Fachbereich Informatik

Parallele SAH KD-Tree Konstruktion auf GPUs für Raytracing dynamischer Szenen

Masterarbeit von

Sascha Kolodzey

5. Oktober 2014

Gutachter:
Prof. Dr. Oliver Vornberger
Prof. Dr.-Ing. Elke Pulvermüller

Danksagung

Hiermit möchte ich mich bei allen Menschen bedanken, die mich bei dieser Arbeit unterstützt haben. Insbesondere bei Henning Wenke, der immer Zeit hatte, um über Probleme zu diskutieren und mich mit hilfreichen Anregungen unterstützte. Ferner bei Frau Prof. Dr.-Ing. Elke Pulvermüller und Herrn Prof. Dr. Oliver Vornberger, die diese Arbeit begutachten. Außerdem möchte ich meiner Mutter für das Korrekturlesen und meinem Vater für die Unterstützung während des Studiums danken. Für die emotionale Unterstützung während der Arbeit möchte ich mich bei meiner Freundin Annika Wegener bedanken.

Zusammenfassung

Ein KD-Tree unter Verwendung der *Surface Area Heuristic* (SAH) zur Wahl der Split-Ebenen gilt in der Praxis als eine der besten Beschleunigungsstrukturen für das Raytracing. Gegenstand der vorliegenden Masterarbeit ist die Formulierung eines parallelen Algorithmus zur Konstruktion eines SAH KD-Trees sowie dessen vollständige Implementation auf einer GPU. Der Algorithmus in dieser Arbeit verfolgt den Ansatz, die Events eines Primitives beim Clipping nur auf der Split-Achse zu modifizieren. Die in vorherigen Ansätzen aufwändige Sortierung der geclippten Events auf jeder Ebene wird dadurch vermieden.

Abstract

A KD-Tree using the *Surface Area Heuristic* (SAH) to select the split planes is one of the best acceleration structures for raytracing applications. Subject of this master's thesis is to develop a parallel algorithm to construct a SAH KD-Tree, which gets fully implemented on a GPU. The algorithm in this thesis does only modify clipped events laying on the same axis as the split. This approach is different to previous ones and avoids the costly sorting operation of clipped events at each level in the construction process.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Raytracing-Algorithmen werden heutzutage hauptsächlich im professionellen Grafikmarkt zur Generierung fotorealistischer Darstellungen eingesetzt, schwerpunktmäßig beim Rendern von 3D Szenen (Beispiel siehe Abb. 1.1). Im Rahmen dieser Entwicklungsprozesse wird zunehmend eine geringere Bildqualität in Kauf genommen zugunsten der Echtzeitfähigkeit eines Raytracers. Echtzeit-Raytracer können zur effizienten Generierung von Grafikinhalten beitragen.

Für das Beschleunigen von Raytracing-Algorithmen ist der SAH KD-Tree die am meist verwendete Datenstruktur [9]. Ein SAH KD-Tree wird unter Berücksichtigung der *Surface Area Heuristic* (SAH) konstruiert. Die Beschleunigung wird durch effizientere Schnittpunkttests mit den Objekten in der Szene erreicht. Für dynamische Szenen muss der KD-Tree bei jeder Änderung eines Objektes neu erstellt werden. Die Erstellung von SAH KD-Trees kann jedoch sehr viel Zeit in Anspruch nehmen, wodurch Raytracing-Algorithmen ihre Echtzeitfähigkeit beim Rendern dynamischer Szenen verlieren können.

Ziel dieser Arbeit ist deshalb, die Formulierung eines parallelen Algorithmus zur Konstruktion eines SAH KD-Trees, der auf einem modernen Grafikprozessor (GPU) implementiert wird. Moderne GPUs zeichnen sich durch ihre auf parallele Berechnungen ausgelegte Hardware aus und können somit zur Beschleunigung eines parallel formulierten Algorithmus beitragen. Durch die beschleunigte Konstruktion des SAH KD-Trees wird die Echtzeitfähigkeit eines Raytracers von dynamischen Szenen unterstützt.

Abbildung 1.1: Rekursives Raytracing, Screenshot auf Basis der in dieser Arbeit entstandenen Software, BMW Modell von [14]

1.2 Methodisches Vorgehen

Zu Beginn dieser Arbeit wird im Kapitel 2 die Rendering-Technik *Raytracing* vorgestellt. Es folgt eine Erläuterung zur Datenstruktur KD-Tree sowie die Motivation der *Surface Area Heuristic* und die Vorstellung von parallelen Standardalgorithmen. Im weiteren Verlauf liegt der Fokus auf der Formulierung des parallelen Algorithmus, bestehend aus einer Initialisierungs- und Konstruktionsphase. Für eine Laufzeitanalyse werden die theoretischen Eigenschaften des formulierten Algorithmus im Kapitel 5 analysiert. Anschließend werden Implementationsdetails und Optimierungen des Algorithmus für eine moderne GPU unter Verwendung von CUDA dargestellt. Abschließend werden die Ergebnisse dieser Arbeit präsentiert.

Kapitel 2

Einführung

2.1 Echtzeit-Raytracing

Generell beschreibt das Raytracing ein Verfahren zur Generierung zweidimensionaler Bilder aus einer dreidimensionalen Szene (Rendering). Hierbei werden von einem imaginären Augpunkt aus Strahlen (Rays) durch jeden Bildpunkt (Pixel) in die Szene hinein verfolgt (Tracing) und auf Schnittpunkte mit den vorhandenen Objekten geprüft (siehe Abb. 2.1). Mit Hilfe der Objekteigenschaften und eines Beleuchtungsmodells ist es möglich, die Pixel einzufärben. Weiterhin können auch transparente und spiegelnde Materialien sehr einfach durch dieses Verfahren umgesetzt werden, indem der Algorithmus rekursiv fortgesetzt wird.

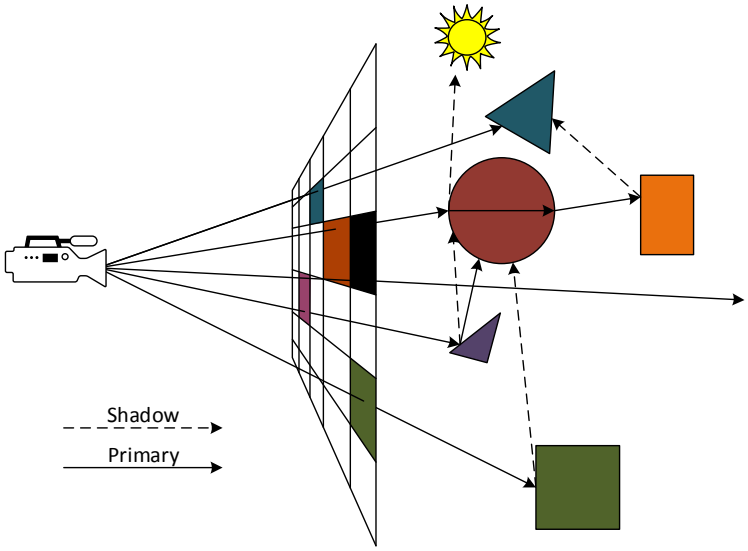

Abbildung 2.1: Rekursives Raytracing, Reflektion, Refraktion und Schatten sind möglich

Ein Ray kann auf Basis des getroffenen Materials (z. B. spiegelnd) neue Rays erzeugen. Aus dieser Sicht ist das Raytracing ein Rendering-Algorithmus, der phy-

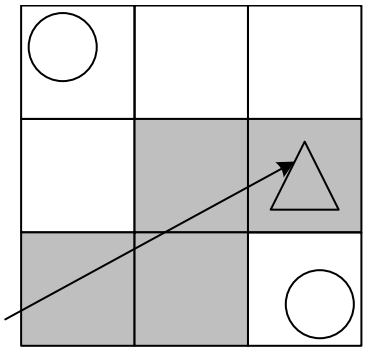

Abbildung 2.2: Reguläres Grid, nur gefärbte Bereiche werden untersucht

sikalische Gesetzmäßigkeiten approximiert und somit in der Lage ist, einen hohen Grad an fotorealistischen Effekten zu erzeugen [25]. Deshalb wird Raytracing hauptsächlich in nicht echtzeitkritischen Applikationen (z. B. Filme) verwendet, bei denen das Rendern eines Bildes mehrere Minuten/Stunden in Anspruch nehmen darf, um die gewünschte Qualität zu erreichen (High-Quality Raytracing). Im Gegensatz zum High-Quality Raytracing wird beim Echtzeit-Raytracing der Kompromiss geschlossen zwischen der benötigten Zeit zum Erstellen eines Bildes und dessen Qualität, um akzeptable Bildwiederholungsraten zu erreichen. In den letzten Jahren wurden viele Echtzeit-Raytracer auf modernen Grafikkarten (GPUs) implementiert und waren in der Lage, in Echtzeit Ergebnisse zu produzieren [12].

Beschleunigungsstrukturen

Für hinreichend komplexe Szenen ist es nicht praktikabel, den für jeden Pixel erzeugten Ray mit jedem Primitiv (z.B. Dreieck) innerhalb der Szene auf einen Schnittpunkt hin zu prüfen. Die Laufzeit eines solchen Ansatzes steigt je Pixel linear mit der Anzahl der Primitive ($\mathcal{O}(n), n$ Anzahl der Primitive) [25]. Es hat sich deshalb etabliert, eine zusätzliche Datenstruktur zu Hilfe zu nehmen, um Gruppen von Primitiven von vornherein für einen Ray ausschließen zu können und somit keine aufwändigen Schnittpunkttests durchführen zu müssen (siehe Abb. 2.2). Die Suche nach Gruppen von Primitiven wird als *Traversierung* bezeichnet.

Für statische Szenen genügt es, eine Beschleunigungsstruktur einmal zu berechnen und diese dann über die gesamte Laufzeit hin zu verwenden. Bei dynamischen Szenen ist dies jedoch bei jeder Positionsveränderung der Primitive innerhalb der Szene erforderlich. Dadurch wird auch der Aufwand für die Erzeugung zu einem wichtigen Faktor der Laufzeit eines Raytracers.

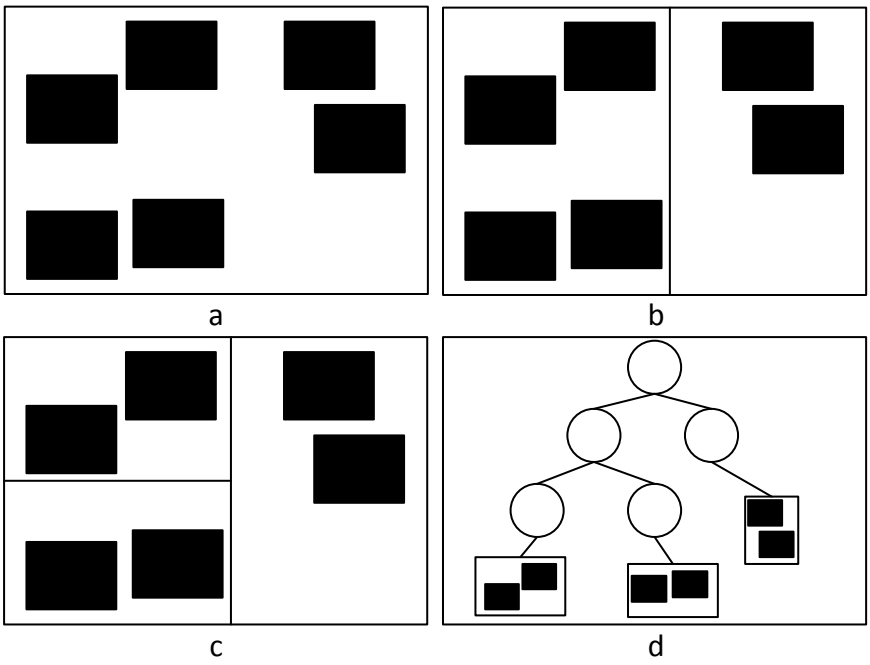

Abbildung 2.3: KD-Tree Konstruktion, erste Ebene (a), zweite Ebene (b) dritte Ebene (c), Baumstruktur (d)

2.2 KD-Tree

Binary Space Partitioning (BSP) Trees zerteilen die Szenen adaptiv. Dies ist ein entscheidender Gegensatz zu regulären Grids (vgl. Abb. 2.2), die lange Zeit für das Raytracing eingesetzt worden sind [6]. Die Zerlegung des Raumes findet bei BSP-Trees unter Berücksichtigung der sich im Raum befindlichen Geometrien statt. Als Konsequenz daraus können BSP-Trees Verteilungen von irregulären Geometrien effizienter speichern [25]. Eine Variante eines BSP-Trees ist der KD-Tree (siehe Abb. 2.3). Ein KD-Tree ist ein unbalancierter und binärer Suchbaum. Er fordert, dass eine gewählte Split-Ebene senkrecht zu einer der kartesischen Koordinatenachsen verläuft. Die Konstruktion eines KD-Trees startet mit einer Axis-Aligned-Bounding-Box (AABB), die die gesamte Szene beinhaltet (vgl. Abb. 2.3). Wenn sich mehr Primitive in der AABB befinden als ein vorher festgelegtes Minimum, wird die AABB in zwei Hälften geteilt. Nun werden die Primitive in drei Kategorien eingestuft:

- Primitiv befindet sich *unterhalb* der Split-Ebene.
- Primitiv befindet sich *oberhalb* der Split-Ebene.
- Primitiv *überdeckt* die Split-Ebene.

Primitive, die vollständig unterhalb oder oberhalb der Split-Ebene liegen, können direkt auf die beiden resultierenden Kinder (Child-Nodes) verteilt werden. Überdeckt

ein Primitiv die Split-Ebene, muss es an beide Child-Nodes weitergereicht werden (Clipping). Dieses Vorgehen wird in der Regel bei linearen Implementierungen für alle entstehenden Nodes rekursiv fortgesetzt, bis eine maximale Tiefe erreicht ist. Für die Konstruktion stellt sich die Frage, an welcher Stelle eine Node geteilt werden sollte, damit der resultierende KD-Tree *optimal* wird. Mit *optimal* ist die Minimierung der theoretischen Kosten zu verstehen, die ein Ray beim Traversieren durch den KD-Tree generiert [9]. Die Konstruktion eines *optimalen* KD-Trees für das Raytracing ist wahrscheinlich ein NP-schweres Problem [1]. Deshalb sind Algorithmen auf gute Heuristiken angewiesen, um die Datenstruktur in angemessener Zeit zu erzeugen [16]. Für das Raytracing hat sich vor allem eine Heuristik durchgesetzt: die *Surface Area Heuristic*.

2.3 Surface Area Heuristic

Die *Surface Area Heuristic* [7] (SAH) ist eine Heuristik zur Wahl der Split-Ebene innerhalb einer Node eines KD-Trees. Ein SAH KD-Tree gilt als eine der besten Datenstrukturen, um das Raytracing zu beschleunigen [28]. Die SAH basiert auf einem Kostenmodell. Hierbei wird folgende Erkenntnis berücksichtigt: Die Wahrscheinlichkeit, dass ein Ray ein konvexes Objekt schneidet ist proportional zu der Größe der Oberfläche des Objektes [13].

Bei der Bearbeitung einer Node innerhalb des Konstruktionsprozesses ergeben sich zwei Möglichkeiten: Die erste Möglichkeit ist, ein Blatt zu erzeugen. Alle Rays, die diese Node passieren, werden gegen alle Objekte innerhalb der Node getestet. Die Kosten $\mathcal{C}$ ergeben sich wie folgt:

$$\mathcal{C} = \sum_{i \in n} c_{obj}(o_i) \tag{2.1}$$

Wobei n die Anzahl der Primitive innerhalb der Node ist.

Die zweite Möglichkeit ist, die Node aufzuteilen. Die Kosten für einen Ray, der auf die Node trifft, können nun durch die SAH Funktion von Gordon Müller und Dieter W. Fellner [15] beschrieben werden:

$$\mathcal{C}_H(j, axis) = C_{trav} + \frac{S_B(L_j)}{S_B(H)} \cdot \sum_{i \in L_j} c_{obj}(o_i) + \frac{S_B(R_j)}{S_B(H)} \cdot \sum_{i \in R_j} c_{obj}(o_i) \tag{2.2}$$

- $C_{obj}(o)$ gemittelte Kosten für einen Schnittpunkttest mit dem Objekt o,
- $axis \in X, Y, Z$ (kartesische Koordinatenachsen),
- C_{trav} Kosten für das Traversieren in das rechte oder linke Kind,

- $S_B(X)$ Surface Area der Teilszene X,
- $S(H)$ Surface Area der Node,
- H Hierarchie mit L_j und R_j als direkte Kinder bei einem Split entlang der Achse $axis$, sodass $|H| = n, |L_j| = j, |R_j| = n - j, j \in 1, 2, ..., n - 1$ gilt. $|H|$ ist die Anzahl der Objekte, die auf die Kinder verteilt werden.

Die Werte für die Kosten einer Schnittpunktberechnung c_{obj} und die Kosten für die Traversierung C_{trav} sind in der Regel von der Hardware abhängig und werden anhand der benötigten Instruktionen festgelegt. Die Terme $\frac{S_B(L_j)}{S_B(H)}$ und $\frac{S_B(R_j)}{S_B(H)}$ ergeben sich aus Überlegungen der Geometrischen Wahrscheinlichkeit. Für ein konvexes Volumen A, welches sich innerhalb eines konvexen Volumens B befindet gilt (siehe Abb. 2.4):
Die bedingte Wahrscheinlichkeit $P(A|B)$, dass ein Ray, der das Volumen B passiert, auch das Volumen A passiert, ergibt sich aus dem Quotienten der beiden Oberflächen (Surface Areas) [7].

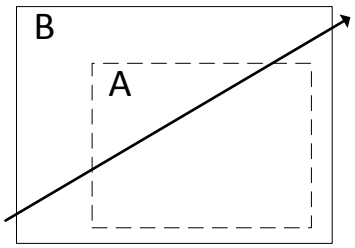

Abbildung 2.4: Ray trifft Volumen A innerhalb von Volumen B

Mögliche Split-Kandidaten (Events), an denen die SAH ausgewertet wird, ergeben sich aus den Ebenen der AABBs, die zuvor für jedes Primitiv berechnet worden sind (siehe Abb. 2.5). Events werden auch "perfect splits" genannt und liefern immer einen besseren SAH Wert als ein Split innerhalb des betrachteten Primitivs, da sich die SAH monoton fallend oder monoton steigend zwischen zwei aufeinanderfolgenden Events verhält [27].

$$p(A|B) = \frac{S_A}{S_B} \tag{2.3}$$

$$\mathcal{C} = \sum_{i \in N} c_{obj}(o_i) \tag{2.4}$$

$$\mathcal{SAH}_{min} = min(C_H(j, axis)) \tag{2.5}$$

Die beste Split-Ebene lässt sich durch das Minimum der SAH für alle Events (siehe Abb.2.5) bestimmen.

