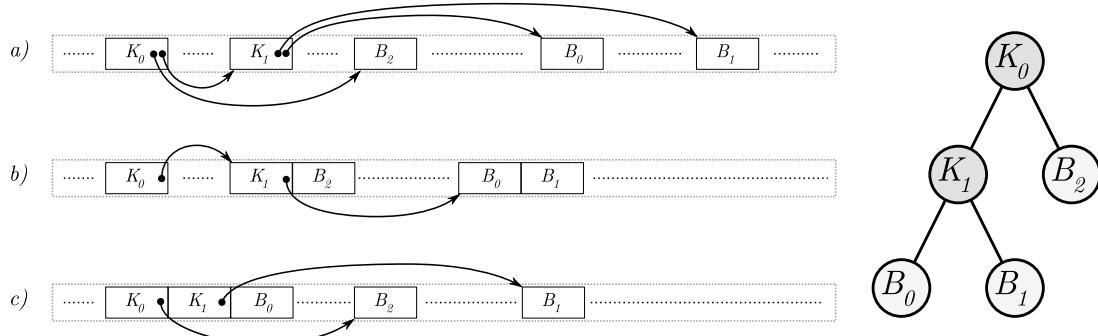


Abbildung 5.3: Drei Möglichkeiten den rechts gezeigten Baum im Speicher abzulegen. a) Die Implementierung mit zwei Zeigern pro innerem Knoten benötigt den meisten Platz und eine möglichst große Entfernung zu den Kindknoten, welches die Wahrscheinlichkeit eines Cache-miss erhöht. b) Das Speichern der Kinder hintereinander ermöglicht eine schlankere Darstellung. Falls beide Teilbäume traversiert werden müssen besteht eine erhöhte Wahrscheinlichkeit, dass der zweite Knoten einen Cache-hit erzeugt. Für Bäume, die größer als die Cache sind, tritt dies in den oberen Ebenen selten ein, da die Cacheline sehr wahrscheinlich schon ersetzt wurde. c) Die Auslegung im Speicher belegt den gleichen Platz. Allerdings gibt es eine 50/50 Chance, dass der Kindknoten, der jetzt direkt auf den aktuellen Knoten folgt, als nächstes traversiert wird und somit einen Cache-hit erzeugt.

Quelltext 5.2: BIH Knoten wie durch Waechter beschrieben. Belegt auf einer 32Bit Architkturen lediglich 12Byte entspr. 16Byte auf 64Bit Hardware

```

typedef struct {
    longIndex; // lowest bits : axis (00, 01, 10) or leaf (11)
union {
    long Items; // leaf only
    float Clip[2]; //internal node only
}
} BIH_Node;
  
```

Quelltext 5.3: kd-tree Knoten in einem acht Byte Layout nach Benthin bzw Wald (ausgehend von 32-Bit Architektur). Verschiedene Attribute werden durch effiziente logische Anweisungen in einem unsigned int kombiniert

```
// efficient 8-byte layout for a kd-Tree node
struct KDTreeNode {
    union
    {
        float split_position; // position of axis-aligned split plane
        unsigned int items; // or number of leaf primitives
    }
    unsigned int dim_offset_flag;
    // the 32 bits of 'dim_offset_flag' are used to encode multiple data
    // bits [0..1] : encode the split plane dimension
    // bits [2..30] : encode an address offset
    // bit [31] : encodes whether a node is an inner node or a leaf
};
// macros for extracting node information
#define ISLEAF(n) (n->dim_offset_flag & (unsigned int)(1<<31))
#define DIMENSION(n) (n->dim_offset_flag & 0x3)
#define OFFSET(n) (n->dim_offset_flag & 0x7FFFFFFC)
```

vorgestellt (siehe Quelltext 5.3). Ein gleichermaßen effizientes Layout für einen BIH-Knoten wird in Wächter u. Keller (2006) vorgestellt (siehe Quelltext 5.2). Für beide Auslegungen wird angenommen, dass lediglich ein Kindknoten eines inneren Knoten explizit referenziert werden muss. Anstatt die Unterscheidung zwischen einem Blatt und einem inneren Knoten, wie in der objektorientierten Programmierung üblich, durch dynamische Bindung umzusetzen, wird explizit ein entsprechendes Flag gespeichert. So kann während der Traversierung vollkommen auf die Verwendung virtueller Methoden verzichtet werden.



Fortgeschrittene Speicherauslegung

Wächter u. Keller (2007) stellen einen Konstruktionsalgorithmus für hierarchische Beschleunigungsdatenstrukturen vor, welcher die Anforderungen an ein effizientes Speichermanagement, für aktuelle Hardware erfüllt. Das Vorgehen ist sowohl für Verfahren, die den Raum disjunkt teilen, als auch für Verfahren, welche die Objektmenge partitionieren, geeignet. Das Verfahren nimmt eine Top-down-Konstruktion vor, welche durch Einführung eines neuen Terminierungskriteriums in einem festen Speicherblock ausgeführt werden kann. Dadurch kann der benötigte Speicher *einmalig* angefordert werden und weitere Allokationsaufrufe sind überflüssig. Das Verfahren ist mit jeder Heuristik zur Wahl der Partitionierung einsetzbar.

Die Vorgehensweise wird in Abbildung 5.4 für die dort dargestellte Szene veranschaulicht. Die Eingabe für den Algorithmus ist die Größe des Speicherblocks und der Zeiger auf den Speicherblock. Die Indizes zu den im Voxel enthaltenen Objekte befinden sich am Anfang des Blocks. Dabei werden die folgenden Schritte ausgeführt:

1. Das erste Element befindet sich links von der Trennebene und verbleibt auf seiner Position. Der Zeiger für das nächste zu bearbeitende Element wird inkrementiert.
2. Das zweite Element liegt vollständig hinter der Trennebene. Der Index wird an das Ende des Block geschrieben.
3. Das letzte Element der noch nicht untersuchten Objekte wird an die frei gewordene Stelle bewegt.
4. Das Objekt fünf überlappt beide Voxel. Der Bereich für Elemente, die in beide Voxel reichen, befindet sich direkt vor dem Bereich der Objekte, die ausschließlich im rechten Voxel liegen.
5. Der Zeiger für das nächste zu bearbeitende Element wird nicht weiterbewegt. Das Objekt, welches sich als letztes in der Liste der noch nicht untersuchten Objekte befindet, wird an die frei gewordene Stelle bewegt.
6. Auch das Objekt vier ragt in beide Voxel und wird an den Anfang des Bereichs für doppelt referenzierte Objekte bewegt.
7. Das letzte nicht klassifizierte Objekt in der Liste wird wieder an die Stelle des Zeigers für das nächste zu bearbeitende Element bewegt.
8. Das aktuell zu untersuchende Objekt drei befindet sich vollständig hinter der Trennebene. Um für dieses Element Platz zu schaffen wird das letzte Element aus der Liste der doppelt referenzierten Objekte, an den Anfang dieser Liste bewegt.
9. Das Objekt drei wird an den frei gewordenen Platz verschoben. Dort ist jetzt der neue Anfang für die Liste der Objekte, die ausschließlich im rechten Voxel liegen. Es sind keine weiteren Objekte in der Liste der nicht klassifizierten Objekte zu finden. Der Sortiervorgang ist abgeschlossen.
10. Die Elemente der Liste der doppelt referenzierten Objekte wird an das Ende der Objekte, die sich vollständig vor der Trennebene befinden kopiert.