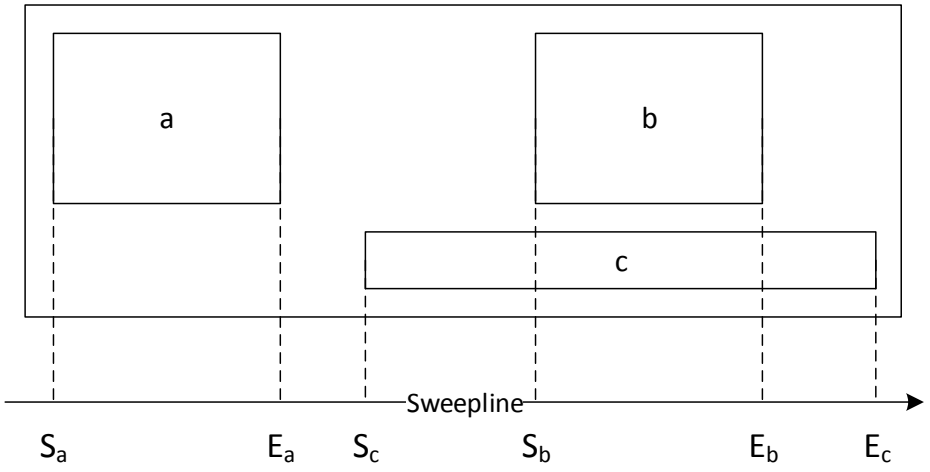

Abbildung 2.5: "Perfect Splits", S=Start, E=End

Die SAH setzt voraus, dass die Richtungen der Rays gleich verteilt im Objektraum liegen und deren Augpunkt hinreichend weit vom Objekt entfernt ist [26]. Die durch einen Raytracer erzeugten Rays liegen in der Regel nicht gleich verteilt im Raum, wodurch die Linearität der Kosten nicht erfüllt ist. In der Praxis wird dieser Fehler jedoch akzeptiert [27].

Eine oft angewandte Modifikation der SAH ist die Priorisierung von Events, für die sich auf einer der beiden Seiten keine Primitive befinden und somit einen leeren Teilbereich aufspannen (cut off empty space). Wenn einer der beiden Bereiche L oder R keine Primitive enthält, wird die SAH mit einem konstanten Faktor $\lambda(j)$ (in diesem Fall 0.8) multipliziert [27]:

$$\lambda(j) = \begin{cases} 0.8 \ ; \ |L| = 0 \wedge |R| = 0 \\ 1 \ ; \ sonst \end{cases} \tag{2.6}$$

Lineare Berechnung der SAH

Für eine gegebene Liste von sortierten Events lässt sich die SAH für jedes Event sequentiell mit einer Sweepline berechnen. Hierzu läuft die Sweepline alle Events von links nach rechts ab, wertet die SAH an jedem Event aus und merkt sich, wie viele END- bzw. START-Events sie passiert hat (vgl. Abb. 2.5). Die Sweepline wählt nach der Berechnung einer SAH immer das Minimum des gerade berechneten Wertes und das Minimum aller Vorgänger (siehe Algorithmus 1).

Algorithm 1 Lineare SAH-Berechnung

```
primwsBelow ← 0
primsAbove ← primCount
sah ← ∞
for i ∈ [0, events.count) do
    e ← events[i]
    if e.type == End then
        primsAbove ← primsAbove - 1
    end if
    sah ← min(sah, SAH(below, above, nodeAABB, e.plane))
    if e.type == Start then
        primwsBelow ← primwsBelow + 1
    end if
end for
```

2.4 Parallele Standardalgorithmen

Im weiteren Verlauf der Arbeit kommen vor allem drei parallele Standard Algorithmen häufig zur Anwendung: *Reduction*, *Scan* und *Compaction*. Die drei Algorithmen werden in ihrer Funktionsweise dargestellt, außerdem wird auf die Möglichkeit einer parallelen Implementierung eingegangen.

2.4.1 Reduction

Reduction [10] gibt es in zwei Varianten: *Reduction* und *Segmented-Reduction*. *Reduction* arbeitet mit einem binären assoziativen Operator $\bigoplus$ (z.B. $min, max, +, -$), einem Array mit N Daten $[X_0, X_1, ..., X_{n-1}]$ und liefert als Ergebnis einen Wert R, der die Verknüpfung aller Elemente aus N darstellt ($R = X_0 \bigoplus X_1 \bigoplus ... \bigoplus X_{n-1}$).
Segmented-Reduction berechnet dieses Ergebnis auf konsekutiven Abschnitten des Arrays N. Das Ergebnis ist somit ein Array der Größe K, wenn K die Anzahl der Segmente ist.

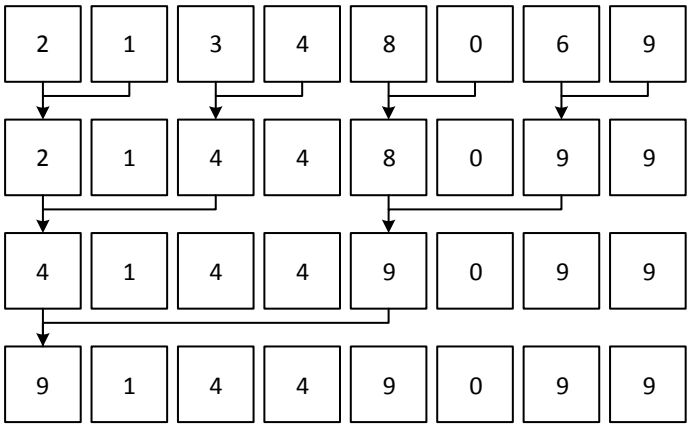

Abbildung 2.6: *Reduction*, Operator: max

Für eine parallele Implementierung ist die Assoziativität des gewählten Operators von Bedeutung. Es darf von keiner Relevanz sein, in welcher Reihenfolge die Elemente verknüpft werden. Eine mögliche parallele Implementierung basiert auf einem binären Baum (siehe Abb. 2.6). Alle Elemente auf einer Ebene können parallel verarbeitet werden.

2.4.2 Scan

Auch der *Scan* [10] existiert in zwei Varianten: Der *Inclusive-Scan* und der *Exclusive-Scan*. Der *Inclusive-Scan* operiert mit einem binären assoziativen Operator $\bigoplus$ auf einem N-elementigen Array. Der Output ist hierbei ein Array der Länge N, welches für jeden Index die Verknüpfung aller vorherigen Elemente (inklusiv des Elements am betrachteten Index) beinhaltet $(X_0, X_0 \bigoplus X_1, X_0 \bigoplus X_1 \bigoplus X_2, ..., X_0 \bigoplus X_1 \bigoplus ... \bigoplus X_{n-1})$.

Der *Exclusive-Scan* liefert die Verknüpfung aller Elemente vor einem Index (exklusiv des betrachteten Index), wobei I an der Stelle 0 steht und das neutrale Element bezüglich des Operators $\bigoplus$ ist $(I, X_0, X_0 \bigoplus X_1, X_0 \bigoplus X_1 \bigoplus X_2, ..., X_0 \bigoplus X_1 \bigoplus ... \bigoplus X_{n-2})$.

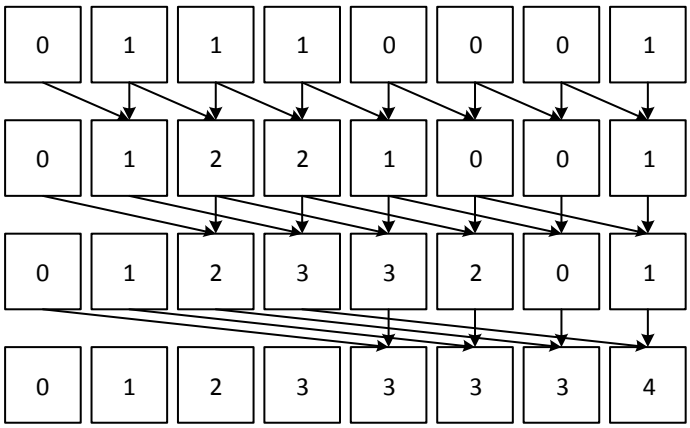

Abbildung 2.7: *Inclusive-Scan*, Operator: +

Die parallele Implementation ist der der *Reduction* ähnlich (siehe Abb. 2.7). Auch der *Scan* ist auf einen assoziativen Operator angewiesen. Ein *Exclusive-Scan* mit dem + Operator wird auch *Prefix Sum* genannt. Die Einträge des berechneten Arrays geben die Summe aller Elemente vor einem Element an.

2.4.3 Compaction

Die *Compaction* [2] beschreibt eine Methode, diejenigen Daten aus einem heterogenen Vektor zu extrahieren, die einer vorher definierten Bedingung genügen. Dies ist insbesondere dann von Vorteil, wenn eine große Menge von Daten vorliegt, jedoch nur ein Teil für eine Berechnung von Interesse ist. Der resultierende, meist kleinere Vektor, gestaltet kommende Operationen effizienter (siehe Abb. 2.8).

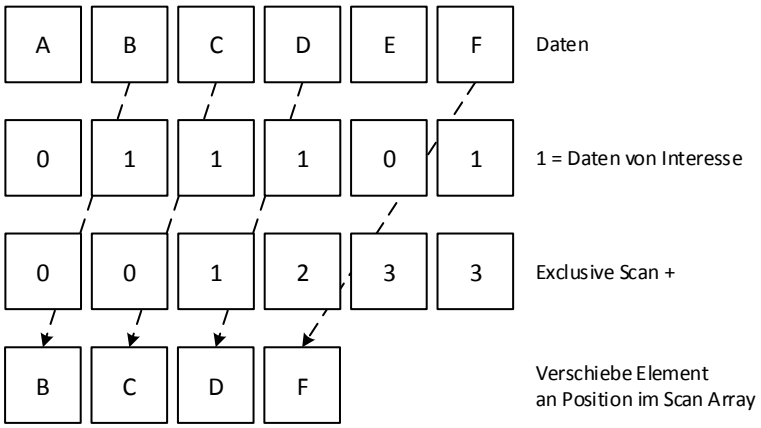

Abbildung 2.8: *Compaction*

Kapitel 3

Stand der Technik und Ansatz dieser Arbeit

Einen nennenswerten Beitrag im Bereich parallele KD-Tree Konstruktion auf GPUs erbrachte im Jahr 2008 die Arbeit *Real-Time KD-Tree Construction on Graphics Hardware* [29]. Der parallele Algorithmus von Zhou et al. erstellt den KD-Tree nicht mittels DFS (depth-first search) sondern mit einer BFS (breadth-first search) (siehe Abb. 3.1). Durch diesen Ansatz wurde es erstmals möglich, das parallele Potential einer GPU auf die Erstellung jeder Ebene des KD-Trees zu fokussieren. Dies war laut Zhou et al. der entscheidende Unterschied zu allen vorangegangenen Arbeiten. Zhou et al. erstellten jedoch keinen vollständigen SAH KD-Tree. Ihr Algorithmus verwendet in den ersten Ebenen die zwei Heuristiken *Median-Split* [9] und *Empty-Space-Partitioning* [27] zur Bestimmung der Split-Ebenen. Eine exakte Auswertung der SAH geschieht nur in den tieferen Ebenen des Trees [29].

Wu et al. entwickelten 2011 einen parallelen SAH KD-Tree Algorithmus, der auf einer GPU implementiert ist und als einer der ersten in der Lage war, die Qualität von aktuellen offline CPU Implementationen zu erreichen [28]. Auch Wu et al. konstruieren den KD-Tree mit einer BFS. Der entscheidende Unterschied zu Zhou et al. ist die effiziente und exakte Berechnung der SAH auf allen Ebenen des KD-Trees. Hierzu wird ein Ansatz verwendet, der auf dem parallelen Algorithmus *Scan* basiert, um

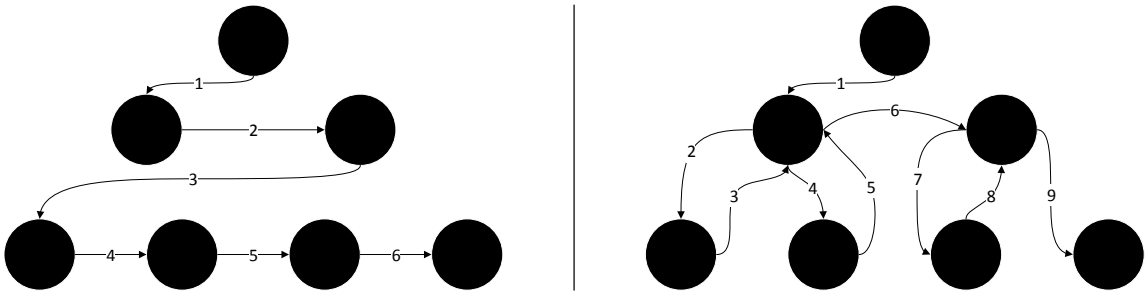

Abbildung 3.1: BFS (links) / DFS (rechts)

die Anzahl der Primitive über und unter den möglichen Split-Kandidaten parallel zu bestimmen. Zusätzlich kommt ein *Bucket-Based* Sortier-Algorithmus zum Einsatz, der die geklippten Primitive und die daraus neu entstandenen Events wieder ordnet. Das Sortieren der neuen Events hat laut Wu et al. den größten Anteil (35%) an der Laufzeit ihrer Implementation [28].

Ansatz dieser Arbeit

In dieser Arbeit wird ein paralleler Algorithmus zur Konstruktion eines SAH KD-Trees formuliert, der keine aufwändige Sortierung bei den durch das Clipping neu entstehenden Events vorsieht. Um die Sortierung zu vermeiden, werden die Events nur auf der Primärachse exakt gegen die Split-Ebene geclippt. Events auf den Sekundärachsen werden nicht exakt angepasst, wodurch eine Sortierung, bedingt durch geclippte Events über die gesamte Konstruktion des KD-Trees vermieden wird.

Kapitel 4

Parallele SAH KD-Tree Konstruktion

4.1 Einleitung und Überblick

Im folgenden Kapitel wird ein entwickelter Algorithmus für eine parallele Konstruktion eines SAH KD-Trees vorgestellt. Hierzu wird der Algorithmus in seine Komponenten zerlegt und die Funktionsweise dieser Komponenten mit Hilfe von Pseudocode Beispielen deutlich gemacht. Im Pseudocode kommt es vor allem zur Verwendung der Schlüsselwörter ***for each*** *Elem* $e \in m$ ***in parallel do***. Diese beschreiben die parallele Verarbeitung von mehreren Elementen e aus einer Menge m und sind zusätzlich mit einem Index nummeriert ($e.id$). Die Konstruktion gliedert sich in eine **Initialisierungsphase** (Phase I) und eine **Konstruktionsphase** (Phase II).

Algorithm 2 CreateSAHKDTree

```
Phase I
    CreateAABBs()
    CreateEventList()
    SortEvents(events)
    CreateRootNode()

Phase II
    while Tree-Levels not done do
        ComputeBestSplits()
        ClippingAndPartitioning()
        CreateChildsAndInitParents()
        MakeLeaves()
    end while
```

In der Initialisierungsphase werden zuerst alle AABBs der Primitive generiert. Im Anschluss daran kann auf Basis der AABBs der Primitive die AABB der gesamten Szene berechnet werden. Danach werden die Events auf allen drei Achsen erzeugt und auf jeder Achse sortiert, um somit eine parallele Berechnung der SAH zu ermöglichen. Zum Schluss wird die Root-Node initialisiert.

Die Konstruktionsphase arbeitet in Schritten (siehe Algorithmus 2 while-Schleife), wobei jeder Schritt eine Ebene des Trees verarbeitet. Eine Gliederung der verwendeten Sub-Algorithmen ist im Pseudocode 2 dargestellt. In jedem Schritt werden zuerst die SAH-Werte berechnet und die bestmöglichen Splits bestimmt (*ComputeBestSplits*). Danach kommt es zum Clippen der Primitive an den gewählten Splits und es findet eine Verteilung auf die entstehenden Child-Nodes statt (*ClippingAndPartitioning*). Es folgt die Initialisierung der Nodes auf der aktuellen Ebene und die Erstellung ihrer Child-Nodes (*CreateChildsAndInitParents*). Anschließend wird geprüft, ob eventuell Child-Nodes existieren, die einem Leaf-Kriterium genügen. Sollte dies der Fall sein, werden diese zu einer Leaf-Node transformiert (*MakeLeaves*).

4.2 Initialisierungsphase

4.2.1 Daten

Der Algorithmus nutzt für die Konstruktion vier Datenstrukturen: Node-Data, Event-Data, Split-Data und Leaf-Data. Die Datenfelder und ihre Bedeutung innerhalb der Datenstrukturen sind in der Tabelle 4.1 abgebildet. Auf die genaue Bedeutung wird an der jeweiligen Stelle ihrer Verwendung nochmals detailliert eingegangen.

Die Datenstrukturen werden in Listen verwaltet und sind über die gesamte Laufzeit des Algorithmus global sichtbar. Die Listen sind wie folgt benannt:

- ***activeNodes*** - verwaltet alle Nodes, die sich auf der aktuell verarbeiteten Ebene des Trees befinden.
- ***childNodes*** - speichert die auf einer Ebene erzeugten Child-Nodes.
- ***interiorNodes*** - speichert alle Nodes, die bereits verarbeitet wurden.
- ***events*** - beinhaltet sortierte Events auf allen drei Achsen.
- ***splits*** - speichert die berechneten Splits auf der aktuellen Ebene des Trees.
- ***leaves*** - speichert alle entstandenen Leaf-Nodes des Trees.
- ***primAABBs*** - speichert für jedes Primitiv seine AABB.

- ***primitives*** - beinhaltet die Primitive, über denen der KD-Tree erzeugt wird.
- ***leafPrimitives*** - speichert Referenzen der Primitive innerhalb der Leaf-Nodes.