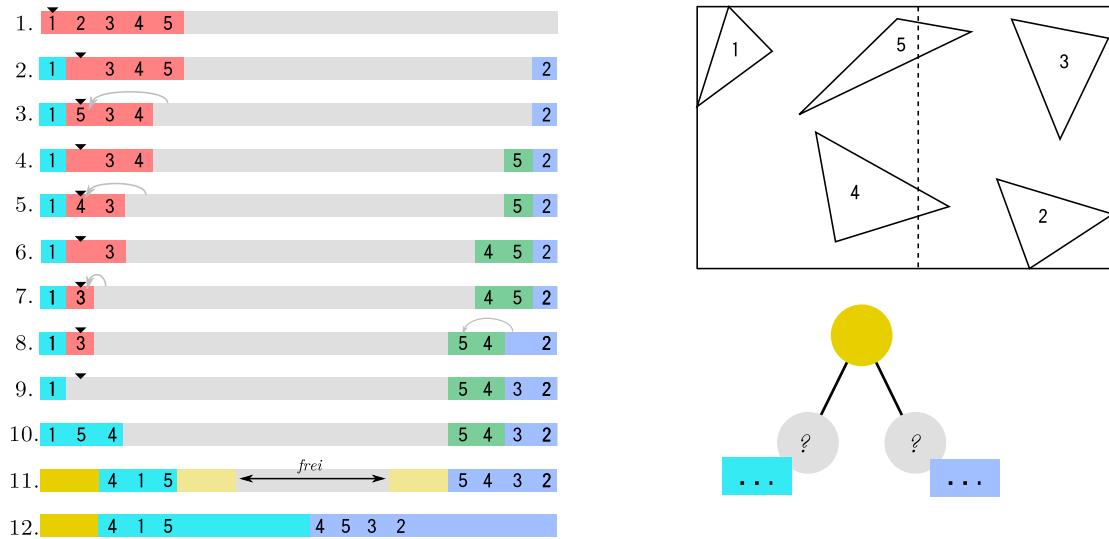


Abbildung 5.4: Vorgehensweise bei der Top-down-Konstruktion in einem festen Speicherblock nach Wächter u. Keller (2007)

11. Um für den aktuell zu erzeugenden inneren Knoten Platz zu schaffen werden so viele Objekt vom Anfang der linken Liste an deren Ende verschoben, dass am Anfang genug Platz für einen Knoten entsteht. Der freie Bereich zwischen den beiden Listen ist der Speicher, der beiden Teilbäumen gemeinsam zur Verfügung steht, um den Raum weiter aufzuteilen. Er wird proportional zu den, in den Teilbäumen enthaltenen, Objekten aufgeteilt. Der Speicherplatz für die beiden Kindknoten selbst, wird aber zuvor davon abgezogen, da dieser auf jeden Fall für jeden der Teilbäume zur Verfügung stehen muss.
12. Die Liste der Objekte für den rechten Teilbaum wird nun um die Menge des dem rechten Teilbaum zugewiesenen freien Speichers nach vorne verschoben.

Danach kann das Verfahren für die beiden eingezeichneten Teilblöcke von vorn begonnen werden. Das neue Terminierungskriterium ergibt sich aus der Menge des freien Speichers. Verbleibt nicht genügend Speicher, um die beiden entstehenden Teillisten und jeweils einen Knoten zu speichern, wird am Anfang des Speicherblocks ein Blattknoten erzeugt auf den die Eingabeliste von Objekten folgt. Diese muss zu diesem Zweck vor der Sortierung gesichert werden. Für Verfahren bei denen die Objektliste partitioniert wird entfällt der Teil für die Behandlung von mehrfach referenzierten Objekten.

Die Auslegung der Knoten, die sich durch diese Art der Konstruktion ergibt, ist damit ähnlich wie in Abbildung 5.3 c). Zusätzlich folgt die Liste der Objekte in einem Blattkno-

acceleration
KdTreeBase
subdivide

ten direkt auf diesen Blattknoten. Dies begünstigt den Zugriff über einen Cache. Auch hierfür wird bei der klassischen Top-down Konstruktion, zumindest für kd-trees, meistens dynamisch Speicher angefordert. Das *lazy building* lässt sich intuitiv mit dem beschriebenen Vorgehen verbinden. Der einzigen Nachteil ist, dass Teile des einmal reservierten Speichers unter Umständen nicht genutzt werden.

Einen weiteren Vorteil dieser Art den Baum zu konstruktion, den [Wächter u. Keller \(2007\)](#) aufzeigen, ist die Möglichkeit verschiedene Baumtypen zu mischen. So lassen sich durch das Hinzufügen eines weiteren Flags beispielsweise kd-Tree Knoten mit BIH Knoten mischen. Dies ermöglicht die effizientere Behandlung von leerem Raum in einer BIH. Wird bei der Konstruktion festgestellt, dass ein Teilbaum keine Objekte enthält kann anstatt einer Neupositionierung der Trennebene wie in [3.7](#) vorgeschlagen, ein kd-Tree Knoten angelegt werden. Dieser würde den leeren Raum effizienter behandeln als die reine Verwendung der BIH.

5.4 Parallelisierung

Parallelisierung lässt sich mit aktuellen Arbeitsplatzrechnern auf zwei unterschiedlichen Ebenen sinnvoll realisieren: zum Einen auf Befehlsebene, zum Anderen auf Threadebene. Die beiden Ansätze stehen orthogonal zueinander und können somit beide gleichzeitig eingesetzt werden, um die Gesamtleistung zu erhöhen.

5.4.1 SIMD

SIMD steht für *Single Instruction Multiple Data*. Ursprünglich nur Großrechnern vorbehalten besitzt inzwischen nahezu jeder Prozessor eines Arbeitsplatzrechners diese Technologie. Es handelt sich dabei um die Fähigkeit, die gleiche Operation auf Vektoren von Daten parallel auszuführen. Die Befehle sind in eigenen Befehlssätzen beschrieben und arbeiten in aktuellen Architekturen auf anderen Registern als die normalen Befehle. Aktuelle Arbeitsplatzrechner unterstützen die Bearbeitung von 128Bit Registern. Der Befehlssatz, der hierfür hauptsächlich genutzt wird, ist der SSE(2)-Befehlssatz, da dieser auf vielen Rechnern zur Verfügung steht. Auf den momentan verbreitetsten Prozessoren von AMD und Intel stehen acht 128Bit Register dafür zu Verfügung, auf den aktuelleren Modellen bereits 16 [AMD \(2006\)](#), [Int \(2008\)](#).

Die für das Raytracing interessante Interpretation der 128Bit stellen vier Gleitkommazahlen einfacher Genauigkeit dar. Der SSE Befehlssatz bietet arithmetische, logische und bitweise Operatoren die unter anderem auf Vektoren von vier Gleitkommazahlen

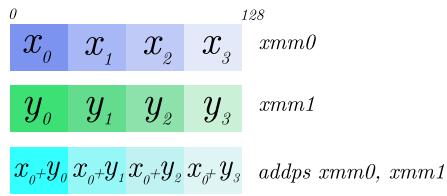


Abbildung 5.5: Mit Intels SSE Befehlssatz können kann mit einer einzigen Anweisung eine Operation auf vier unabhängigen Operanden(-paaren) ausgeführt werden.

Quelltext 5.4: Die Anordnung der Daten im SStructure of arraysLayout ermöglicht eine effizientere Nutzung des SSE Befehlssatzes

```
// Array of structures           // Structure of arrays
typedef struct {               typedef struct {
    float x,y,z;             float x[NumOfVertices];
} Vector3;                   float y[NumOfVertices];
                             float z[NumOfVertices];
Vector3 vertices [NumOfVertices] } VertexList;
                                 VertexList vertices
```

angewendet werden können. Ein typisches Beispiel ist die Addition von vier Paaren von Gleitkommazahlen (Abbildung 5.5).

Für die Verwendung der SSE Befehle ergeben sich zwei Ansätze. Der erste Ansatz versucht, eine Operation auf einem Operandenpaar eines komplexen Datentyps, durch die Verwendung der SSE Befehle, zu beschleunigen. Die Organisation der Daten erfolgt in Feldern von strukturierten Datentypen - so genannten Arrays of structures(AoS). Da beim Raytracing besonders viele Probleme der räumlichen Geometrie gelöst werden müssen, bietet sich der dreidimensionale Vektor als Kandidat für eine effizientere Verarbeitung mit SSE an. Die Operationen für 3D-Vektoren lassen sich leicht kapseln. Eine effizientere Implementierung dieser Operationen durch SSE-Befehle hätte den Vorteil, dass diese für Rest des Programms transparent wäre. Ein Beispiel für eine solche Operation ist die bereits vorgestellte Addition (Abbildung 5.5). Ein Nachteil dieses Ansatzes ist, dass sich die vektorisierte Verarbeitung nur dort effizient einsetzen lässt, wo die Operationen unabhängig auf den Komponenten des Vektors arbeiten. Für Operationen wie das Kreuzprodukt bietet diese Art der Implementierung mit SSE nicht den optimalen Durchsatz, da die Komponenten nicht voneinander unabhängig verarbeitet werden können. Einen weiteren Nachteil stellt die Tatsache dar, dass für *dreidimensionale* Vektoren nur drei Viertel des 128Bit Registers genutzt werden, was eine weitere Verschlechterung gegenüber der idealen Auslastung darstellt.

Die Alternative ist die Anordnung der Daten in einer so genannten *Structure of Arrays* (SoA). Für die Speicherung von geometrischen Vektoren im SoA Format werden die Komponenten verschiedener Vektoren in einem gemeinsamen Feld abgelegt. Quelltext 5.4 zeigt die unterschiedlichen Auslegungen im direkten Vergleich. Durch die SoA Auslegung wird es möglich, statt einzelne Schritte einer Operation, die gesamte Operation parallel auszuführen. Für das Vektorbeispiel heißt das: Anstatt die Operationen auf den Komponenten zu parallelisieren, wird die gesamte Operation für vier Vektoren, beziehungsweise Vektorpaare, ausgeführt. Dies behebt die beiden Schwächen des AoS Formats. Die Anzahl der parallelisierbaren Operationen hängt nun nicht mehr von der Anzahl der Komponenten des Vektors ab. Die Operationen für eine Komponente werden jetzt für vier Vektoren gleichzeitig ausgeführt. Damit werden auch *horizontale*, also Operation die auf verschiedenen Komponenten zweier Vektoren arbeiten, möglich. Aus diesen Gründen ermöglicht die SoA-Auslegung eine effizientere Nutzung der SIMD Kapazitäten eines aktuellen Prozessors als die AoS-Auslegung (Int (2007)).

Parallele Traversierung

Wald (2004) beschreibt in seiner Arbeit die effiziente Verfolgung von vier kohärenten Strahlen. Mit Hilfe des SSE Befehlssatzes werden vier Strahlen gleichzeitig durch einen kd-Tree verfolgt. Die Traversierungsroutine muss hierfür entsprechend angepasst werden. Dabei wird wie bereits in Abschnitt 3.4.2 vorgegangen: Jeder Teilbaum, der von mindestens einem Strahl geschnitten wird, wird weiter untersucht. Strahlen die lediglich einen Teilbaum besuchen, weil ein Nachbarstrahl dessen Voxel schneidet, werden maskiert um zu verhindern, dass diese Strahlen die Traversierung weiterer Teilbäume auslösen. Benthin (2006) beschreibt eine erweiterte Implementierung die den SSE Befehlssatz verwendet um größere, kohärente Strahlpakete zu verfolgen. Dabei setzt er ein von Reshetov u. a. (2005) erstmals beschriebenes Verfahren ein, um Teilbäume für das gesamte Paket von der Traversierung auszuschließen, ohne dass die konkret im Paket enthaltenen Strahlen bekannt sein müssen. Wie auch in Kapitel 3.4 beschrieben, wird stellvertretend ein Pyramidenstumpf betrachtet. Durch Intervallarithmetik ist es möglich, über die Extremwerte des Schnitts des Pyramidenstumpfes mit der Trennebene, darauf zu schließen, welcher Teilbaum traversiert werden muss. Die Grundidee der Intervallarithmetik ist es, die gängigen Operationen, die in der klassischen Arithmetik auf individuellen Werten ausgeführt werden, auf Intervalle auszudehnen. Ein gute und knappe Einführung wird von Boulos u. a. (2006) gegeben.

Für Strahlpakete, die sich aus Kacheln der Projektionsebene ergeben, können wieder

Quelltext 5.5: Definition einer Datenstruktur für Strahlpakete im SoA-Format

```

typedef struct {
    float origin_x[4];
    float origin_y[4];
    float origin_z[4];

    float direction_x[4];
    float direction_y[4];
    float direction_z[4];
} RayQuadruple;

```

die vier “Eck-Strahlen” als Repräsentanten des Pyramidenstumpfes verwendet werden. Wie in Abbildung 5.6 dargestellt, lassen sich die Minima, beziehungsweise Maxima der Schnittpunkte mit der Trennebene verwenden, um zu bestimmen welcher Teilbaum traversiert werden muss. Die Bestimmung der vier Schnittpunkte ist ein idealer Kandidat für die parallele Implementierung mit dem SSE Befehlssatz. Für eine effiziente Lösung müssen (mindestens) die vier Eckstrahlen im SoA-Format gespeichert werden. Eine entsprechende Auslegung eines Strahlpakets könnte wie in Quelltext 5.5 aussehen.

Auch der Schnitttest von Strahlpaketen und Objekten lässt sich durch die Verwendung der SSE Befehle effizienter gestalten. In 3.4 wurde bereits das Verfahren von Dmitriev u. a. (2004) angesprochen. Es verwendet die Eck-Strahlen eines Pakets, um einen konservativen, schnellen Test zu implementieren mit dem sich feststellen lässt, ob das Dreieck überhaupt von dem Paket geschnitten werden kann. Da die Tests auch hier für vier Strahlen unabhängig voneinander ausgeführt werden müssen, ist eine parallele Implementierung mittels SIMD nur konsequent.

Kann nicht ausgeschlossen werden, dass ein Strahl das Dreieck schneidet, müssen alle Strahlen des Pakets auf Schnitt getestet werden. Für größere Pakete lohnt es sich die enthaltenen Strahlen wiederum in 2x2 Paketen und im SoA-Format anzurufen. Diese können dann mit dem gleichen Algorithmus, parallel getestet werden.

Parallele Konstruktion

Ein Schritt der bei fast allen Konstruktionsalgorithmen häufig ausgeführt wird, ist die Bestimmung auf welcher Seite einer Ebene ein Objekt liegt, beziehungsweise ob es die Ebene schneidet. Hier gibt es zwei Möglichkeiten diese Operation durch SIMD effizienter zu gestalten.

Ein Objekt vier Ebenen Müssen mehrere Kandidaten von Ebenen verglichen werden

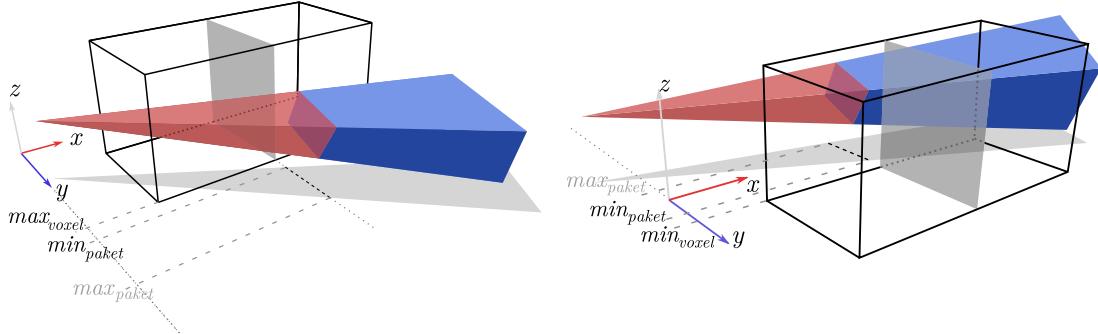


Abbildung 5.6: Unter der Annahme, dass der aktuelle Voxel von dem Frustum geschnitten wird, gilt: Liegt das Minimum des Intervalls hinter dem Maximum der Trennebene wird lediglich der vordere Voxel geschnitten. Liegt das Maximum des Frustums vor dem Minimum der Trennebene wird ausschließlich der hintere Voxel geschnitten. Bei negativer Ausbreitungsrichtung sind die beiden Fälle genau vertauscht. Tritt keiner der beiden Fälle ein müssen beide Teilbäume traversiert werden. Der Test muss immer für die beiden Achsen ausgeführt werden die parallel zur Trennebene liegen

kann der Test eines Objekts durch die Verwendung von SIMD mit mehreren Ebenen gleichzeitig durchgeführt werden.