Tabelle 4.1: Algorithm-Data (Diese Tabelle ist als lose Seite nochmals angefügt und kann zum Verständnis der folgenden Abschnitte beitragen)

Node-Data	
Name	**Beschreibung**
pCount	Anzahl der Primitive in der Node
pBegin	Startindex der Primitive in der Node
aabb	AABB der Node
isLeaf	Node ist ein Blatt: true
split	Split-Ebene, auf der die Node geteilt wurde
axis	Split-Achse, auf der die Node geteilt wurde
leafId	Referenz auf Leaf-Data
leftChild	Referenz auf linke Child-Node
rightChild	Referenz auf rechte Child-Node

Event-Data	
Name	**Beschreibung**
type	START- / END-Type
plane	Schnittebene auf der Achse
node	Referenz auf dazugehörige Node
primId	Referenz auf dazugehöriges Primitive
aabb	AABB für das Clipping
axis	Split-Achse des Events
isLeaf	Event gehört zu einer Leaf-Node: true

Split-Data	
Name	**Beschreibung**
pBelow	Anzahl der Primitive unter dem Split
pAbove	Anzahl der Primitive über dem Split
axis	Split-Achse des Splits
sah	Berechnete Surface Area Heuristic
plane	Schnittebene auf der Achse
Leaf-Data	
Name	**Beschreibung**
pCount	Anzahl der Primitive innerhalb des Leafs
pBegin	Startposition der Primitive innerhalb des Leafs
primIds	Referenz auf enthaltene Primitive

4.2.2 AABB der Primitive und Szene

Die Berechnung der AABBs kann für jedes Primitiv parallel geschehen (siehe Algorithmus 3). Hierzu wird über die Eckpunkte des Primitives iteriert und diese einer vorher initialisierten AABB hinzugefügt. Durch die sequentielle Anwendung der *min* und *max* Operatoren auf die bereits bestehende AABB und auf die verbleibenden Punkte des Primitives, ergibt sich letztendlich die AABB, die durch das Primitiv aufgespannt wird. Nach der Berechnung aller AABBs der Primitive kann die AABB der gesamten Szene bestimmt werden. Die AABB der Szene ist wiederum Ausgangspunkt der Berechnung des KD-Trees. Hierzu muss der maximale und minimale Wert aller Primitive auf jeweils allen drei Achsen bestimmt werden. Zum Einsatz kommt hierfür der in Kapitel 2.4 vorgestellte parallele Algorithmus *Reduction* (siehe 2.6). Die *Reduction* muss jeweils einmal über die Min-Werte und Max-Werte der berechneten AABBs der Primitive durchgeführt werden.

Algorithm 3 CreateAABBs

```
foreach Primitiv p ∈ primitives in parallel do
    AABB aabb
    aabb.min ← +∞
    aabb.max ← −∞
    for Point p ∈ p.points do
        aabb.min ← min(p, aabb.min)
        aabb.max ← max(p, aabb.max)
    end for
    primAABBs[p.id] ← aabb
end foreach
sceneAABB.min ← ReduceMin(primAABBs)
sceneAABB.max ← ReduceMax(primAABBs)
```

4.2.3 Event-Liste

Nachdem die AABBs der Primitive erzeugt worden sind, können die Events für jede Achse erstellt werden. Auch dies geschieht für jedes Primitiv parallel. Ein Event bezeichnet einen möglichen Split-Kandidaten, für den in jedem Schritt der Konstruktion der SAH-Wert berechnet wird. In Abhängigkeit seiner Position auf den euklidischen Achsen hat ein Event entweder den Typ START oder END. Ein START-Event repräsentiert immer die kleinere Split-Ebene der durch die AABB des Primitives erzeugten Split-Ebenen auf einer Achse (vgl. 2.5). Der Pseudocode 4 stellt die Generierung der Events für jedes Primitiv dar. Ein Primitiv erzeugt genau sechs Events, auf jeder der drei Achsen jeweils zwei (START / END). Dazu wird über alle drei Achsen in einer äußeren Schleife und über alle zwei Event-Typen in einer inneren Schleife iteriert. Initial ist jedes Event eines Primitives in der Root-Node (Index=0) enthalten. Zu-

sätzlich speichert jedes Event seine genaue Split-Ebene und eine AABB (*aabb*), um eventuell, durch das spätere Clippen bedingt, Änderungen an der AABB aufnehmen zu können. Die AABB jedes Events ist initial eine Kopie der AABB des dazugehörigen Primitives (*primId*). Über die aktuell betrachtete Achse und den Event-Type lässt sich die Split-Ebene des Events bestimmen (*GetPlane*). Alle Daten eines Events sind nochmals in der Tabelle 4.1 zusammengefasst. Im Anschluss an die Erzeugung werden die Events auf allen drei Achsen sortiert.

Algorithm 4 CreateEvents

```
foreach AABB aabb ∈ primAABBs in parallel do
    Event e
    e.primId ← aabb.primId
    e.aabb ← aabb
    e.node ← 0
    for type ∈ {START, END} × axis ∈ {X, Y, Z} do
        e.type ← type
        e.plane ← GetPlane(e.aabb, axis, type)
        events[axis][2 · e.primId + type] ← e
    end for
end foreach
for axis ∈ {X, Y, Z} do
    SortEvents(events[axis])
end for
```

4.2.4 Root-Node

Die Root-Node ist die im Tree oberste Node und definiert die Ausgangssituation der Konstruktion. Zuerst wird die AABB auf die vorher berechnete AABB der Szene gesetzt (vgl. Algorithmus 5).

Algorithm 5 CreateRootNode

```
root.aabb ← sceneAABB
root.pCount ← primitives.count
root.isLeaf ← false
root.pBegin ← 0
```

Alle Primitive befinden sich per Definition innerhalb der Root-Node. Somit kann der *pCount* auf die Anzahl aller Primitive gesetzt werden. Generell wird die Root-Node kein Leaf sein, weshalb ihr *isLeaf* Flag auf *false* gesetzt wird. Während der Konstruktion speichert die Liste *activeNodes* alle *aktiven* Nodes. Eine *aktive* Node befindet sich auf der aktuell verarbeiteten Ebene des Trees. Zu Beginn der Konstruktion ist

die Root-Node die einzige Node in der Liste *activeNodes*. Die Primitive aller aktiven Nodes werden über die Event-Liste (*events*) verwaltet und sind über den Offset *pBegin* für jede aktive Node erreichbar. Das Feld gibt dementsprechend die Anzahl aller Primitive unterhalb einer Node an. Diese Information ist für weitere parallele Berechnungen von Bedeutung.

Für eine Node werden zusätzlich die gelisteten Daten in der Tabelle 4.1 gespeichert. Außerdem verfügt eine Node über zwei Referenzen auf ihre Child-Nodes, falls die Node kein Leaf ist. Dann nämlich besitzt sie die Informationen, auf welcher Achse sie geteilt worden ist (*axis*) und wo sich die Split-Ebene (*split*) auf der Achse befindet. Die Informationen bezüglich Achse und Split müssen gespeichert werden, um den KD-Tree im Anschluss an die Konstruktion traversieren zu können. Außerdem besitzt jede Leaf-Node eine Referenz (*leafId*) auf die in ihr befindlichen Primitive.

4.3 Konstruktionsphase

In der Konstruktionsphase erstellt jeder Schritt eine Ebene des KD-Trees. Für die Berechnung der maximalen Anzahl an Schritten wird generell eine Heuristik in der Größenordnung $\mathcal{O}(\log n)$ (n = Anzahl der Primitive) verwendet. Durch die Wahl der Heuristik ist die Laufzeit der Traversierung in einem KD-Tree festgelegt. Bei der Wahl einer Heuristik soll die Anzahl der maximalen Schritte möglichst klein bleiben, die Anzahl der Primitive in einem Leaf jedoch gleichzeitig möglichst gering sein. Eine gute Heuristik findet einen Kompromiss zwischen beiden Extremen. Die Traversierung des KD-Trees im *average case* (*worst case* $\mathcal{O}(n)$) besitzt eine Zeitkomplexität von $\mathcal{O}(\log n)$, wenn durch die Heuristik $\mathcal{O}(\log n)$ Ebenen erzeugt werden [25]. Matt Pharr und Greg Humphreys [25] verwenden für ihre KD-Trees die Heuristik $\mathcal{H}$ (4.1) und beschreiben sie als eine maximale Tiefe, die für eine Vielzahl von Szenen eine sinnvolle Tiefe generiert. Zur Anwendung dieser Heuristik kommt es auch im dargestellten Algorithmus.

$$\mathcal{H} = \lceil 8 + 1.3 \cdot \lfloor \log n \rfloor \rceil, n \geq 1 \tag{4.1}$$

Die Konstruktionsphase wird dementsprechend maximal $\mathcal{H}$ Schritte durchführen.

N_0				N_1				
S_0	S_1	E_0	E_1	S_2	E_2	S_3	E_3	Events
2	2	2	2	2	2	2	2	pCount
0	0	0	0	2	2	2	2	pBegin
1	1	0	0	1	0	1	0	type
0	1	2	2	2	3	3	4	sType
0	1	2	3	4	5	6	7	id
0	0	0	1	0	0	1	1	id – sType – pBegin
0	1	2	2	0	1	1	2	pBelow = sType - pBegin
2	2	1	0	2	1	1	0	pAbove = pCount - (id–sType-pBegin) - (~type)

Abbildung 4.1: *pBelow* / *pAbove* Berechnung

4.3.1 Berechnung der besten Split-Kandidaten: Surface Area Heuristic

Die *Surface Area Heuristic* (SAH) ist ein wesentliches Element zur Erstellung des SAH KD-Trees. Für die parallele Berechnung der SAH aller Events ist die parallele Bestimmung der Anzahl der Primitive ober- und unterhalb jedes Events erforderlich (vgl. Kapitel 2.3). Um dies zu ermöglichen, sind die Events in der Initialisierungsphase sortiert worden, sodass mit Hilfe einer *Prefix Sum* über die Event-Typen die Anzahl der START Events unterhalb eines Events bestimmt werden kann. Die Werte der Event-Typen sind dementsprechend mit 1 für START und 0 für END gewählt. Die Anzahl der END-Events lässt sich mit einer Umformulierung der *Prefix Sum* berechnen.

Algorithm 6 ComputeBestSplits

```
events.sType ← PrefixSum(events.types)
foreach Split split ∈ splits in parallel do
    split.sah ← ∞
    for axis ∈ {0, 1, 3} do
        e ← events[axis][split.id]
        pBelow ← e.sType - e.node.pBegin
        pAbove ← e.node.pCount - (split.id - e.sType) + e.node.pBegin
        MinusOneIfEndEvent(e, pAbove)
        sah ← SAH(e.node.aabb, axis, e.plane, pBelow, pAbove)
        if sah < split.sah then
            split ← {pAbove, pBelow, sah, axis, e.plane}
        end if
    end for
end foreach
foreach Node n ∈ activeNodes in parallel do
    Reduce(splits, 2· n.pBegin, 2· n.pCount)
end foreach
```

Der Algorithmus zur parallelen Berechnung der SAH ist im Pseudocode 6 dargestellt. In Abbildung 4.1 ist die parallele Berechnung für eine Anzahl von zwei aktiven Nodes, mit jeweils zwei enthaltenen Primitiven zu sehen. Die entsprechenden Datenfelder für jedes Event und die Zwischenergebnisse sind untereinander aufgetragen.

Wie zuvor beschrieben, wird zuerst eine *Prefix Sum* über die Event-Liste bestimmt. Danach kann die SAH für alle drei Events auf dem Index aller drei Achsen sequentiell ausgewertet werden. Von den drei sich ergebenden Möglichkeiten wird das Minimum ausgewählt.

Zuerst wird die Anzahl an Primitiven unterhalb des Events über eine Subtraktion des Feldes *pBegin* von der *Prefix Sum* der Event-Typen (*sType*) ermittelt. Alle Primitive unterhalb der betrachteten Node werden somit ausgeschlossen (vgl. Abb. 4.1).

Danach werden die Primitive oberhalb des Events berechnet. Wie in Kapitel 1 beschrieben, ergeben alle END-Events oberhalb eines Events die Anzahl der Primitive oberhalb an. Hierzu wird zuerst über eine Subtraktion der *Prefix Sum* vom Index des Events (*e.id*) die Anzahl der END-Events unterhalb eines Events bestimmt (vgl. Abb. 4.1). Um alle Primitive unterhalb der Node auszuschließen, wird zusätzlich die Anzahl aller Primitive unterhalb der betrachteten Node (*pBegin*) subtrahiert. Jetzt kann von allen Primitiven unterhalb des Events auf die oberhalb gelegenen geschlossen werden, indem das Zwischenergebnis von der Anzahl aller Primitive innerhalb der Node (*pCount*) subtrahiert wird (siehe Algorithmus 6). Falls es sich bei dem betrachteten Event um ein END-Event handelt, muss dies zusätzlich vom Ergebnis abgezogen werden, da ein END-Event immer links neben einem möglichen Split-Kandidaten eingeordnet wird (vgl. Abb. 2.5).

Anschließend wird das Ergebnis der berechneten SAH in einem Datensatz (*Split*) festgehalten und in die Liste *splits* eingetragen. Ein Split speichert, neben der Anzahl der Primitive ober- und unterhalb seiner Split-Plane, den berechneten SAH-Wert und die Split-Achse (siehe Tabelle 4.1). In der Liste *splits* steht jetzt das lokale Minimum der drei Events zu einem Index auf allen drei Achsen.

Zum Schluss wird über eine *Reduction* das globale Minimum aller Splits innerhalb jeder Node berechnet. Die Segmente für die *Reduction* ergeben sich aus den Nodes innerhalb der Liste *activeNodes* und lassen sich durch das Auslesen der Felder *pCount* und *pBegin* bestimmen. Da für jedes Primitiv innerhalb einer Node genau zwei Events existieren, können die ausgelesenen Werte mit dem Wert 2 multipliziert werden, um von der Anzahl der Primitive auf die Anzahl der Events zu schließen (vgl. Algorithmus 6). Nach Abschluss der *Reduction* liegen die minimalen SAH-Werte aller Splits eines Segmentes an dessen erster Position. Diese sind somit im weiteren Verlauf für jede Node über das Feld *pBegin* erreichbar.

4.3.2 Clipping und Partitioning

Nach der Berechnung der besten Splits müssen die Events gegen die Split-Ebene ihrer Node geclippt werden (Clipping), um sie dann auf die zwei entstehenden Child-Nodes verteilen zu können (Partitioning).

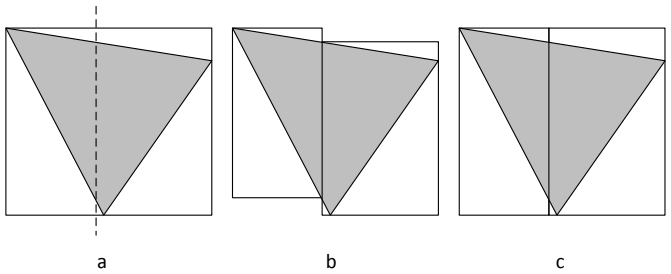

Abbildung 4.2: Split und AABB (a), AABB wird auf allen Achsen angepasst (b), AABB wird nur auf der Split-Achse angepasst (c)

Das Clippen der Events muss für alle drei Achsen durchgeführt werden. Ein Event kann dementsprechend in drei Kategorien eingestuft werden:

- Event liegt vollständig unterhalb des Splits. Die Position eines Events unterhalb des Splits lässt sich durch den Vergleich der Split-Plane mit dem minimalen Wert der AABB des Events auf der Split-Achse feststellen (vgl. Abb. 4.3).
- Event liegt vollständig oberhalb des Splits. Die Position eines Events oberhalb des Splits lässt sich durch den Vergleich der Split-Plane mit dem maximalen Wert der AABB des Events auf der Split-Achse feststellen (vgl. Abb. 4.3).
- Event überlagert den Split. Ist das Event weder vollständig unterhalb noch oberhalb vom Split liegend, überlagert das Event den Split. Hierbei wird noch einmal zwischen zwei Fällen unterschieden, ob das Event auf der gleichen Achse liegt wie der Split oder nicht. Wenn das Event auf der gleichen Achse liegt, erfolgt eine Prüfung, ob es sich um ein START- oder END-Event handelt. Wenn das Event ein START-Event ist, muss dieses an das rechte Child weitergegeben werden. Im Gegensatz dazu wird ein END-Event an das linke Child weitergegeben. Ist die Achse eines Events verschieden zu der Split-Achse, muss das Event in jedes der beiden Child-Nodes weitergereicht werden (vgl. Abb. 4.3).

Um die parallele Partitionierung aller Events auf den drei Achsen zu ermöglichen, wird ein konfliktfreies Pattern eingesetzt, welches ein Event auf Basis der eingestuften Kategorie an dafür individuelle Adressen schreibt. Da es im *worst-case* passieren kann, dass alle Events den Split überlagern und somit auf beide Child-Nodes übertragen werden müssen, ergibt sich für diesen Adressraum eine maximale Adressweite

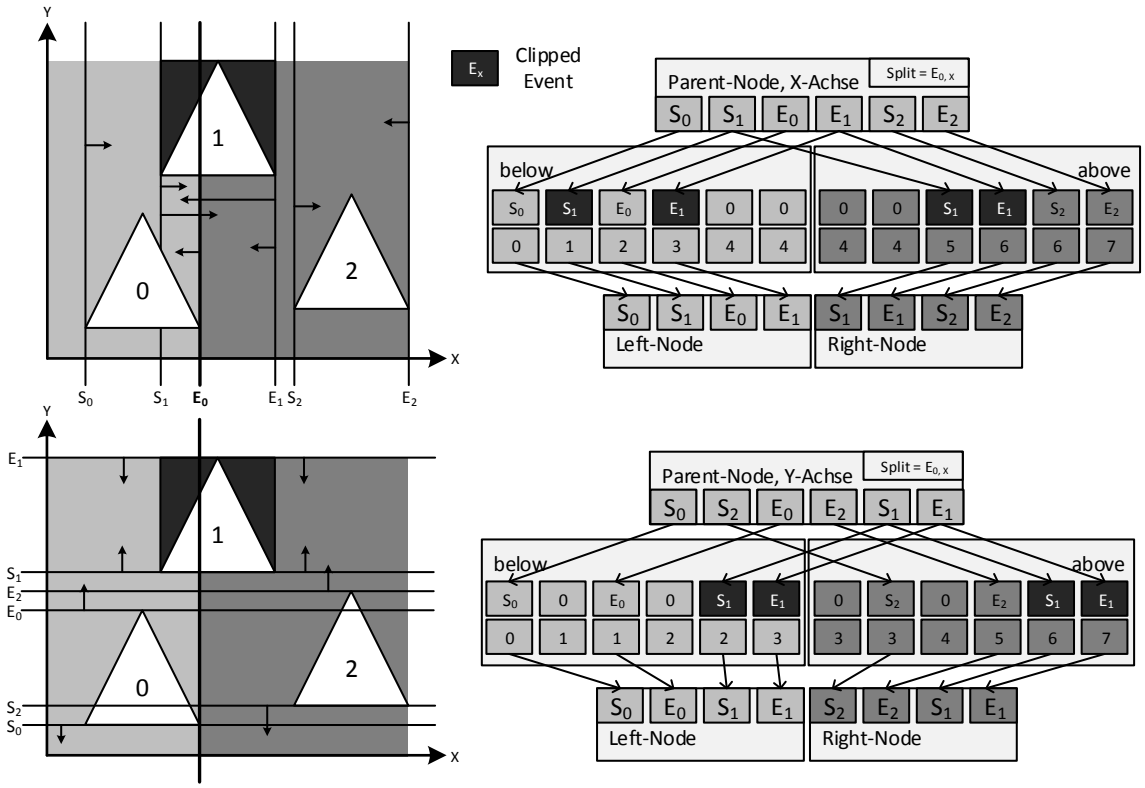

Abbildung 4.3: Clipping und Partitioning (liegt in vergrößerter Form lose bei und kann zum Verständnis des Algorithmus 7 beitragen)

von $4 \cdot n.pCount$ für jedes Event-Intervall einer Node.

Für das konfliktfreie Kopieren werden zwei Intervalle definiert: Events unterhalb des Splits fallen in das Intervall $[0, 2 \cdot n.pCount)$ und Events oberhalb des Splits in das Intervall $[2 \cdot n.pCount, 4 \cdot n.pCount)$. Diese sind in Abbildung 4.3 mit *below* und *above* dargestellt. Der untere Adressraum lässt sich über den Index des Events direkt bestimmen. Um in den oberen Adressraum zu schreiben, muss das untere Intervall hinzuaddiert werden $(+(2 \cdot n.pCount))$. Die parallele Bearbeitung der Events aus verschiedenen Nodes erfordert die Reservierung eines Bereiches für jede Node. Diese Bereiche lassen sich durch eine Addition aller Events vor einer Node erreichen $(2 \cdot n.pBegin)$.

Der Pseudocode 7 stellt das Clipping und Partitioning der Events dar. Zuerst werden die Fälle untersucht, in welchen das Event vollständig unterhalb bzw. oberhalb des Splits liegt. Hierbei kann das Event ohne eine Modifikation in den jeweiligen Bereich geschrieben werden.

Sollte eine Überlagerung vorliegen, wird im ersten Schritt geprüft, ob sich das Event auf der gleichen Achse befindet wie auch der Split. Wenn dies der Fall ist, kann die Split-Ebene des Events, ohne die Sortierung zu beeinflussen, auf die des Splits gesetzt

Algorithm 7 ClippingAndPartitioning

```
for axis ∈ {X, Y, Z} do
    foreach Event e ∈ events[axis] in parallel do
        s ← splits[n.pBegin]
        if getAxis(e.bbox.max, s.axis) ≤ s.plane then
            addBelow(clippedEvents, e)
        else if getAxis(e.bbox.min, s.axis) ≥ s.plane then
            addAbove(clippedEvents, e)
        else
            if axis = s.axis then
                e.plane ← s.plane
            end if
            if (e.plane < s.plane or (e.plane = s.plane and e.type = END)) then
                setAxis(e.bbox.max, s.axis, s.plane)
            else
                setAxis(e.bbox.min, s.axis, s.plane)
            end if
            addBelow(clippedEvents, e)
            addAbove(clippedEvents, e)
        end if
    end foreach
    PrefixSum(clippedEvents.isValid)
    events[axis] ← Compact(clippedEvents)
end for
```

werden. Split-Ebenen von Events, die sich nicht auf der Split-Achse befinden, werden hierbei nicht modifiziert. Bei dieser Art des Clippings kann es sein, dass die AABB eines Primitives dieses auf den beiden Sekundärachsen nicht mehr zu 100% umschließt (siehe Abb. 4.2).

Der nächsten Schritt prüft, ob die Split-Ebene (*e.plane*) des Events unterhalb bzw. oberhalb des Splits liegt und setzt die entsprechende Plane der AABB des Events auf den des Splits (*setAxis*). Ein Event liegt unterhalb des Splits genau dann, wenn die Split-Ebene des Events kleiner ist als die des Splits (*s.plane*) oder die beiden Ebenen gleich sind (*e.plane=s.plane*) und es sich um ein END-Event handelt. Wenn ein END-Event auf den gewählten Split fällt, wird das Event immer unterhalb des Splits liegen (vgl. Abb. 2.5). Das zur Anwendung kommende konfliktfreie Pattern hat die Eigenschaft, dass die Events ihre Sortierung auf allen Achsen beibehalten (vgl. Abb. 4.3).

In Abbildung 4.3 ist das Clipping beispielhaft für die Primär- und Sekundärachse grafisch dargestellt. Die Situation für die Primärachse (X-Achse) ist im oberen Teil

aufgetragen. Die Primitive und ihre Events werden im Koordinatensystem dargestellt. Durch Pfeile wird symbolisiert, welchen Weg ein Event beim Clippen einschlägt. Pfeile, die den Split (E_0) kreuzen, stellen eine Clipping-Operation dar. Auf der Y-Achse (unten) wird das Clippen eines Events durch das Ausgehen von zwei Pfeilen deutlich gemacht. In diesem Fall sind dies die beiden Events von Primitiv "1". Im rechten Bereich der Abbildung 4.3 sind die beiden Intervalle des Adressraumes abgebildet und die entsprechenden Bewegungen der Events beim Partitioning. Es wird deutlich, dass die Events nach dem Clippen und Partitioning nicht mehr dicht beieinander liegen. Diese Lücken (gekennzeichnet durch Nullen) entstehen durch die Überlagerung des Splits von etwa $\sqrt{n}$ Primitiven im *average case* [28]. Somit ist es nötig, die Events in einem weiteren Schritt zusammenzuschieben, um Berechnungen (wie z. B. die SAH) wieder durchführen zu können.

Zur Beseitigung der entstandenen Lücken innerhalb der Event-Liste kommt eine *Compaction* zum Einsatz. Hierfür muss die *Prefix Sum* der validen Einträge innerhalb der Event-Liste bestimmt werden. Die Berechnung der *Prefix Sum* geschieht über einem Array, in dem alle gültigen Event-Einträge auf 1 und alle ungültigen Event-Einträge (Lücken) auf 0 gesetzt sind. Im Anschluss daran liegen die Events wieder dicht zusammen (siehe Abb. 7.4).

4.3.3 Node Verwaltung

Nach dem Clipping und Partitioning der Events werden die entstehenden Child-Nodes erzeugt und zur *activeNodes* Liste hinzugefügt. Die Split-Plane, die Split-Achse sowie die Adressen der beiden Child-Nodes müssen ebenfalls für die Parent-Nodes gesetzt werden, um auf diese beim späteren Traversieren zugreifen zu können. Entstandene Leaf-Nodes müssen aus der Liste *activeNodes* und deren Events aus der Liste *events* entfernt werden. Alle Nodes, die kein Leaf sind, werden im Folgenden als Interior-Node bezeichnet.

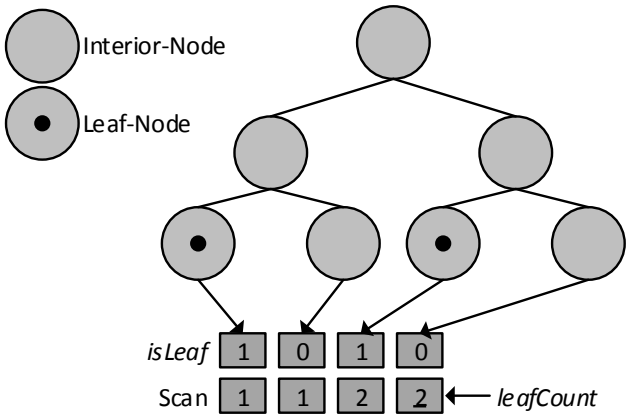

Abbildung 4.4: Anzahl der Leaf-Nodes ergibt sich mittels *Inclusive-Scan*

Interior-Nodes

Die Verarbeitung aller Parent-Nodes kann ohne Konflikte parallel geschehen (vgl. Algorithmus 8). Hierbei wird die berechnete Split-Ebene und die Split-Achse direkt übertragen. Anschließend werden die Nodes von der aktuellen *activeNodes* in die *interiorNodes* linear eingefügt. Über den Index der Node kann die Position innerhalb der Liste bestimmt werden. Durch das lineare Einfügen der Interior-Nodes aus jeder Ebene, werden die Nodes und damit der gesamte Tree in einer Breadth-First Ordnung (vgl. Abbildung 3.1) angelegt. Die *interiorNodes* Liste speichert alle Interior-Nodes des Trees und kann später in dieser Form zum Traversieren des Trees verwendet werden.

Im Anschluss an die Initialisierung der aktiven Nodes werden für jede Node zwei Child-Nodes erzeugt. Die Anzahl der Primitive einer Child-Node lässt sich aus dem Split der dazugehörigen Parent-Node bestimmen (*s.above*, *s.below*). Zusätzlich wird die AABB der Parent-Node mit Hilfe der Split-Plane und Split-Achse des Splits aufgeteilt. Die obere Hälfte wird dem rechten Child und die untere Hälfte dem linken Child zugeschrieben. Die resultierenden AABBs sind für die Berechnung der SAH im nächsten Schritt der Konstruktion wichtig. Es erfolgt an dieser Stelle eine Prüfung,

ob eines der beiden Child-Nodes später zu einem Leaf wird. Eine Child-Node wird genau dann zu einer Leaf-Node, wenn die Anzahl der Primitive einer Child-Node kleiner ist als der vorher festgelegte Wert für die minimale Anzahl an Primitiven innerhalb einer Interior-Node. Falls sich der Algorithmus in der letzten Ebene des Trees befindet, werden alle entstandenen Child-Nodes automatisch zu Leaf-Nodes ($maxDepthReached = true$).

Nach der Initialisierung wird die Liste der Child-Nodes zur aktuellen Liste der aktiven Nodes. Der Offset zu den Primitiven jeder Child-Node (*pBegin*) lässt sich im Anschluss daran durch die Berechnung einer *Prefix Sum* über das Feld *pCount* der entstandenen Child-Nodes berechnen.

Algorithm 8 CreateChildsAndInitParents

```
foreach Node n ∈ activeNodes in parallel do
    s ← splits[n.pBegin]
    n.plane ← s.plane
    n.splitAxis ← s.axis
    Node l, r
    l.pCount ← s.below
    r.pCount ← s.above
    l.isLeaf ← (s.below ≤ maxPrimsPerLeaf) or maxDepthReached
    r.isLeaf ← (s.above ≤ maxPrimsPerLeaf) or maxDepthReached
    splitAABB(n.bbox, s.plane, s.axis, l, r)
    insert(interiorNodesList, n.id]) ← n
    childNodes[2 · n.id] ← c0
    childNodes[2 · n.id + 1] ← c1
end foreach
activeNodes ← childNodes
activeNodes.pBegin ← PrefixSum(activeNodes.pCount)
```

Leaf-Nodes

Bei der Initialisierung der Interior-Nodes wird geprüft, ob eine entstandene Child-Node eine Leaf-Node ist. Leaf-Nodes müssen, bevor es zur Bearbeitung der nächsten Ebene im KD-Tree kommt, aus der Liste *activeNodes* entfernt werden. Außerdem müssen alle Events der Leaf-Nodes aus der Datenstruktur *events* in die Liste *leafPrimitives* übertragen werden, um diese später beim Traversieren zu erreichen.

Für die Trennung zwischen Leaf-Node und Interior-Node wird zunächst eine *Prefix Sum* über das Flag *isLeaf* aller auf dieser Ebene erstellten Child-Nodes generiert (siehe Abb. 4.4). Die Anzahl der Leaf-Nodes findet sich im letzten Eintrag der *Prefix Sum*. Sollte die Berechnung der *Prefix Sum* ergeben, dass eine Verarbeitung von Leaf-Nodes nötig ist, wird im Anschluss daran eine weitere *Prefix Sum* über die Anzahl der Elemente aller Interior-Nodes berechnet. Somit sind aus dieser *Prefix Sum* alle Leaf-Nodes ausgeschlossen (siehe Algorithmus 9).

Im Pseudocode 9 wird zuerst die *activeNodes* List bearbeitet. Wenn die zu verarbeitende Node eine Leaf-Node ist, ($isLeaf \neq 0$) wird ein neues Leaf erstellt, initialisiert und in die Liste aller Leaves (*leaves*) mit aufgenommen. Ein Leaf benötigt zum einen die Information, wo sich die Primitive des Leafs befinden (*pBegin*), zum anderen wie viele Primitive es selbst enthält (*pCount*). Das Feld *pBegin* lässt sich durch die vorher berechnete *Prefix Sum* über die Anzahl der Primitive aller Interior-Nodes dieser Ebene bestimmen. Hierzu wird die *Prefix Sum* vom Feld *pBegin* der Node subtrahiert. Zusätzlich muss die Anzahl aller Primitive aus vorherigen Ebenen (*leafPrimOffset*) hinzuaddiert werden. Das Feld *pCount* kann aus der Interior-Node einfach übertragen werden. Für das spätere Traversieren des Trees ist das Feld *leafId* wichtig, um die Primitive des Leafs erreichen zu können. Zur Bestimmung dieses Feldes wird im Laufe der Konstruktion die aktuelle Anzahl der Leaf-Nodes in einer Variablen gespeichert (*currentLeafCount*) und beim Erstellen der Leaves diese zu der *Prefix Sum* hinzuaddiert (siehe Abb. 4.4). Somit erhält jedes Leaf einen eindeutigen, aufsteigenden Index.

Algorithm 9 MakeLeaves

```
leafScan ← PrefixSumInclusive(activeNodesList.isLeaf)
if leafScan.last ≠ 0 then
    return
end if
intCScan ← PrefixSum(activeNodesList.pCount, isLeaf = 0)
foreach Node n ∈ activeNodes in parallel do
    if n.isLeaf ≠ 0 then
        l.pBegin ← leafPrimOffset + n.pBegin - intCScan[n.id]
        n.leafId ← currentLeafCount + leafScan[n.id]
        l.pCount ← n.pCount
        leaves[currentLeafCount + leafScan[n.id] ← l
    else
        n.pBegin ← intCScan[n.id]
        newActiveNodes[n.id - leafScan[n.id]] ← n
    end if
end foreach
for axis ∈ {0, 1, 3} do
    foreach Event e ∈ events in parallel do
        e.isLeaf ← isLeafAndStartEvent(e)
    end foreach
    eventIsLeafScan ← PrefixSum(events[axis].isLeaf)
    foreach Event e ∈ events in parallel do
        if e.node.isLeaf ≠ 0 and axis = 0 then
            leafPrimitives[leafPrimOffset + eventIsLeafScan[e.id]] ← e.primId
        else if e.isLeaf = 0 then
            interiorEvents[axis][e.id - 2 · (n.pBegin - intCScan[e.node.id]] ← e
            e.node.id ← e.node.id - leafScan[e.node.id]
        end if
    end foreach
    events[axis] ← interiorEvents[axis]
end for
activeNodes ← newActiveNodes
```

Falls es sich um eine Interior-Node handelt (else), muss nur der Offset *pBegin* der Node angepasst werden, da die Events, die zu einem Leaf gehören, aus der Liste *events* entfernt werden. Anschließend wird die Node zurück in eine temporäre Liste (*newActiveNodes*) geschrieben, welche nach dem Verarbeiten aller Nodes zur neuen *activeNodes* Liste wird. Der entsprechende Index in der Liste lässt sich durch die Berechnung der *Prefix Sum* über alle Interior-Nodes bestimmen ($n.id - leafScan[n.id]$).

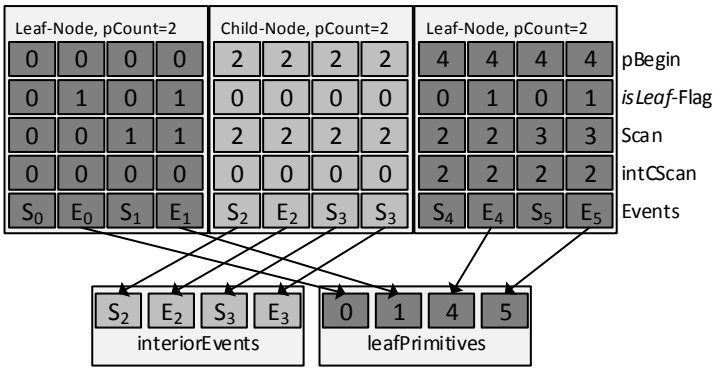

Abbildung 4.5: Zusammenschieben der Event-Liste und das Einfügen der Leaf-Events in die Liste *leafPrimitives*

Nachdem die Daten für jede Node angepasst worden sind, muss dies nun auch für alle Events geschehen. Events, die zu einem entstandenen Leaf gehören, werden aus der Event-Liste entfernt und deren Primitive (*primId*) in die Liste *leafPrimitives* übernommen. In Abbildung 4.5 ist dieser Vorgang beispielhaft für eine Interior-Node und zwei Leaf-Nodes grafisch dargestellt. Die Umsetzung erfolgt durch die Verwendung des in Kapitel 4.2 angesprochenen Flags *isLeaf* eines Events.

Zuerst wird parallel für jedes Event geprüft, ob die dazugehörige Node eine Leaf-Node ist und ob es sich um ein END-Event handelt. Sollte dies der Fall sein, wird das *isLeaf* Flag des Events gesetzt. Durch die Beschränkung auf END-Events wird vermieden, dass beide Events eines Primitives später ihre *primId* in die Liste *leafPrimitives* übertragen.