Eine Ebene mit vier Objekten Alternativ lässt sich auch eine Ebene parallel mit vier Objekten testen. Die Objekte, beziehungsweise die AABBs, müssen für eine effiziente Umsetzung im *SOA* Format gespeichert werden.

Welches Verfahren die höhere Leistung liefert hängt von dem eingesetzten Konstruktionsalgorithmus ab. Neben der Tatsache, dass vier Rechenoperationen gleichzeitig ausgeführt werden, wird besonders für den ersten Ansatz auch eine bessere Nutzung des Caches erreicht. Für den parallelen Test mit vier Ebenen muss ein Objekt lediglich einmal aus dem Hauptspeicher geladen werden.

Paralleles Shading

Eine allgemeine Parallelisierung der Shadingberechnungen ist nicht möglich. Da Programmcode für die Nutzung von SIMD besondere Befehle (zum Beispiel SSE) verwenden muss, ist es unabdingbar, jeden Shader einzeln für die parallele Verarbeitung anzupassen. Dies ist besonders anspruchsvoll für Anwendungen in denen die Shader durch den Benutzer zur Laufzeit gestaltet werden können.

Der gängige Ansatz besteht daraus, vier Schnittpunkte eines Strahls mit der Geometrie gleichzeitig zu *shaden*. Der Shader bezeichnet den Algorithmus, der bestimmt, wieviel von

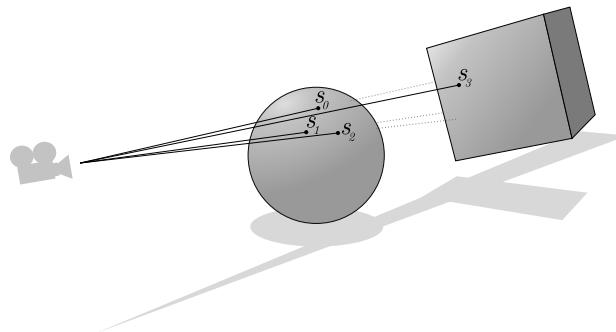


Abbildung 5.7: Der Schnittpunkt S_3 liegt auf einem anderen Objekt als die anderen drei Schnittpunkte. Parallelisierung mit SIMD kann nur angewendet werden, wenn die Kugel und der Würfel den gleichen Shader verwenden.

dem eintreffenden Licht an einem Punkt im Raum in Richtung der Kamera reflektiert wird. Für die Unterstützung mit SIMD wird er so angepasst, dass er vier Punkte gleichzeitig verarbeiten kann. Welcher Shader ausgeführt wird hängt von dem Material des Objekts ab, das der Primärstrahl geschnitten hat. Die Schnittpunkte, die als Ergebnis einer Traversierung mit SIMD Befehlen entstehen, liegen meist bereits im *SoA*-Format vor (welches nun auch für den Shadingalgorithmus benötigt wird). Da ein Shader jedoch lediglich für *ein* Material verwendet werden kann, ist der parallele Shadingalgorithmus lediglich dann anwendbar, wenn alle Schnittpunkte den gleichen Shader verwenden. Insbesondere für kohärente Strahlpakete ist die Wahrscheinlichkeit jedoch hoch, dass dies der Fall ist. Für Pakete, die Schnittpunkte mit Objekten unterschiedlicher Materialien enthalten, muss entweder die nicht parallele Version des Algorithmus verwendet werden oder für die unterschiedlichen Shader werden solange Schnittpunkte “gesammelt” bis genügend für die Ausführung der parallelen Version zur Verfügung stehen. Der letztere Ansatz erfordert jedoch zusätzlich die Verfolgung, welcher Schnittpunkt zu welchem Bereich auf der Projektionsfläche gehört.

5.4.2 Multithreading

Um die Rechenleistung eines Rechners zu steigern wird in aktuellen Prozessoren die Anzahl der enthaltenen Rechenkerne erhöht und nicht mehr die Pipeline verlängert. Die verschiedenen Kerne bekommen dabei teilweise eigene Caches und teilen sich die höheren Caches. Abbildung 5.8 zeigt eine typische Anordnung. Um den Zustand des Speichers, der in den verschiedenen Caches der Kerne repräsentiert wird, konsistent zu halten, wird ein vierstufiges Protokoll genutzt. Dieses Protokoll bewirkt unter anderem, dass wenn ein

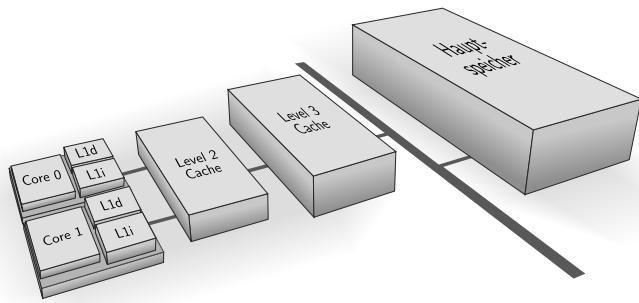


Abbildung 5.8: Typische Cachestruktur aktueller Prozessoren. Die Abbildung zeigt einen Prozessor mit zwei Kernen. Jeder Kern verfügt über einen eigenen Level 1 Befehls-(L1i) und Datencache(L1d). Die Caches der höheren Level werden von beiden Kernen gemeinsam genutzt.

Kern eine Cacheline verändert, die Cachelines der anderen Kerne, welche den gleichen Speicherbereich abbilden, als ungültig markiert werden. Dadurch werden sie automatisch beim nächsten Zugriff neu geladen. Um den Synchronisationsaufwand, welcher durch die Verwendung von mehreren Kernen entsteht, so gering wie möglich zu halten ist darauf zu achten, dass unterschiedliche Threads nicht auf den selben Speicherbereich zugreifen, wenn einer der Zugriffe schreibend ist. Auch hierfür ist die Trennung von Daten, die ausschließlich gelesen werden, von solchen, die wiederholt geschrieben werden, wichtig (siehe auch Abschnitt 5.2.3). Durch die Verwendung des Schlüsselwortes `const` ist der Compiler in der Lage, entsprechende Variablen zu erkennen und von den anderen Variablen getrennt zu speichern ([Drepper \(2007\)](#)).

Parallele Traversierung

Die Parallelisierung der Traversierung durch Threads ist im Vergleich zu SIMD verhältnismäßig einfach. Jedem Thread werden Bereiche der Projektionsfläche zugewiesen, die dieser zu rendern hat. Diese sollten nicht zu wenig Strahlen enthalten, da sonst der Aufwand die Threads zu administrieren größer wird als der Leistungsgewinn. Sind die Kacheln hingegen zu groß kann dies dazu führen, dass ein Thread seinen Bereich viel schneller abgearbeitet hat als andere. Sind keine ungerenderten Kacheln mehr vorhanden muss der Thread warten bis die anderen Threads ihre Arbeit beendet haben.

Für den Traversierungsvorgang an sich ist es günstig, wenn die Bereiche der verschiedenen Threads nah beieinander liegen, da die Threads dann ähnliche Knoten der Beschleunigungsstruktur besuchen, und diese in den gemeinsamen Cacheebenen eher Cache-hits erzeugen. Andererseits ist es dann auch wahrscheinlicher, dass sich einige der Cachelines

des Framebuffers in die geschrieben wird überlappen. Um dies zu verhindern, bieten sich folgende drei Möglichkeiten an:

1. Die Kacheln verschiedener Threads in getrennten Regionen starten
2. Die Framebufferinhalte in getrennte Speicherbereiche schreiben und später zusammenfügen
3. Bei der Kacheleinteilung die Cachelinebreite beachten, sodass eine Kachelkante immer an einer Cacheline beginnt beziehungsweise endet.

Parallele Konstruktion

Der einfachste Weg mehrere Threads bei der Konstruktion einer hierarchischen Bechlebungsdatenstruktur einzusetzen, ist jedem Thread die Konstruktion einzelner Teilbäume zu überlassen. Voraussetzung hierfür ist, dass genügend unabhängige Teilbäume zur Verfügung stehen. Dies ist besonders für den Wurzelknoten nicht gegeben. Eine effiziente Nutzung aller Kerne kann also erst geschehen, wenn die obersten Ebenen der Hierarchie konstruiert wurden.

Alternativ kann zunächst eine grobe Vorsortierung in so genannte “*Bins*” vorgenommen werden. Hierfür wird der Raum der Szene entlang einer Achse in diskrete Raumelemente aufgeteilt und jedes Objekt, je nach Überlappung, einem oder mehreren dieser Voxel zugeordnet. BVH und BIH profitieren dabei davon, dass jedes Objekt lediglich einmal referenziert wird. Für die Zuordnung wird die Liste der Objekte in genau so viele Partitionen aufgeteilt, wie es Threads gibt. Danach untersucht jeder Thread die Objekte einer Partition und weist sie dem entsprechenden *Bin* zu. Um zu verhindern, dass es zu Schreibkonflikten, beziehungsweise hohem Synchronisationsaufwand zwischen den Threads kommt, schreibt jeder Thread in einen eigenen Satz von *Bins*. Nachdem alle Threads die Klassifizierung der Objekte beendet haben, werden die *Bin*-Sätze der verschiedenen Threads zusammengeführt. ([Wald, 2007](#))

Eine weitere Alternative, die Konstruktion von BVHs mit Threads zu parallelisieren, welche ebenfalls von [Wald \(2007\)](#) vorgeschlagen wird, wählt die *Bins* in Form eines dreidimensionalen, gleichmäßigen Gitters. Die Gitterzellen lassen sich durch Berechnung der AABBs der enthaltenen Objekte und paarweiser Kombination mit den Nachbarzellen zu einer gültigen BVH zusammenführen. Die Zellen, die nun noch mehrere Objekte enthalten, können wiederum parallel weiter durch unterschiedliche Threads unterteilt werden. Alternativ können sie erst in der Traversierungsphase bei Bedarf konstruiert werden (vgl. Abschnitt “Schnelle Konstruktion” [4.2.3](#)).

6 Praxis

Im ersten Abschnitt dieses Kapitels wird kurz auf die Erfahrungen eingegangen, die während der Analyse und Implementierung einiger der beschriebenen Verfahren gemacht wurden. Darauf folgt eine knappe Einführung in die Struktur der erstellten Software.

6.1 Implementierung

Die im Zusammenhang mit dieser Arbeit erstellte Anwendung unterstützt als Objekte der Szene lediglich Dreiecke. Damit folgt sie den Implementierungen vieler aktueller Realtime Raytracer ([Wald \(2004\)](#), [Wächter \(2008\)](#)). Prinzipiell ermöglicht der allgemeine Raytracing Algorithmus die Verwendung beliebiger Objekte, für die ein Strahlschnittest zur Verfügung steht. Die verschiedenen Strahlschnitttests müssten jedoch mit in den Traversierungsalgorithmus aufgenommen werden. Die entsprechende Vergößerung dieses Codeabschnitts, in Verbindung mit den datenabhängigen Sprüngen für die Schnitttests, würden eine signifikante Verschlechterung der Gesamtleistung bewirken.

Für die erste Implementierung konnten interaktive Frameraten lediglich für mittlere bis kleine Szenen erreicht werden (<50k Dreiecke). Daraufhin wurde von einer weiteren Optimierung des Codes abgesehen, da dies die Lesbarkeit weiter herabgesetzt hätte und die Leistungsfähigkeit der vorgestellten Algorithmen bereits von den Autoren demonstriert wurde. Anstatt dessen wurde die Architektur zugunsten einer leicht verständlichen Struktur umgestaltet. Für die Anwendung können viele Parameter, wie zum Beispiel die verwendete Beschleunigungsdatenstruktur, zur Laufzeit konfiguriert werden. Die implementierte Struktur ermöglicht die einfache Implementierung weiterer Verfahren.

Einige der vorgestellten Verfahren sind sowohl in einer leicht verständlichen (zum Beispiel rekursiven) als auch in einer effizienteren Variante (kompakte Speicherauslegung, iterative Traversierung) implementiert. Um die Lesbarkeit beizubehalten wurden einige Kompromisse eingegangen. So wird zum Beispiel auch in häufig ausgeführtem Code, nicht gänzlich auf die Verwendung virtueller Methoden verzichtet.

Die im Zusammenhang mit der Arbeit erstellte Codebasis soll nicht als Leistungsdemonstration der Verfahren dienen, sondern vielmehr Einblick in die verschiedenen

Verfahren zur Raumaufteilung geben. Zusätzlich soll durch die Möglichkeit, naive mit optimierten Implementierungen direkt zu vergleichen, veranschaulicht werden, welche Änderungen für die Verwendung bestimmter Techniken und Technologien am Code vorgenommen werden müssen.

6.2 Programmstruktur

Die Quelldateien der Software sind in folgende Unterverzeichnisse gegliedert:

acceleration Das Verzeichnis enthält alle Klassen, die Beschleunigungsdatenstrukturen umsetzen oder ausschließlich von diesen genutzt werden.

core Geometrische Primitive sowie weitere Klassen, die für die Konstruktion eines Raytracers nötig sind, werden hier abgelegt.

renderer Unabhängig von den Beschleunigungsstrukturen lassen sich verschiedene Strategien für die finale Bildsynthese wählen (einzelne Strahlen, Pakete etc.). Die unterschiedlichen Strategien werden in *Renderern* gekapselt, die in diesem Verzeichnis zu finden sind.

shader Im *shader*-Verzeichnis befinden sich die Klassen, die das unterschiedliche Verhalten von Materialien beim Auftreffen von Licht implementieren.

Diagramm 6.1 zeigt das Zusammenspiel der wichtigsten Klassen. Die Klasse **AccellerationStructure** bildet die Schnittstelle für die verschiedenen Beschleunigungsdatenstrukturen. Die drei Methoden außer **construct()** dienen der Beantwortung der in Kapitel 2 vorgestellten drei Problemen. Neben den gezeigten enthält die Schnittstelle auch die Signaturen für die Beantwortung dieser Probleme für mehrere Strahlen gleichzeitig mit Unterstützung des SSE Befehlssatzes.

AccellerationStructure wird von folgenden Klassen implementiert

PrimitiveList Implementiert den in Abschnitt 2.5 beschriebenen naiven Ansatz ohne jegliche Beschleunigung

RegularGrid Ein gleichmäßiges Gitter welches den in Kapitel 3.1.1 beschriebenen Algorithmus zur Traversierung nutzt.

KdTreeSimple Zeigt eine unoptimierte Version des kd-Trees zu Verdeutlichung der grundsätzlichen Funktionsweise.

KdTreeBase Oberklasse für verschiedene kd-Tree Varianten, die alle die in Abschnitt 5.3.2 beschriebene Speicherauslegung verwenden.

KdTreeSAHNaive Implementiert den naiven Ansatz zur Konstruktion mit der SAH mit $O(n^2)$

KdTreeSahNlog2N Implementiert den in Abschnitt 4.2.3 vorgestellten Algorithmus zur Konstruktion mit $O(n \log^2 n)$.

KdTreeSahNlogN Konstruiert den kd-tree mit $O(n \log n)$ wie in Abschnitt 4.2.3 beschrieben.

Bih Nicht optimierte Version der BIH, zur Veranschaulichung der allgemeinen Vorgehensweise

BihCompact Verwendet die kompakte Speicherauslegung für BIH-Knoten

BihIterative Verwendet die kompakte Speicherauslegung und traversiert die Hierarchie iterativ

Die Shader dienen dazu, für einen Punkt der Szenengeometrie zu ermitteln, wieviel Licht von dort in eine bestimmte Richtung reflektiert wird. Die Klasse **Shader** definiert hierzu zwei Signaturen, eine für einzelne Punkte und eine für mehrere Punkte im SIMD Format. Zur Zeit sind drei einfache Shader implementiert:

FlatShader gibt immer einen konstanten Farbwert zurück

DirectShader berechnet den reflektierten Anteil des Lichts der direkt von den Lichtquellen abgegeben wird

SpecularShader berechnet die Anteile, die durch perfekte Reflektion und Transmission in die gegebene Richtung abgegeben werden.