Im Anschluss daran wird eine *Prefix Sum* über das Flag *isLeaf* berechnet. Wenn das Event zu einem Leaf gehört, muss lediglich der Index des Primitives in die Liste *leafPrimitives* übertragen werden. Die dafür vorgesehene Adresse ergibt sich aus der Anzahl aller eingetragenen Primitive aus vorherigen Ebenen (*leafPrimsOffset*) und der *Prefix Sum* über das *isLeaf* Flag eines Events. Dies muss allerdings nur für eine der drei Achsen der Event-Liste geschehen (z.B. $axis = 0$), denn auf allen Achsen innerhalb einer Node befinden sich immer die gleichen Primitive, deren Events lediglich eine andere Sortierung vorweisen.

Sollte das Event nicht zu einer Leaf-Node gehören, wird dies in die Liste *interiorEvents* übertragen. Für die Bearbeitung der Interior-Nodes wurde bereits eine *Prefix Sum* über das Feld *pCount* berechnet. Hiermit kann festgestellt werden, wie viele Interior-Primitive unterhalb eines Events stehen. Über eine Subtraktion der *Prefix Sum* von dem Feld *pBegin* kann die Anzahl der Leaf-Primitives unterhalb eines Events bestimmt werden. Da zu jedem Primitiv zwei Events existieren, kann durch eine Multiplikation mit dem Faktor 2 auf die Anzahl der Leaf-Events, die unterhalb des Events

stehen, geschlossen werden. Eine Subtraktion dieses Wertes vom aktuellen Index des Events ergibt die Anzahl an Interior-Events unterhalb des Events. Dieser Index ist die neue Position des Events in der Liste *events*.

Zum Schluss muss die Referenz jedes Interior-Events auf seine Node angepasst werden. Hierzu wird vom vorherigen Node-Index des Events die Anzahl aller Leaf-Nodes bis zu diesem Event subtrahiert.

Kapitel 5

Asymptotisches Verhalten

Für eine Komplexitätsanalyse des vorgestellten Algorithmus wird das Modell der *Cuncurrent Read, Exclusive Write Parallel Random Access Machine* (CREW PRAM) zugrunde gelegt [3] [4]. Das Modell beschreibt identische Prozessoren, die gleichzeitig Berechnungen durchführen und sich einen gemeinsamen Speicher teilen, um Ergebnisse auszutauschen. Hierbei wird beim CREW gefordert, dass jeder Prozessor gleichzeitig Lese-, jedoch keine gleichzeitigen Schreibrechte für eine Speicherzelle besitzt. Dieses Modell ist in der Praxis weiter verbreitet, da gleichzeitige Leseoperationen unbedenklich sind, gleichzeitige Schreiboperationen jedoch zu Problemen führen können [4]. Für die Analyse des parallelen Algorithmus werden folgende Begriffe eingeführt:

- **parallel-work** $\mathcal{W}_A(n, p)$ bezeichnet die Anzahl an Operationen aller Prozessoren p bei der Ausführung eines Algorithmus $\mathcal{A}$.
- **parallel-time** $\mathcal{T}_\mathcal{A}(n, p)$ bezeichnet die Anzahl aller parallelen Schritte bei der Ausführung eines Algorithmus $\mathcal{A}$ mit p Prozessoren.
- **work-optimal** ist ein paralleler Algorithmus, wenn er die gleiche Anzahl an Operationen (bis auf einen konstanten Faktor) benötigt wie ein optimaler sequentieller Algorithmus $\mathcal{T}_s(n)$, i.e. $\mathcal{W}_A(n, p) = \mathcal{O}(\mathcal{T}_s(n))$.

Die sequentielle Konstruktion des SAH KD-Trees benötigt nicht mehr als $\mathcal{O}(n \log n)$ Operationen [27]. Wald et al. legen zu Grunde, dass beim Clipping von n Primitiven zwei Listen der Größe $\mathcal{O}(n/2)$ entstehen. Diese Annahme wird im Folgenden übernommen.

Der entwickelte Algorithmus ist **work-optimal**, wenn er bei seiner Ausführung nicht mehr als $\mathcal{O}(n \log n)$ Operationen benötigt. Hierzu bedarf es einer Analyse aller Sub-Algorithmen des Gesamtalgorithmus. Das Ergebnis sowie die Teil-Ergebnisse können in der Tabelle 5.1 abgelesen werden. Für die Analyse wird die Anzahl der zur Verfügung stehenden Prozessoren durch die Variable p abgebildet. Die Anzahl der zu verarbeitenden Daten ist jeweils die Variable n.

In Phase I des Algorithmus werden zuerst die AABBs der Primitive erzeugt, anschließend können die Events erstellt und sortiert werden. Phase I wird, im Gegensatz zu

Phase I	**parallel-time**	**parallel-work**
CreateAABBs	$\mathcal{O}(n/p)$	$\mathcal{O}(n)$
CreateEvents	$\mathcal{O}(n/p)$	$\mathcal{O}(n)$
SortEvents	$\mathcal{O}(n/p)$	$\mathcal{O}(n)$
Phase II ($\log n$ times)		
ComputeBestSplits	$\mathcal{O}(n/p)$	$\mathcal{O}(n)$
ClippingAndPartitioning	$\mathcal{O}(n/p)$	$\mathcal{O}(n)$
CreateChildsAndInitParents	$\mathcal{O}(n/p)$	$\mathcal{O}(n)$
MakeLeaves	$\mathcal{O}(n/p)$	$\mathcal{O}(n)$
Overall	$\mathcal{O}(n/p \log n)$	$\mathcal{O}(n \log n)$

Tabelle 5.1: parallel-time und parallel-work, mit $p = n/\log n$ Prozessoren

Phase II, nur einmal zu Anfang der Konstruktion ausgeführt. Die Berechnung der AABBs kann bei $p < n$ Prozessoren in einer *parallel-time* von $\mathcal{O}(n/p)$ geschehen, wobei jeder Prozessor n/p AABBs berechnet. Für die Berechnung einer AABB wird eine konstante Anzahl an Operationen benötigt, sodass die *parallel-work* $\mathcal{O}(n)$ beträgt. Ebenso können die Events in einer *parallel-work* von $\mathcal{O}(n)$ erstellt werden. Für die Sortierung der Events wird ein paralleler Radix-Sort Algorithmus hinzugezogen [8]. Radix-Sort benötigt unter anderem zwei *Scan* Operationen. Es kann gezeigt werden, dass der *Scan* mit $p < n$ Prozessoren in einer *parallel-time* von $\mathcal{O}(n/p + \log p)$ berechnet wird [3]. Unter der Nebenbedingung, dass $n/p \geqslant \log p$ gilt, ergibt sich $\mathcal{O}(n/p)$. Dies ist optimal in Bezug auf den linearen Algorithmus [3]. Es kann weiterhin gezeigt werden, dass der *Scan* mit einer *parallel-work* von $\mathcal{O}(n)$ möglich ist [3].

Phase II wird im Gegensatz zu Phase I für jedes Level des Trees ausgeführt ($\mathcal{O}(\log n)$ mal). Somit müssen die ermittelten Werte für *parallel-time* und *parallel-work* noch einmal mit dem Faktor $\mathcal{O}(\log n)$ multipliziert werden, um auf die Gesamtwerte zu kommen.

Berechnung bester Splits: Phase II beginnt mit der Berechnung der besten Splits. Hierzu wird für alle möglichen Split-Kandidaten die SAH berechnet. Um eine parallele Berechnung der SAH zu ermöglichen, wird eine *Prefix Sum* über die Event-Typen erstellt. Die *Prefix Sum* kann von p Prozessoren ($p < n$) in einer *parallel-time* von $\mathcal{O}(n/p)$ und einer *parallel-work* von $\mathcal{O}(n)$ berechnet werden [4]. Im Anschluss daran wird eine *Reduction* verwendet, um die besten Split-Kandidaten zu ermitteln. Die *Reduction* kann ebenfalls wie der *Scan* in einer *parallel-time* von $\mathcal{O}(n/p)$ und *parallel-work* von $\mathcal{O}(n)$ ermittelt werden [3]. Die gesamte *parallel-time* ist somit $\mathcal{O}(n/p)$ und die *parallel-work* $\mathcal{O}(n)$ für diesen Sub-Algorithmus.

Clipping und Partitioning: Der nächste Schritt ist das Clippen der Primitive an den gewählten Split-Kandidaten. Hierzu werden die Events zuerst an dem Split geclippt und in die entsprechenden Adressräume geschrieben. Dies kann für p Prozessoren in einer *parallel-time* von $\mathcal{O}(n/p)$ und mit einer *parallel-work* von $\mathcal{O}(n)$ geschehen. Nach der Berechnung der *Prefix Sum* über die validen Einträge, kommt eine *Compaction* zum Einsatz, um die Event-Liste wieder zu einer konsekutiven Liste zusammenzuführen. Eine *Compaction* kann mit p Prozessoren ausgeführt werden, wobei jeder Prozessor n/p Daten verarbeitet. Es ergibt sich somit eine *parallel-time* von $\mathcal{O}(n/p)$ und eine *parallel-work* von $\mathcal{O}(n)$. Der Sub-Algorithmus kann somit in einer *parallel-time* von $\mathcal{O}(n/p)$ und einer *parallel-work* von $\mathcal{O}(n)$ bearbeitet werden.

Node Verwaltung: Der letzte Schritt einer Iteration aus Phase II ist die Erstellung der Child-Nodes und die Verarbeitung entstandener Leaf-Nodes. Hierzu wird über alle Events eine *Prefix Sum* erstellt. Mit Hilfe einer *Compaction* werden die Events, deren Primitive zu einer Leaf-Node gehören, aus der Event-Liste gelöscht und in die Liste der Leaf-Primitive eingefügt. Somit ergeben sich für diesen Sub-Algorithmus eine *parallel-time* von $\mathcal{O}(n/p)$ und eine *parallel-work* von $\mathcal{O}(n)$.

In der Gesamtheit hat der Algorithmus folgende Eigenschaften:

$$\mathcal{W}_A(n,p) = \mathcal{O}(n \log n) \tag{5.1}$$

$$\mathcal{T}_{\mathcal{A}}(n,p) = \mathcal{O}(n/p \log n) \tag{5.2}$$

$$\text{s.t. } n/p \geqslant \log p \tag{5.3}$$

Es konnte gezeigt werden, dass der Algorithmus in der Gesamtheit $\mathcal{O}(n \log n)$ Operationen ausführt und somit **work-optimal** ist. Ein Vergleich mit dem parallelen Algorithmus von Wu et al. ist an dieser Stelle nicht möglich, da die theoretischen Eigenschaften in der Veröffentlichung nicht angegeben sind.

Kapitel 6

Implementationsdetails und Optimierungen

Im folgenden Kapitel wird auf Implementationsdetails des vorgestellten Algorithmus mit Hilfe von CUDA eingegangen. Im Zuge dessen werden Möglichkeiten der Optimierung beschrieben. In diesem Kontext ist mit Optimierung das Beschleunigen und die Anpassung des implementierten Algorithmus für eine GPU gemeint.

Der Algorithmus lässt sich in Sub-Algorithmen unterteilen (vgl. Kapitel 4). Für die Messungen der Implementationen der einzelnen Sub-Algorithmen werden die Testszenen *Bunny*, *Happy* und *Dragon* aus dem Standford 3D Scanning Repository sowie *Angel* aus dem Georgia Tech Large Geometric Model Archive verwendet. Alle Messungen sind unter der Verwendung einer NVIDIA GTX Titan [23] entstanden. Die für diesen Algorithmus speziellen Optimierungen der Standard Algorithmen *Scan* und *Reduction* werden, wenn nicht anders angegeben, zuerst außerhalb des Kontextes des Algorithmus gemessen und dann in ihrer Auswirkung auf den gesamten Algorithmus dargestellt.

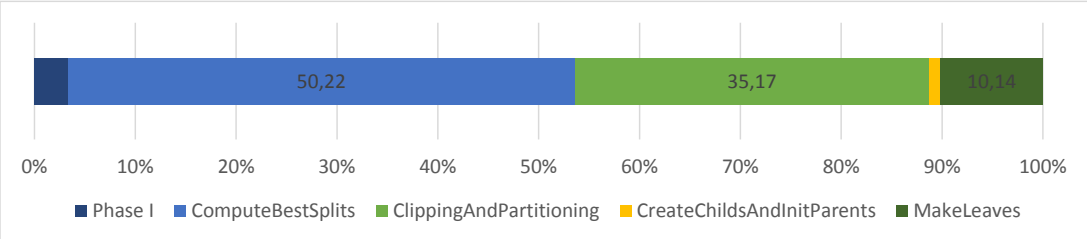

Abbildung 6.1: Laufzeitverteilungen der Sub-Algorithmen vor den Optimierungen

Abbildung 6.1 zeigt die prozentuale Laufzeitverteilung der Sub-Algorithmen für eine frühe, nicht optimierte Version der Implementierung. Deutlich stechen die Berechnungen der besten Split-Kandidaten (*ComputeBestSplits*) und das Clipping und Partitioning der Events (*ClippingAndPartitioning*) hervor.

Im Folgenden werden die Eckpunkte der gesamten Implementierung und vor allem die gezielten Optimierungen der Sub-Algorithmen *ComputeBestSplits* und *ClippingAndPartitioning* vorgenommen.

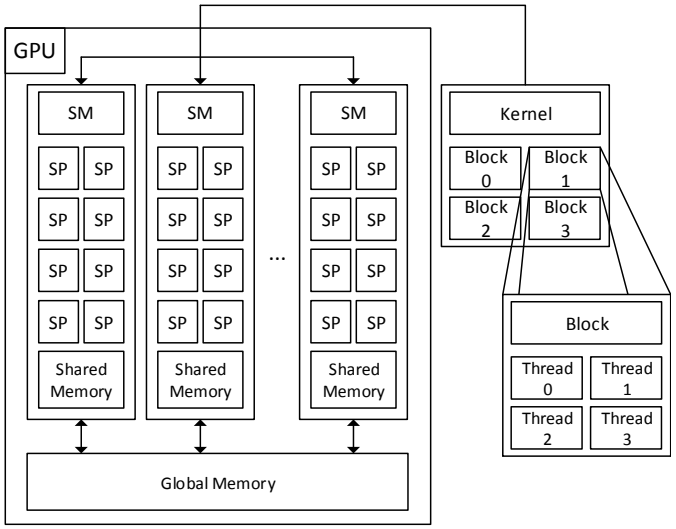

Abbildung 6.2: CUDA Architektur und Programming Model

6.1 CUDA

CUDA (Compute Unified Device Architecture) [21] ist eine parallele Plattform für NVIDIAs Grafikprozessoren, um allgemeine Berechnungen (GPU Computing) parallel auszuführen. CUDA existiert seit 2006 und ist aktuell (August 2014) in der Version 6.5 erhältlich. GPUs wurden bis vor nicht allzu langer Zeit ausschließlich für das Berechnen von Grafikinhalten verwendet. CUDA ist aus der Motivation heraus entstanden, die GPU für allgemeine Berechnungen zu nutzen. Die im Laufe der Arbeit vorgestellten Algorithmen werden deshalb mit Hilfe von CUDA implementiert und für die NVIDIA Kepler Architektur [17] optimiert.

Programming Model

CUDA-Programme werden als Kernel bezeichnet und bestehen aus einem Grid von Thread-Blöcken (Blocks) (vgl. Abb. 6.2). NVIDIA GPUs sind aus einem Array von *Streaming-Multiprozessoren* (SM) zusammengesetzt. Ein SM besteht wiederum aus mehreren *Streaming-Processoren* (SP), die darauf ausgelegt sind, hunderte Threads auszuführen [21]. Blöcke eines Kernels werden bei der Ausführung an einen freien SM verteilt. Nur Threads innerhalb eines Blocks können über spezielle Funktionen synchronisiert werden. Ein SM unterteilt einen Block vor der Ausführung in sogenannte Warps. Ein Warp besteht auf NVIDIA GPUs derzeit aus 32 Threads.

CUDA C

Das Programming Model von CUDA wird über die von NVIDIA eigens entwickelte Sprache CUDA C angesprochen. Entwickler können dadurch Programme formulie-

ren, die dann auf der GPU ausgeführt werden. Die Sprache CUDA C ist eine Erweiterung der Sprache C/C++ und bietet einige spezielle Features, um NVIDIA GPUs zu programmieren. Durch das Schlüsselwort ___*global*___ werden in CUDA C Kernel definiert, die vom Host (CPU) aus gestartet und auf dem Device (GPU) ausgeführt werden. Der Aufruf eines Kernels erfolgt, indem über die spezielle Syntax (<<< ..., ... >>>) die Parallelität des Kernels (Anzahl der Blöcke und Anzahl der Threads pro Block) festgelegt wird [20]. Innerhalb eines Kernels kann auf weitere Built-In Variablen zurückgegriffen werden, um z. B. den Index des Threads (*threadIdx*) aus einem Block auszulesen und somit die diesem Thread *zugeordneten* Daten zu laden und zu verarbeiten. Anhand eines minimalen Beispiels der parallelen Vektoraddition ist die Formulierung eines Kernels dargestellt (siehe Codebeispiel 6.1).