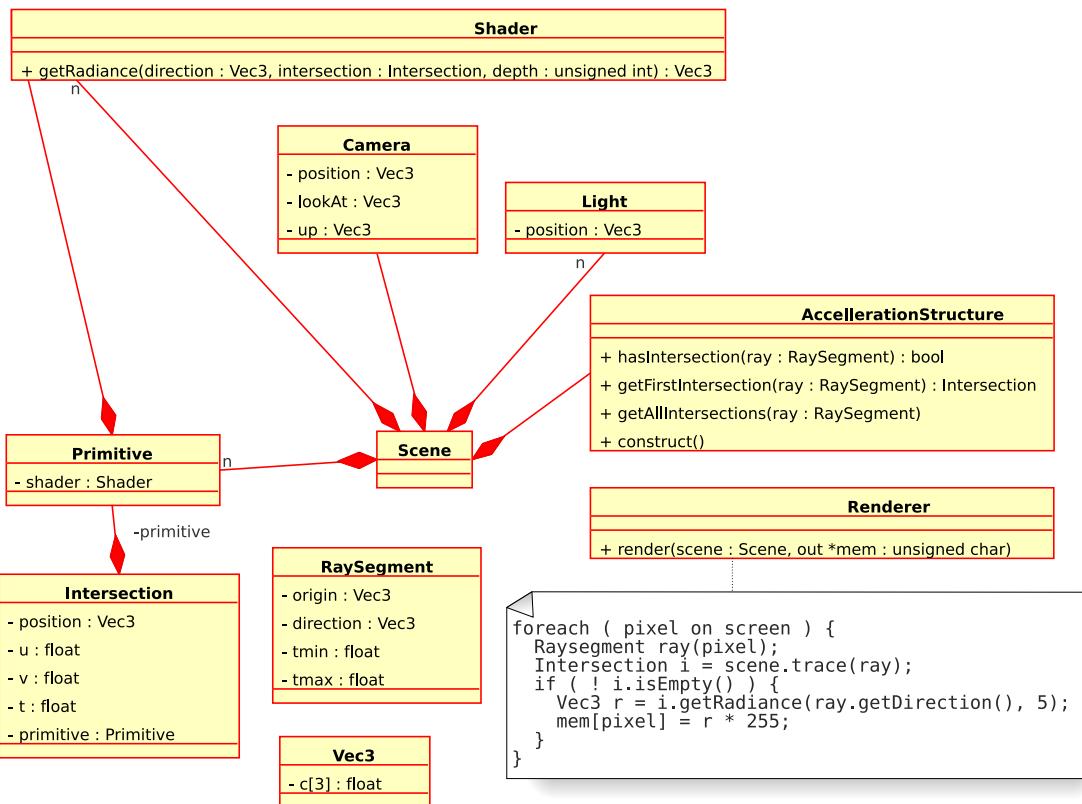


Abbildung 6.1: Das Diagramm zeigt nicht alle Details der Implementierung. Der Codeschnipsel rechts unten zeigt den prinzipiellen Ablauf des Renderings eines Bilds. Der entsprechende C++-Code zum gezeigten Pseudocode finde sich in der Klasse SingleRayRenderer wieder.

7 Fazit

Raytracing allgemein ist ein Forschungsgebiet auf dem seit nun fast 30 Jahren geforscht wird. Einsteigern in diesem Gebiet steht deswegen eine umfangreiche, aber nicht immer übersichtliche Menge an Literatur zur Verfügung. Durch die ständig steigende Leistungsfähigkeit der Computer ist das Interesse am Raytracingverfahren, besonders in der Kombination mit echtzeitfähigen Algorithmen, in den letzten Jahren stark angestiegen. Dementsprechend ist auch die Anzahl und Frequenz der Veröffentlichungen zu diesem Thema angestiegen. Die unterschiedlichen Arbeiten sind besonders vor dem Hintergrund dynamischer Szenen oft nur schwer zu bewerten, beziehungsweise zu vergleichen. Hingegen sind sich alle Autoren einig, dass es nicht **das** beste Verfahren gibt. Zuviele Einflussfaktoren bestimmen die letztendliche Leistung eines Raytracers. Zusätzlich gilt, dass eine für einen bestimmten Anwendungsfall optimierter Lösung immer eine allgemeinere Lösung übertreffen wird. In dieser Arbeit wurden die drei aktuell konkurrierenden Verfahren zur Raumaufteilung mit ihren Vor- und Nachteilen vorgestellt (Gitter, hierarchische Raumauflösung, Partitionierung der Objektmenge). Bei der Auswahl eines Verfahrens für einen konkreten Anwendungsfall sollten die Vor- und Nachteile, die sich für die verschiedenen Verfahren aus der Parametern der Anwendung ergeben, genau untersucht werden. Je nach Szenengröße, Art der Dynamik, Verteilung der Objekte usw. kann sich das eine oder das andere Verfahren besser für einen bestimmten Fall eignen.

Die vorgestellten Richtlinien für die Anwendungsentwicklung auf aktueller Hardware gelten allgemein für die Entwicklung leistungsorientierter Anwendungen. Dabei sei noch einmal darauf hingewiesen, dass die Optimierung für beste Cachebenutzung mit Abstand die wichtigste Optimierung darstellt. Für eine ausführliche Erläuterung der Optimierungsmöglichkeiten bei der Anwendungsentwicklung sei auf [Drepper \(2007\)](#) und [Fog \(2008a\)](#) verwiesen.

Unter Umständen wird die Hardwarearchitektur der Rechner in Zukunft soweit optimiert werden, dass Raytracing in Echtzeit auch ohne spezielle Berücksichtigung der Hardware möglich wird. Fakt ist aber, dass auch aktuelle Implementierungen immer noch viele grobe Annäherungen bei der Simulation des Lichtflusses machen. Eine optimierte Anwendung kann die gewonnenen Kapazitäten in die Simulation bisher nur mäßig

darstellbarer Phänomene investieren. So stellen zum Beispiel Global Illumination und Distributed Raytracing bisher noch genug Probleme, welche momentan nicht in Echtzeit gelöst werden können, dass auch die Prozessoren kommender Rechnergenerationen, allein mit der Darstellung von 3d-Szenen voll ausgelastet werden können..

Diese Entwicklung deutet zwar darauf hin, dass in Zukunft immer ansprechendere Grafiken, auch in Echtzeit, generiert werden können. Sie zeigt aber auch, dass immer der Bedarf für eine auf aktuelle Hardware optimierte Lösung bestehen wird.

Anhang

7.1 Baryzentrische Koordinaten

Bei den baryzentrischen Koordinaten handelt es sich um eine Darstellungsform eines Punktes. In dieser Form lässt sich einfach feststellen, ob ein Punkt innerhalb eines Dreiecks liegt. Dies lässt sich für das Raytracing beim Strahl-Objekt-Schnittest für Dreiecke ausnutzen. Zunächst muss berechnet werden, ob der Schnittpunkt des Strahls mit der Ebene des Dreiecks innerhalb des interessanten Segments, also zwischen t_{min} und t_{max} liegt. Für den Test, ob der Schnittpunkt innerhalb des Dreiecks liegt, wird zunächst folgende Annahme getroffen: Durch die gestrichelten Linien in Abbildung 7.1 entstehen drei Teildreiecke. Liegt der Schnittpunkt S innerhalb des Dreiecks, muss die Summe der Flächen der Dreiecke A_0, A_1, A_2 der Fläche A_t des gesamten Dreiecks entsprechen:

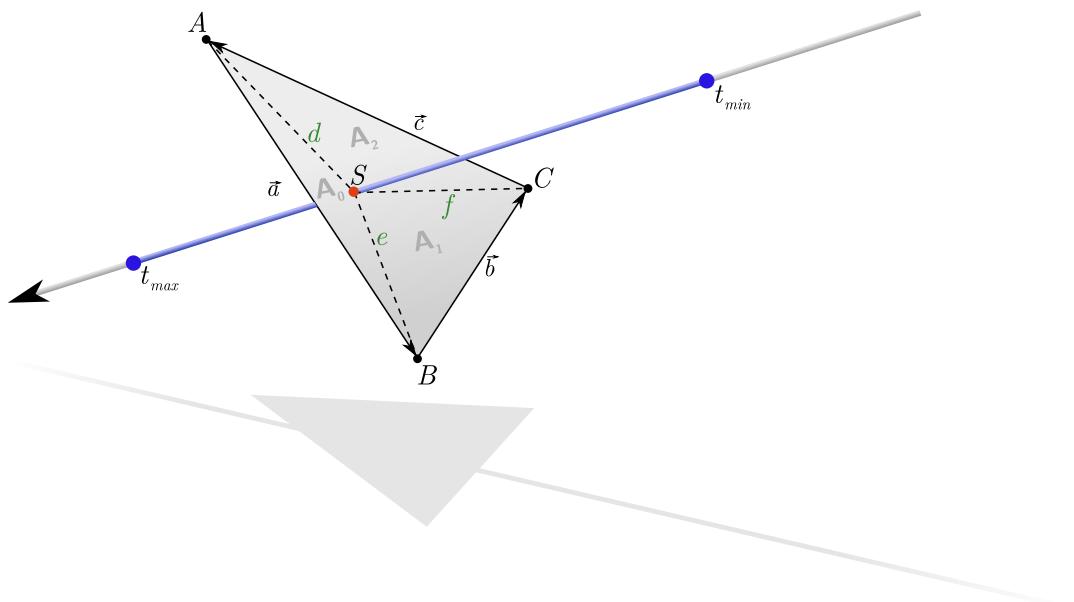


Abbildung 7.1: Strahl-Dreieck Schnittest mit baryzentrischen Koordinaten

$$A_t = A_0 + A_1 + A_2$$

Jede Fläche der Teildreiecke lässt sich als Bruchteil der Gesamtfläche darstellen:

$$A_0 = u * A_t, A_1 = v * A_t, A_2 = w * A_t$$

So dass gilt:

$$0 \leq u, v, w \leq 1 \text{ und } u + v + w = 1$$

Aus den Kanten des Dreiecks und den gestrichelt eingezeichneten Verbindungslien zwischen den Eckpunkten und dem Schnittpunkt lassen sich u, v, w wie folgt bestimmen:

$$u = \frac{u * A_t}{A_t} \frac{A_0}{A_t} = \frac{\frac{1}{2} \|\vec{d} \times \vec{a}\|}{\frac{1}{2} \|\vec{a} \times \vec{b}\|}$$

$$v = \frac{v * A_t}{A_t} \frac{A_1}{A_t} = \frac{\frac{1}{2} \|\vec{e} \times \vec{b}\|}{\frac{1}{2} \|\vec{a} \times \vec{b}\|}$$

$$w = \frac{w * A_t}{A_t} \frac{A_2}{A_t} = \frac{\frac{1}{2} \|\vec{f} \times \vec{c}\|}{\frac{1}{2} \|\vec{a} \times \vec{b}\|}$$

Durch den Test ob die Summe aus u, v und w für den gegebene Fall eins ergibt, lässt sich einfach bestimmen, ob der Schnittpunkt im Dreieck liegt oder nicht.

Von $w = \frac{\frac{1}{2} \|\vec{f} \times \vec{c}\|}{\frac{1}{2} \|\vec{a} \times \vec{b}\|}$ sind die roten Komponenten konstant. Die Berechnung lässt sich dadurch wie folgt umstellen:

$$w = \frac{\frac{1}{2} \|\vec{f} \times \vec{c}\|}{\frac{1}{2} \|\vec{a} \times \vec{b}\|} = \frac{1}{2} \parallel \vec{f} \times \vec{p}r\vec{e}_w \parallel$$

$$\vec{p}r\vec{e}_w = \frac{\vec{c}}{2 * \|\vec{a} \times \vec{b}\|}$$

Analog lassen sich die Formeln für u und v umstellen. $\vec{p}r\vec{e}_u, \vec{p}r\vec{e}_v, \vec{p}r\vec{e}_w$ sind für jedes Dreieck konstant und kann vorberechnet werden. Dadurch werden bei mehreren Schnitttests mit dem gleichen Dreieck die entsprechenden Berechnungen gespart. Wächter (2008)



Literaturverzeichnis

- [AMD 2006] Advanced Micro Devices, Inc: *AMD Athlon™ 64 Processor Product Data Sheet*. September 2006
- [Akenine-Möller 2001] AKENINE-MÖLLER, Tomas: Fast 3D Triangle-Box Overlap Testing. In: *journal of graphics tools* 6 (2001), Nr. 1, 29–33. <http://jgt.akpeters.com/papers/AkenineMoller01/>
- [Amanatides u. Woo 1987] AMANATIDES, John ; WOO, Andrew: A Fast Voxel Traversal Algorithm for Ray Tracing. Version: 1987. citesear.ist.psu.edu/amanatides87fast.html. In: *Eurographics '87*. Amsterdam, North-Holland : Elsevier Science Publishers, 1987, 3–10
- [Appel 1968] APPEL, Arthur: Some Techniques for Shading Machine Renderings of Solids. In: *AFIPS 1968 Spring Joint Computer Conf.* Bd. 32, 1968, S. 37–45
- [Benthin 2006] BENTHIN, Carsten: *Realtime Ray Tracing on current CPU Architectures*, Saarland University, Diss., 2006
- [Boulos u. a. 2006] BOULOS, Solomon ; WALD, Ingo ; SHIRLEY, Peter: Geometric and Arithmetic Culling Methods for Entire Ray Packets / School of Computing, University of Utah. 2006 (UUCS-06-010). – Forschungsbericht
- [Cook u. a. 1984] COOK, Robert L. ; PORTER, Thomas ; CARPENTER, Loren: Distributed ray tracing. In: *SIGGRAPH Comput. Graph.* 18 (1984), Nr. 3, S. 137–145. <http://dx.doi.org/http://doi.acm.org/10.1145/964965.808590>. – DOI <http://doi.acm.org/10.1145/964965.808590>. – ISSN 0097–8930
- [Djeu u. a. 2007] DJEU, Peter ; HUNT, Warren ; WANG, Rui ; ELHASSAN, Ikrima ; STOLL, Gordon ; MARK, William R.: Razor: An Architecture for Dynamic Multiresolution Ray Tracing / The University of Texas at Austin, Department of Computer Sciences. 2007 (TR-07-52). – Forschungsbericht
- [Dmitriev u. a. 2004] DMITRIEV, Kirill ; HAVRAN, Vlastimil ; SEIDEL, Hans-Peter: Faster Ray Tracing with SIMD Shaft Culling / Max-Planck-Institut für Informatik. Saarbrücken, Germany, December 2004 (MPI-I-2004-4-006). – Research Report. – 13 S. – ISBN 0946–011X
- [Drepper 2007] DREPPER, Ulrich: What Every Programmer Should Know About Memory / Red Hat, Inc. 2007. – Forschungsbericht