Listing 6.1: Parallele Vektoraddition mit CUDA [20]

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N) {
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   C[i] = A[i] + B[i];
}
// Host code
int main(void) {
    int N = ...;
    size_t size = N * sizeof(float);
    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    // Initialize input vectors ...
    // Allocate vectors in device memory
    float* d_A; cudaMalloc(&d_A, size);
    float* d_B; cudaMalloc(&d_B, size);
    float* d_C; cudaMalloc(&d_C, size);
    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    // Free host memory ...
}
```

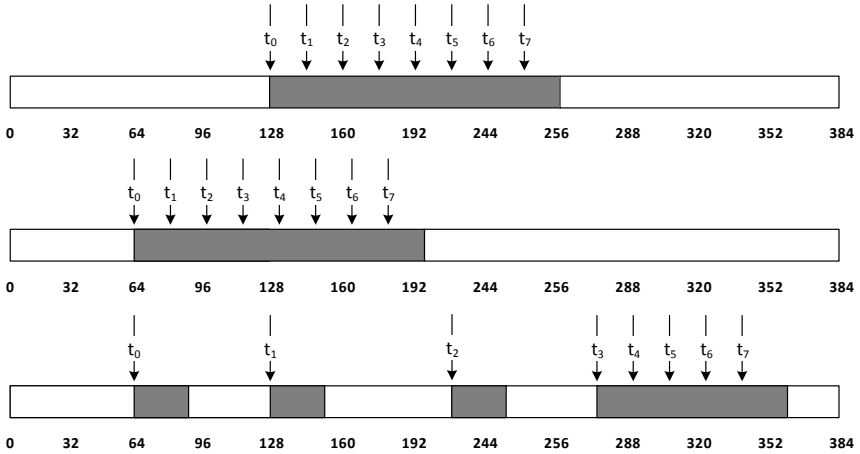

Abbildung 6.3: Speicherzugriffe: *coalesced* und *aligned* (oben), nicht *aligned* (mittig), weder *coalesced* noch *aligned* (unten)

Host und Device

Um eine simultane Ausführung von Host und Device zu gewährleisten, sind fast alle CUDA Funktionen asynchron aus der Sicht des Hosts. Die aufgerufenen Funktionen geben die Kontrolle fast immer unmittelbar an den Host zurück, bevor das Device diese abgearbeitet hat. Für eine explizite Synchronisation zwischen Device und Host werden Funktionen zur Verfügung gestellt. Speziell für den Austausch von Daten zwischen Host und Device sind bestimmte Techniken nötig, um globale Synchronisationen zu vermeiden. Generell ist darauf zu achten, so wenig wie möglich zwischen Host und Device zu synchronisieren [21].

Globale Speicherzugriffe

Eine der wichtigsten Bedingungen für die maximale Performance eines Kernels ist das *coalesced* (vereinigte) Lesen von globalem Speicher [20]. Speicher-Transaktionen in einem Warp werden vom Device, falls möglich, *coalesced* durchgeführt. Das bedeutet, mehrere Anfragen werden zu einer Anfrage zusammengefasst. Eine *coalesced* Transaktion kann immer dann stattfinden, wenn der k-te Thread das k-te Wort in einer Cache-Line anfordert [20]. In Abbildung 6.7 ist dies schematisch dargestellt. Die Größe einer Cache-Line beträgt hier 128 Byte und ein Warp besteht beispielhaft aus 8 Threads. In Abbildung 6.7 ist oben die optimale Situation dargestellt: Alle Threads fordern konsekutiv liegende Daten aus einer Cache-Line an. In der Mitte sind die Speicheradressen zwar konsekutiv, jedoch erstrecken sie sich über zwei Cache-Lines, wodurch es zu zwei Transaktionen kommt. Um dies zu vermeiden, sollten zusätzlich die Anfragen eines Warps immer mit der jeweiligen Größe einer Cache-Line *aligned* sein. Das untere Beispiel zeigt einen nicht *coalesced* und nicht *aligned* Zugriff, welchen es in jedem Fall zu vermeiden gilt. Der Einfluss von nicht *aligned* und nicht *coalesced*

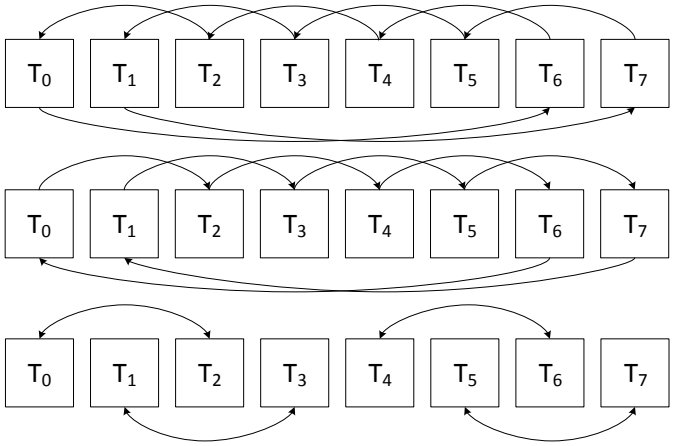

Abbildung 6.4: __shfl: down (oben), up (mittig) und xor (unten), warpSize=8

Speicherzugriffen wird im *CUDA C Best Practices Guide* im Kapitel 9.2.1 [19] noch einmal grafisch dargestellt.

Shared-Memory

CUDA stellt dem Entwickler einen selbst zu verwaltenden (relativ kleinen) Cache, das sogenannte Shared-Memory, pro Block zur Verfügung. Das Shared-Memory befindet sich direkt *on-chip* und hat dementsprechend eine sehr geringe Latenz. Somit ist das Shared-Memory für das Cachen globaler Daten geeignet, um diese innerhalb eines Blocks schnell mit anderen Threads auszutauschen, ohne den langsameren Weg über den globalen Speicher gehen zu müssen. Shared-Memory ist logisch in gleichgroße Blöcke (32 Bit) eingeteilt, sogenannte Banks (32 Banks auf aktuellen GPUs). Es gilt zu beachten, dass Threads aus einem Warp nicht gleichzeitig auf dieselbe Bank zugreifen. Speicherzugriffe in dieselbe Bank (Bank-Conflicts) müssen von der Hardware serialisiert werden und können die Performance verringern.

Warp-Vote Funktionen

Mit der Einführung der Kepler Architektur [17] wird CUDA um die Möglichkeit erweitert, Daten innerhalb eines Warps direkt und ohne den Weg über das Shared-Memory auszutauschen. Dies kann zu einer Entlastung des begrenzten Shared-Memory beitragen. Zusätzlich sind Warp-Vote Funktionen schneller als Shared-Memory Zugriffe. Der Zugriff auf das Shared-Memory benötigt drei Instruktionen: Schreiben, Synchronisieren und Lesen. Shuffle-Funktionen hingegen benötigen lediglich eine Instruktion [5]. Der Austausch von Daten zwischen Threads innerhalb eines Warps geschieht mit Hilfe der Warp-Shuffle Funktion, welche in mehreren Versionen zur Verfügung steht. Ein Thread kann darüber ein Register eines beliebigen anderen Threads innerhalb des Warps auslesen.

```
1 int __shfl(int var, int lane, int width=warpSize);
2 int __shfl_up(int var, unsigned int delta, int width=warpSize);
3 int __shfl_down(int var, unsigned int delta, int width=warpSize);
4 int __shfl_xor(int var, int lane, int width=warpSize);
```

Listing 6.2: Warp Shuffle-Funktionen Definitionen [20]

Gängige Lesemuster werden in den Funktionen *up down* und *xor* zusammengefasst. Shuffle-Funktionen bekommen entweder direkt den Index des Threads, von dem sie ein Register lesen (*lane*) oder eine Sprungweite (*delta*), woraus sich der Index relativ zu ihrem eigenen Index ergibt (vgl. Abb. 6.4).

6.2 Data-Management

In der Implementation des Algorithmus werden Datensätze, die eine Vielzahl von Feldern beinhalten (z. B. Node-Data und Split-Data) über *Structures of Arrays* (SoA) anstelle von *Array of Structures* (AoS) realisiert (vgl. Abb. 6.5). Auf diese Weise wird eine optimale Ausnutzung des GPU Caches ermöglicht [28]. Bei der Verwendung von AoS liegen die Daten eines Structs immer konsekutiv im Speicher und werden bei jeder Anfrage eines Datensatzes vollständig geladen. Dies ist aber nur dann effizient, wenn auch alle Datenfelder benötigt werden. Bei der Konstruktion des KD-Trees werden nicht immer alle Daten in jedem Kernel gebraucht, wodurch sich die Verwendung von AoS als ungünstig erweist.

Der Algorithmus allokiert initial eine von der Anzahl der Primitive abhängige Menge an Speicher für die benötigten Datensätze. Sollte dieser Speicher im Laufe der Konstruktion zu klein werden, wird die Speichermenge jeweils verdoppelt.

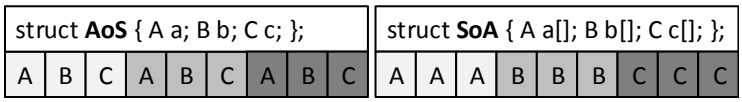

Abbildung 6.5: AoS vs. SoA

6.3 Event-Sorting

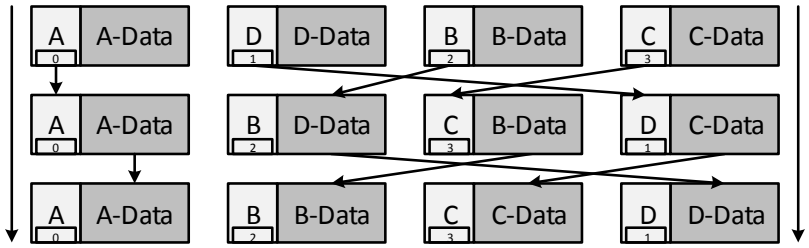

Abbildung 6.6: Indexed-Sorting

Da die Implementation mit AoS arbeitet, muss das Sortieren der Events mit einem Schlüssel-Wert Paar geschehen (siehe Abb. 6.6). Nach dem Sortieren der Events anhand des numerischen Wertes auf der Achse, wird über den mitsortierten Index auf die nicht mitsortierten Daten zurückgegriffen. Für die Sortierung der Events wird eine parallele Radix-Sort [8] Implementation aus der Bibliothek *thrust* [11] verwendet.

6.4 Optimierung globaler Speicherzugriffe

Anhand der Berechnung der AABBs der Primitive wird nun die Implementation mit optimierten globalen Speicherzugriffen vorgestellt. Diese Technik lässt sich auf viele Kernels innerhalb des implementierten Algorithmus übertragen und wird dort angewandt.

Für die Generierung der AABBs wird jeweils pro AABB ein Thread gestartet, der dann auf die drei Eckpunkte des Dreiecks zugreift, die AABB erstellt und diese dann zurück in den globalen Speicher schreibt. Eine naive Implementation in CUDA ist im Quelltext 6.3 dargestellt.

Hierbei ergibt sich ein Problem: Die Speicherzugriffe beim Laden der drei Punkte A, B und C sind nicht *coalesced*, denn es werden keine konsekutiven Daten geladen. Jede Anfrage eines Threads überspringt 24 Byte.

Listing 6.3: Naive Berechnung der AABBs

```
struct AABB {
    float3 _min; float3 _max;
    void AddVertex(float3 v) {
        _min = min(_min, v);
        _max = max(_max, v);
    }
    void Reset(void) {
        _min = FLT_MAX3;
        _max = -FLT_MAX3;
    }
};
__global__ void createAABBsNaive(float3* tris, AABB* aabbs) {
    uint idx = blockIdx.x * blockDim.x + threadIdx.x;
    float3 A = tris[3 * idx + 0];
    float3 B = tris[3 * idx + 1];
    float3 C = tris[3 * idx + 2];
    AABB bb;
    bb.Reset();
    bb.AddVertex(A); bb.AddVertex(B);  bb.AddVertex(C);
    aabbs[idx] = bb;
}
```

Eine für CUDA optimierte Version ist im Quelltext 6.4 dargestellt. Im Gegensatz zur naiven Variante werden hier die Dreiecke als *float* Array interpretiert. Jetzt laden alle Threads konsekutiv Daten in 32 Bit Paketen in das Shared-Memory (*coalesced*). Jeder Thread eines Blocks berechnet die AABB eines Dreiecks, folglich müssen pro Block $3 \cdot blockSize$ Dreiecke geladen werden. Hierdurch ergibt sich eine Shared-Memory Größe von $9 \cdot blockSize \cdot 4$ Byte, denn jedes Dreieck besteht aus drei *float3* Datensätzen. Um alle Dreiecke in das Shared-Memory zu laden, muss dementsprechend jeder Thread 9 *float* Werte anfordern. Dies geschieht mit Hilfe einer for-Schleife. Die Adresse pro Thread wird nach jedem Durchlauf um *blockSize* erhöht, um das konsekutive Laden der Daten sicherzustellen.

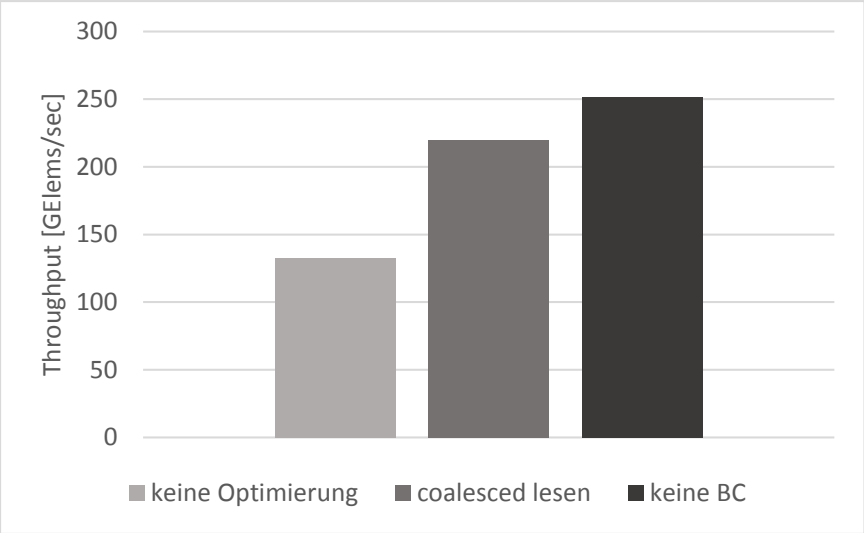

Abbildung 6.7: Optimierung von *createAABBs*

Listing 6.4: Berechnung der AABBs mit Shared-Memory

```
template<int blockSize>
__global__ void createAABBs(float* triangles, AABB* aabbs, uint N) {
    __shared__ float s_data[9 * blockSize];
    uint offset = blockIdx.x * blockDim.x * 9;
    for(int i = 0; i < 9; ++i) {
       s_data[threadIdx.x + i * blockSize] = triangles[offset + threadIdx.x + i * blockSize];
    }
    __syncthreads();
    float3 A = ((float3*)s_data)[3 * threadIdx.x + 0];
    float3 B = ((float3*)s_data)[3 * threadIdx.x + 1];
    float3 C = ((float3*)s_data)[3 * threadIdx.x + 2];
    AABB bb;
    bb.Reset();
    bb.AddVertex(A); bb.AddVertex(B); bb.AddVertex(C);
    aabbs[blockIdx.x * blockDim.x + threadIdx.x] = bb;
}
```

Im Anschluss daran muss innerhalb des Blocks synchronisiert (*__syncthreads*) werden, um zu garantieren, dass alle Threads die Daten in das Shared-Memory geladen haben. Als nächstes lädt jeder Thread seine drei Punkte A, B und C aus dem Shared-Memory. Dies erfolgt über das bereits in der naiven Version genutzte Zugriffsmuster.

An dieser Stelle wurde das nicht *coalesced* Lesen der naiven Variante behoben. Durch die Verwendung von Shared-Memory ergibt sich jedoch jetzt ein neues Problem: Bank-Conflicts.

Bank-Conflicts

Wie bereits im Kapitel zu Shared-Memory erwähnt, erfolgt der Zugriff auf das Shared-Memory über Banks. Wenn mehrere Threads aus einem Warp in die gleiche Bank gleichzeitig zugreifen, müssen die Zugriffe synchronisiert werden. In der vorgestellten Implementation liest jeder Thread immer drei konsekutive 32 Bit Werte aus dem Shared-Memory. Infolgedessen wird jede Bank genau dreimal gleichzeitig von unterschiedlichen Threads gelesen ($3 \cdot 32\ Werte\ /\ 32\ Banks$).

Listing 6.5: Berechnung der AABBs, keine Bank-Conflicts

```
__global__ void createAABBs(float* triangles, float* aabbs, uint N) {
  ...
    const uint stride = 9;
    A.x = s_data[stride * threadIdx.x + 0];
    A.y = s_data[stride * threadIdx.x + 1];
    A.z = s_data[stride * threadIdx.x + 2];
    B.x = s_data[stride * threadIdx.x + 3];
    B.y = s_data[stride * threadIdx.x + 4];
    B.z = s_data[stride * threadIdx.x + 5];
    C.x = s_data[stride * threadIdx.x + 6];
    C.y = s_data[stride * threadIdx.x + 7];
    C.z = s_data[stride * threadIdx.x + 8];
    ...
}
```

Zur Vermeidung der Bank-Conflicts findet eine gängige Technik Anwendung: Pro Thread werden 32 Bit Worte mit einer Schrittgröße (stride) aus dem Shared-Memory geladen, die kein ganzzahliger Teiler der Anzahl der Banks (32) ist (siehe Codebeispiel 6.5).

Abbildung 6.7 zeigt den Performancegewinn durch *coalesced* Speicherzugriffe und zusätzlich durch die Vermeidung von Bank-Conflicts.

6.5 Algorithm-Cascading

Algorithm-Cascading beschreibt die Aufteilung eines parallelen Algorithmus in parallele und sequentielle Teile. Dies kann insbesondere dann von Vorteil sein, wenn ein paralleler Algorithmus in seiner maximalen Parallelität nur eine geringe Anzahl an Operationen pro Thread generiert. Hierbei wird die Implementierung eines Algorithmus so angepasst, dass von jedem Thread mehr als ein Element verarbeitet wird. Der Einfluss ist insbesondere auf modernen GPUs zu erkennen, wie anhand einer *Cascaded-Reduction* zu sehen ist (siehe Abb. 6.8). Der Throughput steigt zuerst mit einer Erhöhung der zu verarbeitenden Elemente pro Thread an. Ab einer Anzahl von etwa 32K Elementen lässt die Performance jedoch nach, da die Anzahl an sequentiellen Operationen zu groß wird und zusätzlich die Anzahl der Threads immer geringer wird.

In der Praxis wird dieses Verhalten vor allem durch spezielle Hardware Details beeinflusst. Der höchste Throughput der *Cascaded-Reduction* wird in diesem Beispiel mit einer Anzahl von 64 Elementen (32 Bit) pro Thread erreicht (16384 Elemente per Block), wobei insgesamt 16 Mio. Elemente verarbeitet werden.

Algorithm-Cascading kommt deshalb auch in der Implementation des Algorithmus zum Einsatz. Beispiele der Implementierung werden im Folgenden anhand der angepassten *Reduction* und des *Scan* gegeben.

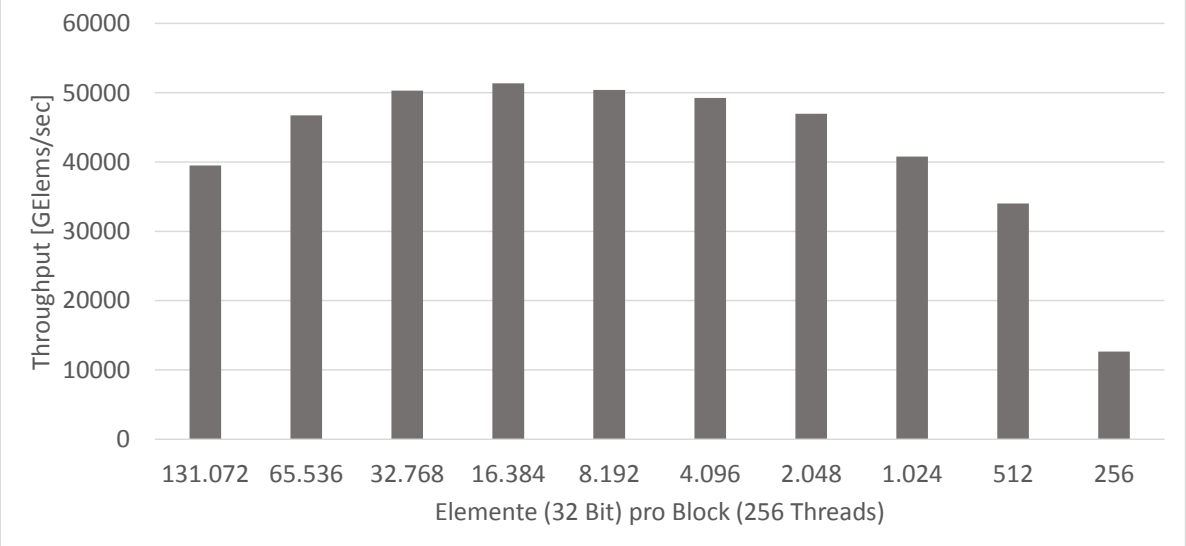

Abbildung 6.8: Cascaded *Reduction*

6.6 Streams

Wie bereits im Abschnitt 6.1 erwähnt, arbeiten der Host und das Device asynchron zueinander. Über Streams ergibt sich die Möglichkeit in CUDA, Kernel auf dem Device auszuführen und gleichzeitig Kopiervorgänge zwischen dem Host und dem Device zu koordinieren [20]. Um Daten in einer Anwendung effizient zwischen Host und Device auszutauschen, ohne eine Synchronisation zwischen beiden zu bewirken, ist die Verwendung von Streams unerlässlich.

Ein Stream bildet eine Liste von Kommandos ab, die in einer FIFO-Reihenfolge abgearbeitet werden. Um einen asynchronen Kopiervorgang vom Device zum Host herzustellen, werden zwei Streams benötigt. Ein Stream S_1 für die Ausführung des Kernels und ein Stream S_2 für die Ausführung des Kopiervorgangs. Sobald der Kernel abgearbeitet worden ist, liegen die Ergebnisse im Speicher des Devices und ab diesem Zeitpunkt startet die Kopie zum Host. Realisiert wird die Synchronisation der Streams über ein CUDA-Event, welches nach Abarbeitung des Kernels vom Stream S_1 geworfen wird. Stream S_2 wartet auf dieses Event und fängt unmittelbar mit der Abarbeitung an (vgl. Abb. 6.9). Zwischenzeitlich kann Stream S_1 seine Arbeit fortsetzen (soweit Arbeit zur Verfügung steht) und wird nicht über den Kopiervorgang blockiert. Nur wenn der Kopiervorgang durch weitere Kernels überlagert werden kann, macht die Verwendung von Streams Sinn.

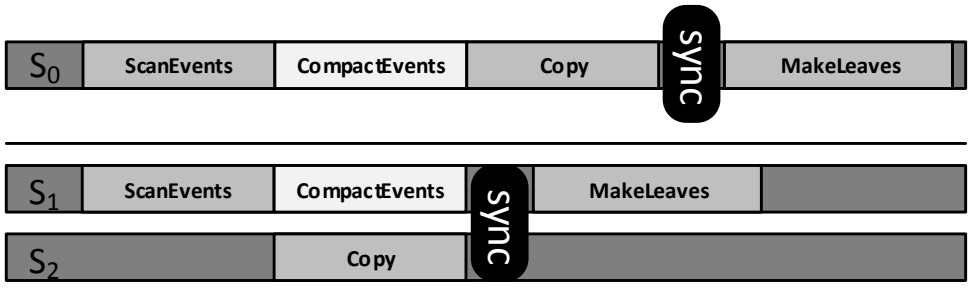

Abbildung 6.9: ein Stream (oben) vs. zwei Streams (unten)

In der Konstruktionsphase des Algorithmus muss die Anzahl der nach einem Schritt vorhandenen aktiven Nodes und die Anzahl der durch das Clipping entstandenen Events vom Device zum Host kopiert werden, um im nächsten Schritt die Parallelitäten der Kernel bestimmen zu können. Diese Kopiervorgänge lassen sich mit Hilfe von Streams optimieren.

Die Anzahl der Events steht nach der Clipping Operation mit der Generierung der *Prefix Sum* für die *Compaction* der Events fest. Benötigt wird die Anzahl erst wieder bei der Verarbeitung der Leaf-Nodes bzw. falls keine existieren, bei der Berechnung der SAH im nächsten Schritt. Die *Compaction* der Events kann somit dazu genutzt werden, den Kopiervorgang zu überlagern (vgl. Abb. 6.9). Ebenso ist die Anzahl der

aktiven Nodes für den nächsten Schritt nach der Berechnung der *Prefix Sum* über das *isLeaf* Flag der Child-Nodes bekannt. Auch dieser Kopiervorgang lässt sich mit der Verarbeitung der Leaf- und Interior-Nodes (Node Verwaltung) überlagern.

6.7 Binary-Scan (Prefix Sum)

Die Berechnung einer *Prefix Sum* wird an drei Stellen im Algorithmus benötigt: Bei der Bestimmung der besten Split-Kandidaten, beim Clipping und Partitioning der Events und bei der Generierung der Leaf-Nodes. In allen Fällen ist die Ausgangssituation ein binäres Array $(0, 1)$, wodurch sich als Datentyp ein Byte pro Eintrag anbietet. Speicherzugriffe sind aber nur dann *coalesced*, wenn jeder Thread konsekutiven Speicher anfordert und die Größe ein gerades Vielfaches eines 32 Bit Wortes ist [20]. Da ein Byte jedoch aus 8 Bit besteht, ist eine naive Implementierung der *Prefix Sum* (jeder Thread liest ein Byte) nicht *coalesced*. Bei der Verwendung eines 32 Bit Datentyps pro Eintrag sind die Zugriffe wieder *coalesced*, jedoch werden drei mal soviel Daten wie nötig geladen.

Im Folgenden wird die Berechnung einer *Prefix Sum* auf binären Daten entwickelt, die das *coalesced* Lesen sicherstellt und zusätzlich die Shuffle-Funktion verwendet.

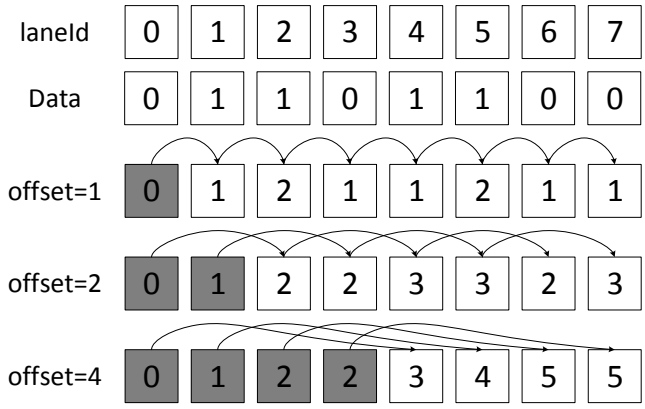

Abbildung 6.10: Warp-Prefix Sum mit shfl_up

Die Implementation der *Prefix Sum* mit Hilfe von *___shfl_up* gliedert sich in zwei Phasen: Zuerst berechnet jeder Warp die *Prefix Sum* auf seinen 32 Werten. Danach kümmert sich der Warp mit dem Index 0 (*warpId==0*) um die Berechnung der *Prefix Sum* über alle Summen der Warps. Die Zwischensummen werden deshalb zuvor von jedem Warp im Shared-Memory hinterlegt. Im Anschluss daran wird die *Prefix Sum* der Zwischensummen wieder in das Shared-Memory zurück geschrieben. Danach kann sich jeder Thread eines Warps die berechnete *Prefix Sum* bis zu dem Index seines

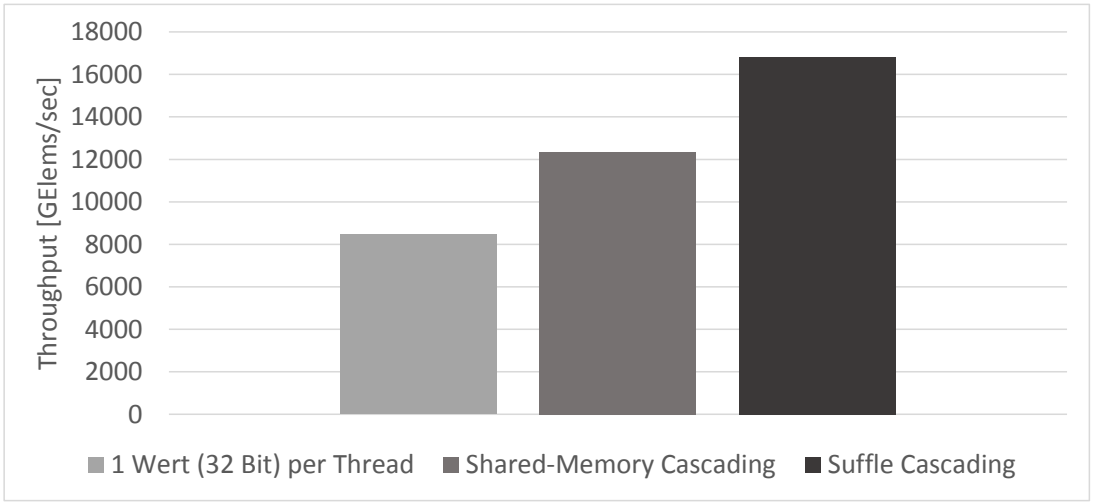

Abbildung 6.11: *Scan* Implementationen im Vergleich

Warps aus dem Shared-Memory auslesen und auf seine lokale *Prefix Sum* addieren.

Die Implementation einer *Prefix Sum* pro Warp ist in Codeabschnitt 6.6 abgebildet. Hierzu wird in $\log(warpSize)$ Schritten jeweils ein Wert um $\log(warpSize)$ Positionen bewegt (vgl. Abb. 6.10).

Für die Sicherstellung der *coalesced* Speicherzugriffe verarbeitet jeder Thread vier Eintrage aus dem binären Array (32 Bit).

Zusätzlich kommt es zur weiteren Anwendung von Algorithm-Cascading (siehe Kapitel 6.5). Jeder Block verarbeitet N Einträge, die ein Vielfaches seiner Größe sind.

Die vollständige Implementation der *Prefix Sum* ist im Codebeispiel 6.6 dargestellt. Von einem Thread werden immer Daten vom Typ *uchar4* (4 Einträge) gelesen. Nach dem Laden eines Datensatzes werden zwei Partialsummen bestimmt (*partSum1* / *partSum2*). Der erste Summand (*partSum1*) speichert die Summe der ersten zwei Bytes und der zweite Summand (*partSum2*) speichert die Summe der ersten drei Bytes. Diese Summen werden am Schluss für die korrekte Konstruktion der *Prefix Sum* benötigt, da die *Prefix Sum* für jeden Block über die Summen aller Einsen aus den geladenen 4 Werten berechnet wird (siehe Codebeispiel 6.6).

```
__device__ int warp_shfl_ps(int value) { // PrefixSum per Warp
    int warpId = threadIdx.x / warpSize;
    int laneId = threadIdx.x % warpSize;
    for(int offset = 1; offset < warpSize; offset <<= 1) {
        int n = __shfl_up(value, offset, width);
        if(laneId >= offset) value += n;
    }
    return value;
}
__device__ int shuffleBlockScan(uint* sdata, int value) { //PrefixSum per Block
    int warpPrefix = warp_shfl_ps(x);
    int warpIdx = threadIdx.x / warpSize;
    int laneIdx = threadIdx.x & (warpSize-1);
    if(laneIdx == warpSize-1) { sdata[warpIdx] = warpPrefix; }
    __syncthreads();
    if(threadIdx.x < warpSize) { sdata[idx] = warp_shfl_ps(sdata[idx]); }
    __syncthreads();
    return sdata[warpIdx] + warpPrefix - x;
}
__global__ void binaryGroupScanN(const uchar4* g_data, uint4* scanned, uint* sums, uint N) {
    __shared__ uint shrdMem[blockSize/32];
    uint globalOffset = blockIdx.x * N;
    uint allSum = 0;
    for(int i = 0; i < N; i += blockDim.x) { //cascading
        uint ai = i + threadIdx.x;
        uchar4 a = g_data[globalOffset + ai];
        uint partSum1 = a.y + a.x;
        uint partSum2 = a.z + partSumA1;
        uint sum = shuffleBlockScan(shrdMem, partSum2 + a.w);
        scanned[globalOffset + ai].x = sum;
        scanned[globalOffset + ai].y = sum + a.x;
        scanned[globalOffset + ai].z = sum + partSum1;
        scanned[globalOffset + ai].w = sum + partSum2;
        allSum += shrdMem[blockDim.x/32-1];
    }
    if(threadIdx.x == blockDim.x-1) { sums[blockIdx.x] = allSum; }
}
```

Listing 6.6: Cascaded binary *Prefix Sum*, (Basis: [18])

Im Diagramm 6.11 ist der Throughput der drei möglichen *Prefix Sum* Implementationen aufgetragen. Die beste Performance kann unter Verwendung der Shuffle-Funktion und mit der Verarbeitung von 32 *uchar4*-Werten pro Thread (Cascading) erreicht werden. Die Shared-Memory Variante basiert auf der *Prefix Sum* Implementation aus GPU Gems, welche in dieser Implementation 4 Werte pro Thread verarbeitet [24].

6.8 Berechnung der besten Splits

Im Folgenden wird auf die Implementation und Optimierung des Sub-Algorithmus *"ComputeBestSplits"* eingegangen.

6.8.1 SAH Shuffle-Reduction

Für die Bestimmung der besten Split-Kandidaten wird eine *Reduction* über alle Split-Kandidaten einer Node verwendet. Ein Split-Kandidat kann durch die Datenstruktur *SAHSplit* (vgl. Codeabschnitt 6.7) abgebildet werden. Dieser Datentyp speichert den berechneten SAH-Wert und zusätzlich einen Index. Der Index wird benötigt, um nach der *Reduction* wieder an die Informationen des Split-Kandidaten zu gelangen.

Für die *Reduction* eines *SAHSplit* müssen zwei 32 Bit Variablen übertragen werden. Die zuvor vorgestellten Shuffle-Funktionen sind aber lediglich für 32 Bit Datentypen ausgelegt. Durch die zweimalige Anwendung der Shuffle-Funktion lässt sich dieses Problem jedoch lösen. Die Funktion *___shfl64_down* bekommt eine 64 Bit Variable, verarbeitet erst die unteren 32 Bit und im Anschluss daran die oberen 32 Bit (vgl. Codeabschnitt 6.7). Das Ergebnis wird mit dem aktuellen, besten Split-Kandidaten verglichen und gegebenenfalls gesetzt.

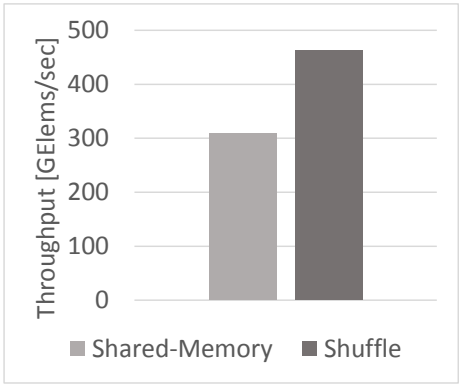

Abbildung 6.12: Shared-Memory vs. Shuffle-*Reduction*

Nachdem jeder Warp seinen Split-Kandidaten bestimmt hat, muss in einem letzten Schritt noch einmal über die Ergebnisse jedes Warps eine *Reduction* durchgeführt werden. Das Ergebnis eines Warps wird dazu im vorher allokierten Shared-Memory hinterlegt. Das Shared-Memory muss so viele Werte speichern, wie Warps in einem Block existieren. Für die letzte *Reduction* wird der Warp mit dem Index 0 (*warpId==0*) gewählt. Die Blockgröße ist bei dieser Implementation somit auf 1024 beschränkt, was aber in der Praxis keine Auswirkung hat, da die Blockgröße in der Regel kleiner als 1024 gewählt wird.

Der Performancegewinn durch die Verwendung der Shuffle-Funktion, im Vergleich zu einer Shared-Memory Implementation auf dem gleichen System, lässt sich im Diagramm 6.12 ablesen.