- [Fog 2008a] FOG, Agner: Optimizing software in C++ / Copenhagen University College of Engineering. 2008. – Forschungsbericht
- [Fog 2008b] FOG, Agner: Optimizing subroutines in assembly / Copenhagen University College of Engineering. Version: January 2008. <http://agner.org/optimize/>. 2008. – Forschungsbericht
- [Fuchs u. a. 1980] FUCHS, Henry ; KEDEM, Zvi M. ; NAYLOR, Bruce F.: On visible surface generation by a priori tree structures. In: *SIGGRAPH Comput. Graph.* 14 (1980), Nr. 3, S. 124–133. <http://dx.doi.org/http://doi.acm.org/10.1145/965105.807481>. – DOI <http://doi.acm.org/10.1145/965105.807481>. – ISSN 0097–8930
- [Glassner 1988] GLASSNER, Andrew: Space subdivision for fast ray tracing. (1988), S. 160–167. ISBN 0–8186–8854–4
- [Glassner 1989] GLASSNER, Andrew: *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989
- [Goldsmith u. Salmon 1987] GOLDSMITH, Jeffrey ; SALMON, John: Automatic Creation of Object Hierarchies for Ray Tracing. In: *IEEE Comput. Graph. Appl.* 7 (1987), Nr. 5, S. 14–20. <http://dx.doi.org/http://dx.doi.org/10.1109/MCG.1987.276983>. – DOI <http://dx.doi.org/10.1109/MCG.1987.276983>. – ISSN 0272–1716
- [Günther u. a. 2006] GÜNTHER, Johannes ; FRIEDRICH, Heiko ; WALD, Ingo ; SEIDEL, Hans-Peter ; SLUSALLEK, Philipp: Ray Tracing Animated Scenes using Motion Decomposition. In: *Computer Graphics Forum* 25 (2006), September, Nr. 3, 517–525. <http://www.mpi-inf.mpg.de/~guenther/modecomp/>. – (Proceedings of Eurographics)
- [Havran 2000] HAVRAN, Vlastimil: *Heuristic Ray Shooting Algorithms*, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, Ph.D. Thesis, November 2000. <http://www.cgg.cvut.cz/~havran/phdthesis.html>
- [Havran u. a. 2006] HAVRAN, Vlastimil ; HERZOG, Robert ; SEIDEL, Hans-Peter: On Fast Construction of Spatial Hierarchies for Ray Tracing / Max-Planck-Institut für Informatik. Version: June 2006. citeseer.ist.psu.edu/758008.html. Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, June 2006 (MPI-I-2006-4-004). – Research Report. – ISSN 0946–011X
- [Hunt u. a. 2006] HUNT, Warren ; MARK, William R. ; STOLL, Gordon: Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In: *2006 IEEE Symposium on Interactive Ray Tracing*, IEEE, September 2006
- [Int 2007] INTEL CORPORATION (Hrsg.): *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-016. Santa Clara, USA: Intel Corporation, November 2007. <http://developer.intel.com/design/processor/manuals/248966.pdf>

- [Int 2008] INTEL CORPORATION (Hrsg.): *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 253665-026US. : Intel Corporation, Februar 2008
- [Lantzman 2007] LANTZMAN, Eyal: *Iterative vs. Recursive approaches*. http://www.codeproject.com/KB/recipes/Iterative_vs_Recursive.aspx. Version: November 2007
- [Lauterbach u. a. 2006] LAUTERBACH, Christian ; YOON, Sung-Eui ; TUFT, David ; MANOCHA, Dinesh: RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In: *IEEE Symposium on Interactive Ray Tracing 2006*, 2006
- [Lexit u. a. 2000] LEXT, Jonas ; ASSARSSON, Ulf ; MOELLER, Thomas: *BART: A benchmark for animated ray tracing*. citeseer.ist.psu.edu/lext00bart.html. Version: 2000
- [Meyers 2006] MEYERS, Scott: *BookEffective C++*. 3. Addison-Wesley, 2006
- [NVIDIA 2003] NVIDIA: *Using Vertex Buffer Objects (VBOs)*. http://developer.nvidia.com/object/using_VBOs.html. Version: 2003
- [Pharr u. Humphreys 2004] PHARR, Matt ; HUMPHREYS, Greg: *Physically Based Rendering*. Morgan Kaufmann, 2004. – 203–214 S. <http://www.pbrt.org/>. – ISBN 012553180X
- [Popov u. a. 2006] POPOV, Stefan ; GÜNTHER, Johannes ; SEIDEL, Hans-Peter ; SLUSALLEK, Philipp: Experiences with Streaming Construction of SAH KD-Trees. In: WALD, Ingo (Hrsg.) ; PARKER, Steven G. (Hrsg.) ; IEEE (Veranst.): *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. Salt Lake City, USA : IEEE, September 2006. – ISBN 1-4244-0693-5, S. 89–94. – Best Paper Award
- [Rechenberg u. Pomberger 2002] KAPITEL C2.4. In: RECHENBERG, Peter ; POMBERGER, Gustav: *Informatikhandbuch*. 3. Hanser Verlag, 2002, S. 307–308
- [Reshetov u. a. 2005] RESHETOV, Alexander ; SOUPIKOV, Alexei ; HURLEY, Jim: Multi-level ray tracing algorithm. In: *ACM Trans. Graph.* 24 (2005), Nr. 3, S. 1176–1185. <http://dx.doi.org/http://doi.acm.org/10.1145/1073204.1073329>. – DOI <http://doi.acm.org/10.1145/1073204.1073329>. – ISSN 0730-0301
- [Wächter 2008] WÄCHTER, Carsten: *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*, Universität Ulm, Diss., 2008
- [Wächter u. Keller 2006] WÄCHTER, Carsten ; KELLER, Alexander: Instant Ray Tracing: The Bounding Interval Hierarchy. In: AKENINE-MÖLLER, T. (Hrsg.) ; HEIDRICH, W. (Hrsg.): *Rendering Techniques 2006 (Proc. of 17th Eurographics Symposium on Rendering)*, 2006, S. 139–149
- [Wächter u. Keller 2007] WÄCHTER, Carsten ; KELLER, Alexander: Terminating Spatial Hierarchies by A Priori Bounding Memory. In: *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007

- [Wald 2004] WALD, Ingo: *Realtime Ray Tracing and Interactive Global Illumination*, Diss., 2004. <http://www.sci.utah.edu/~wald/Publications/2004///WaldPhD/download//phd.pdf>
- [Wald 2007] WALD, Ingo: On fast Construction of SAH based Bounding Volume Hierarchies. In: *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*, 2007
- [Wald u. a. 2007] WALD, Ingo ; BOULOS, Solomon ; SHIRLEY, Peter: Ray tracing deformable scenes using dynamic bounding volume hierarchies. In: *ACM Trans. Graph.* 26 (2007), Nr. 1, S. 6. <http://dx.doi.org/http://doi.acm.org/10.1145/1189762.1206075>. – DOI <http://doi.acm.org/10.1145/1189762.1206075>. – ISSN 0730-0301
- [Wald u. Havran 2006] WALD, Ingo ; HAVRAN, Vlastimil: On building fast kd-trees for ray tracing, and on doing that in $O(N \log N)$. (2006), September, 61–69. <http://www.cgg.cvut.cz/members/havran/ARTICLES/ingo06rtKdtree.pdf>
- [Wald u. a. 2006] WALD, Ingo ; IZE, Thiago ; KENSLER, Andrew ; KNOLL, Aaron ; PARKER, Steven G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. In: *ACM Transactions on Graphics* (2006), 485–493. <http://www.sci.utah.edu/~wald/Publications/2006///Grid/download//grid.pdf>. – (Proceedings of ACM SIGGRAPH 2006)
- [Whitted 1980] WHITTED, Turner: An Improved Illumination Model for Shaded Display. 23 (1980), Juni, Nr. 6, S. 343–349. – ISSN 0001-0782
- [Williams u. a. 2005] WILLIAMS, Amy ; BARRUS, Steve ; MORLEY, R. K. ; SHIRLEY, Peter: An efficient and robust ray-box intersection algorithm. In: *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. New York, NY, USA : ACM, 2005, S. 9
- [Zuniga u. Uhlmann 2006] ZUNIGA, Miguel R. ; UHLMANN, Jeffrey K.: Ray Queries with Wide Object Isolation and the DE-Tree. In: *journal of graphics tools* 11 (2006), Nr. 3, S. 27–45

Abkürzungsverzeichnis

AABB	Axis Aligned Bounding Box
AoS	Arrays of structures
BIH	Bounding Intervall Hierarchy
BSP	Binary Space Partitioning
BVH	Bounding Volume Hierarchies
SAH	Surface Area Heuristic
SIMD	Single Instruction Multiple Data
SoA	Structure of Arrays

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Ort, Datum

Axel Tetzlaff