```
struct SAHSplit {
    float sah;
    uint index;
};
__device__ double __shfl64_down(double var, unsigned int lane) {
    int2 a = *reinterpret_cast<int2*>(&var);
    a.x = __shfl_down(a.x, srcLane, warpSize);
    a.y = __shfl_down(a.y, srcLane, warpSize);
    return *reinterpret_cast<double*>(&a);
}
__device__ SAHSplit warpReduceSAHSplit(SAHSplit val) {
    double d; SAHSplit _v;
    for(int offset = warpSize/2; offset > 0; offset >>= 1) {
        d = __shfl64_down(*((double*)&( val.sah)), offset);
        memcpy(&_v, &d, sizeof(double));
        val = min(_v, val);
    }
    return val;
}
template<uint WARP_COUNT>
__device__  IndexedSAHSplit blockReduceSAHSplit(SAHSplit val) {
    __shared__ SAHSplit shared[WARP_COUNT];
    int laneId = threadIdx.x % warpSize;
    int warpId = threadIdx.x / warpSize;
    val = warpReduceSAHSplit(val);
    if(laneId == 0) shared[warpId] = val;
    __syncthreads();
    val = (threadIdx.x < blockDim.x / warpSize) ? shared[laneId] : 0;
    if(warpId == 0) val = warpReduceSAHSplit(val);
    return val;
}
```

Listing 6.7: 64 Bit SAH Block-*Reduction* (Basis: [22])

6.8.2 Reduction vs. Segmented-Reduction

Wie zuvor beschrieben, muss die *Reduction* über alle Split-Kandidaten für jede Node separat berechnet werden, wodurch sich die Verwendung einer im Kapitel 2.4 angesprochenen *Segmented-Reduction* anbietet. Die Implementation der *Segmented-Reduction* ist im Quelltext 6.8 dargestellt. Demnach ist ein Block für die Berechnung einer *Reduction* über ein Segment (Node) zuständig.

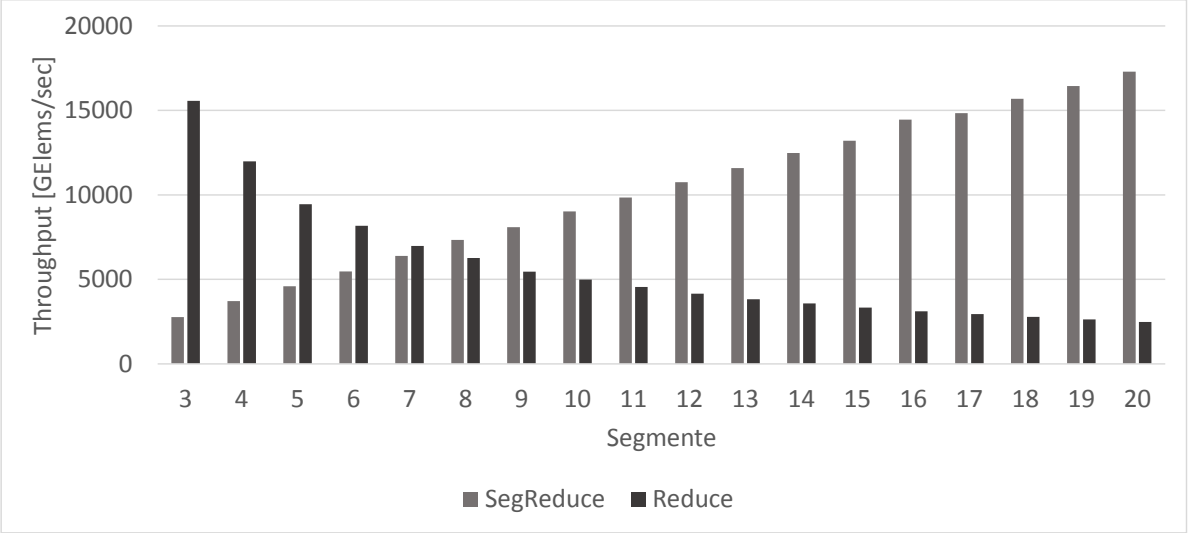

Abbildung 6.13: *Segmented-Reduction* vs. *Reduction*

Zu Beginn der Konstruktionsphase ist die Anzahl der aktiven Nodes jedoch gering, wodurch nur sehr wenige Blöcke ausgeführt werden. Die Anzahl der Events pro Node ist in den ersten Ebenen des Trees sehr hoch. Somit ergeben sich bei einer durchgängigen Verwendung der *Segmented-Reduction* gerade in den ersten Ebenen Performance Probleme, da eine geringe Anzahl an Threads sehr viele Daten sequentiell verarbeitet (vgl. Quelltext 6.8).

```
template <uint blockSize>
__global__ void segReduce(const SAHSplit* splits, Nodes nodes) {
    uint segOffset = 2 * nodes.pBegin[blockIdx.x];
    uint segLength = 2 * nodes.pCount[blockIdx.x];
    uint tid = threadIdx.x;
    SAHSplit split;
    split.index = 0;
    split.sah = FLT_MAX;
    while(tid < segLength) {
        split = minSAH(split, minSAH(splits[segOffset + tid], splits[segOffset + tid + blockSize]));
        tid += 2 * blockSize;
    }
    split = blockReduceSAHSplit<blockSize/32>(split);
    if(threadIdx.x == 0) splits[segOffset] = split;
}
```

Listing 6.8: Per Node *Segmented-Reduction*

Die Lösung hierzu ist ein Hybridansatz. In den ersten Ebenen wird eine herkömmliche *Reduction* pro Node vom Host aus gestartet und erst ab einer gewissen Anzahl an Nodes (Segmenten) wird die *Segmented-Reduction* angewendet. In Abbildung 6.13 wird die durchgängige Verwendung einer *Reduction* der einer *Segmented-Reduction* gegenübergestellt. Der Throughput der *Segmented-Reduction* wird mit steigender Anzahl an Segmenten immer höher, im Gegensatz zur Standard *Reduction*.

6.9 Clipping und Partitioning: Clip-Mask

Ein Problem beim Clipping und Partitioning der Events ist die hohe Anzahl an Daten, die bewegt werden muss. Wie in Kapitel 4.3.2 beschrieben, werden die Events erst an der Split-Ebene geclippt und anschließend wieder mit Hilfe einer *Compaction* zusammengeschoben (vgl. Abb. 4.3). Beim Clipping müssen alle Daten eines Events verschoben werden. Die Clip-Mask ist aus der Motivation heraus entstanden, die Datenmenge beim Clipping auf ein Minimum zu reduzieren. Hierzu wird eine Datenstruktur angelegt, die für jede der drei Achsen Informationen über ein Event in Bezug auf den Split der Node speichert (vgl. Tabelle 6.1).

Algorithm 10 CreateClipMask

```
foreach Event e ∈ events[axis] × axis ∈ {X,Y,Z} in parallel do
    n ← activeNodes[e.nodeId]
    s ← splits[n.pBegin]
    if getAxis(e.bbox.max, s.axis) ≤ s.plane then
        maskBelow ← {below, s.axis, e.id}
    else if getAxis(e.bbox.min, s.axis) ≥ s.plane then
        maskAbove ← {above, s.axis, e.id}
    else
        if axis = s.axis then
            if event is below then
                maskBelow ← {below, s.axis, index, overlaps, sameAxis}
            else
                maskAbove ← {above, s.axis, index, overlaps, sameAxis}
            end if
        else
            maskLeft ← {below, s.axis, e.id, overlapping}
            maskRight ← {above, s.axis, e.id, overlapping}
        end if
    end if
end foreach
```

Tabelle 6.1: Clip-Mask Bits und ihre Bedeutung

Bit	0x01	0x02	0x04	0x08	0x10	0x20	0x40
Bedeutung	below	above	overlaps	X-Achse	Y-Achse	Z-Achse	sameAxis

Das Event wird jetzt nicht direkt geclippt. Lediglich ein minimaler Satz an Informationen wird in der Clip-Mask kodiert, um erst beim späteren *Compaction* alle Daten des Events zu verschieben. Die Clip-Mask reserviert für jedes Event ein Byte. In

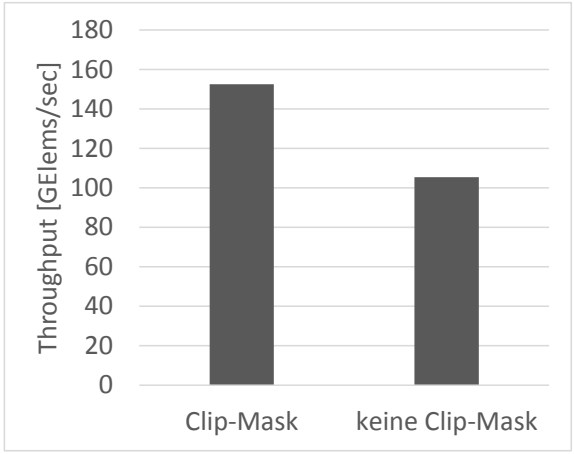

Abbildung 6.14: Clipping und Partitioning mit und ohne Clip-Mask

diesem Byte werden alle Informationen über die acht verfügbaren Bits kodiert und können durch einfache Bit-Masken abgefragt werden. Für ein Event wird seine Lage bezüglich des Split-Kandidaten codiert. Entweder ist es unter (below) oder über (above) der Split-Ebene oder überdeckt (overlaps) diese. Neben der Lage wird auch die Achse gespeichert, gegen die das Event geclippt wird. Dies erspart bei der späteren *Compaction* den Zugriff auf den Split-Kandidaten der Node (siehe Kapitel 4.3.2). Des Weiteren wird auch in der Clip-Mask festgehalten, ob es sich um ein Event handelt, welches sich auf der gleichen Achse wie der Split befindet.

Die Generierung der Clip-Mask ist im Pseudocode 10 beschrieben. Dies geschieht, wie in Kapitel 4.3.2 beschrieben, durch die Einordnung des Events bezüglich des gewählten Splits. Das Setzen der Bits ist im Pseudocode 10 durch die symbolische Zuweisung einer Menge zu den beiden Masken *maskBelow* und *maskAbove* dargestellt. Nach der Erstellung der Clip-Mask kann jetzt die *Compaction* der Events stattfinden. Hierzu muss zuerst eine *Prefix Sum* aus den validen Einträgen der Clip-Mask berechnet werden. Ein invalider Eintrag bezeichnet einen Eintrag in der Clip-Mask, an dessen Stelle kein Eintrag geschrieben worden ist. Alle validen Einträge bestehen aus einer Kombination aus gesetzten Bits. Somit kann ein valider Eintrag durch den Vergleich mit Null ermittelt werden.

In Abbildung 6.14 ist der Performance Gewinn bei der Konstruktion des KD-Trees unter Verwendung der Clip-Mask dargestellt.

6.10 Ergebnisse der Optimierungen

Der Einfluss der besprochenen Optimierungen auf die Laufzeit des Algorithmus ist in Tabelle 6.2 dargestellt. Zusammen ergeben alle Optimierungen eine durchschnittliche Beschleunigung um den Faktor $3,32$ gemessen über alle Test-Instanzen.

Tabelle 6.2: Ergebnisse der Optimierung und Einflüsse auf die Sub-Algorithmen

Optimierung	Faktor	SA 1	SA 2	SA 3	SA 4	SA 5
SAH Shuffle-Reduction	1,03		X			
Binary-Scan	1,11		X	X		X
Streams	1,2	-	-	-	-	-
Clip-Mask	1,32			X		
Hybrid-Reduction	2,25		X			
Alle Optimierungen	3,32					

In Abbildung 6.15 sind die neuen prozentualen Verteilungen der Sub-Algorithmen für die optimierte Version der Implementierung dargestellt. Der größte Anteil ist nun dem Sub-Algorithmus *ClippingAndPartitioning* zuzuschreiben. Die Berechnung der besten Split-Kandidaten (*ComputeBestSplits*) ist jetzt mit weniger als 20% vertreten. Zusätzlich ist durch ein X deutlich gemacht, welche Optimierung Einfluss auf welche Sub-Algorithmen (nach Abarbeitungsreihenfolge durchnummeriert) haben. Da Streams eine generelle Optimierung zwischen der Kommunikation zwischen Host und Device darstellen, besteht von ihnen kein direkter Einfluss (-) auf die Sub-Algorithmen.

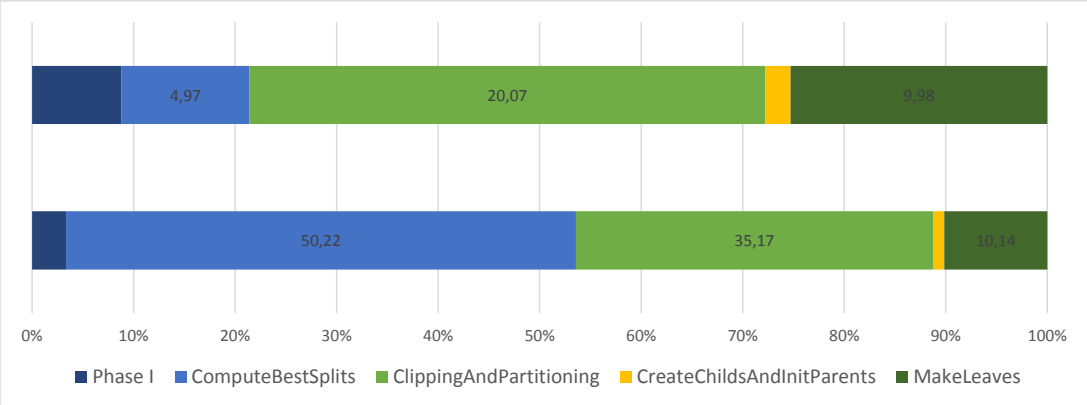

Abbildung 6.15: Laufzeitverteilung nach (oben) und vor (unten) den Optimierungen

Abbildung 6.16 zeigt zum Vergleich die absoluten Laufzeiten der Sub-Algorithmen. Die Optimierung der zwei Sub-Algorithmen (*ComputeBestSplits* und *ClippingAndPartitioning*) ist gelungen. Beide Sub-Algorithmen profitieren vom optimierten *Scan*. Die Berechnung der besten Split-Kandidaten wird vor allem durch die *Hybrid-Reduction* optimiert. Das Clipping und Partitioning wird hauptsächlich durch die Verwendung der Clip-Mask beschleunigt.

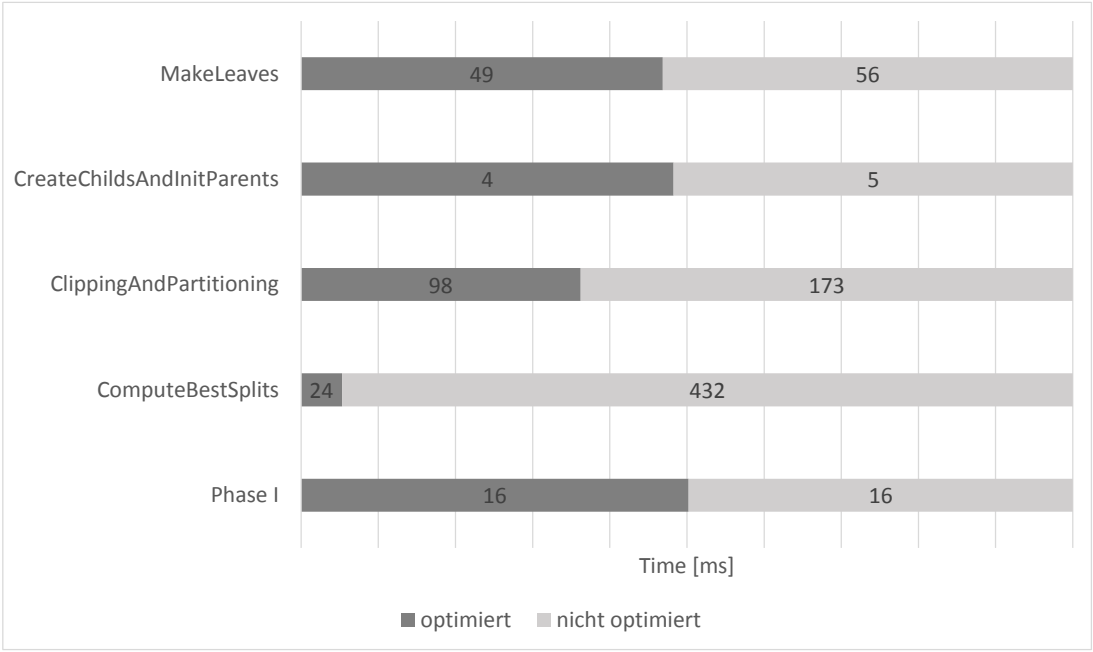

Abbildung 6.16: Laufzeiten nach und vor den Optimierungen

Kapitel 7

Schluss

7.1 Ergebnisse

Um die Qualität des KD-Trees zu testen, wurde ein GPU-basierter rekursiver Raytracer implementiert und die Performance unter Verwendung einer NVIDIA GeForce GTX Titan gemessen. Die Traversierung des KD-Trees auf der GPU basiert auf der Arbeit *Interactive k-D Tree GPU Raytracing* [12] und verwendet den dort vorgestellten *short-stack* Algorithmus.

Für die Messungen werden die bereits genannten Testszenen *Bunny*, *Happy*, *Dragon* und *Angel* verwendet. Die Testszenen sind in Abbildung 7.1 dargestellt. Leaf-Nodes werden in dieser Implementation ab einer Anzahl kleiner als 64 Primitiven erzeugt.

Tabelle 7.1: Wu et al. und diese Arbeit im Vergleich. Tracing mit 1024 × 1024 Pixeln

		GPU (Wu)		GPU (Diese Arbeit)		CPU (KD-Tree)
Szene	Prims	Build	Trace	Build	Trace	Trace
Bunny	69K	0.059s	0.031s	0.035s	0.013s	0.013s
Dragon	871K	0.511s	0.041s	0.296s	0.021s	0.021s
Happy	1087K	0.645s	0.051s	0.448s	0.020s	0.019s
Angel	474K	0.311s	0.036s	0.161s	0.015s	0.014s

Verglichen wird die Implementation des vorgestellten Algorithmus mit der von Wu et al. . Hierzu sind die Konstruktionszeiten (Build) und die Trace-Zeiten (Trace) in der Tabelle 7.1 für jede Testszene aufgetragen. Im Rahmen dieser Arbeit ist es nicht möglich gewesen, die Implementation von Wu et al. und die dieser Arbeit auf dem gleichen System zu messen. Die Implementation von Wu et al. ist nicht öffentlich im Netz verfügbar und auf eine Anfrage per E-Mail wurde keine Antwort erhalten. Deshalb sind in der Tabelle 7.1 immer noch die Messungen von Wu et al. eingetragen (NVIDIA GeForce GTX 280). Die Implementation aus dieser Arbeit liefert im Schnitt bei allen Testszene eine um den Faktor 1.67 geringere Laufzeit.

Zusätzlich sind in der Tabelle die Trace-Zeiten dargestellt, welche auf einem implementierten CPU Algorithmus zur Erzeugung des SAH KD-Trees basieren. Diese Implementation passt die AABBs der geclippten Primitive (wie bei Wu et al.) auf allen Achsen an. Durch das optimale Clippen der AABBs kann bei den Szenen *Angel* und *Happy* die Performance leicht verbessert werden. Die Szenen *Dragon* und *Bunny* zeigen keinen relevanten Unterschied. Die leicht höhere Trace-Zeit bei der Testszene *Dragon* gegenüber *Happy* ist dadurch zu erklären, dass mehr Rays auf das Objekt treffen.

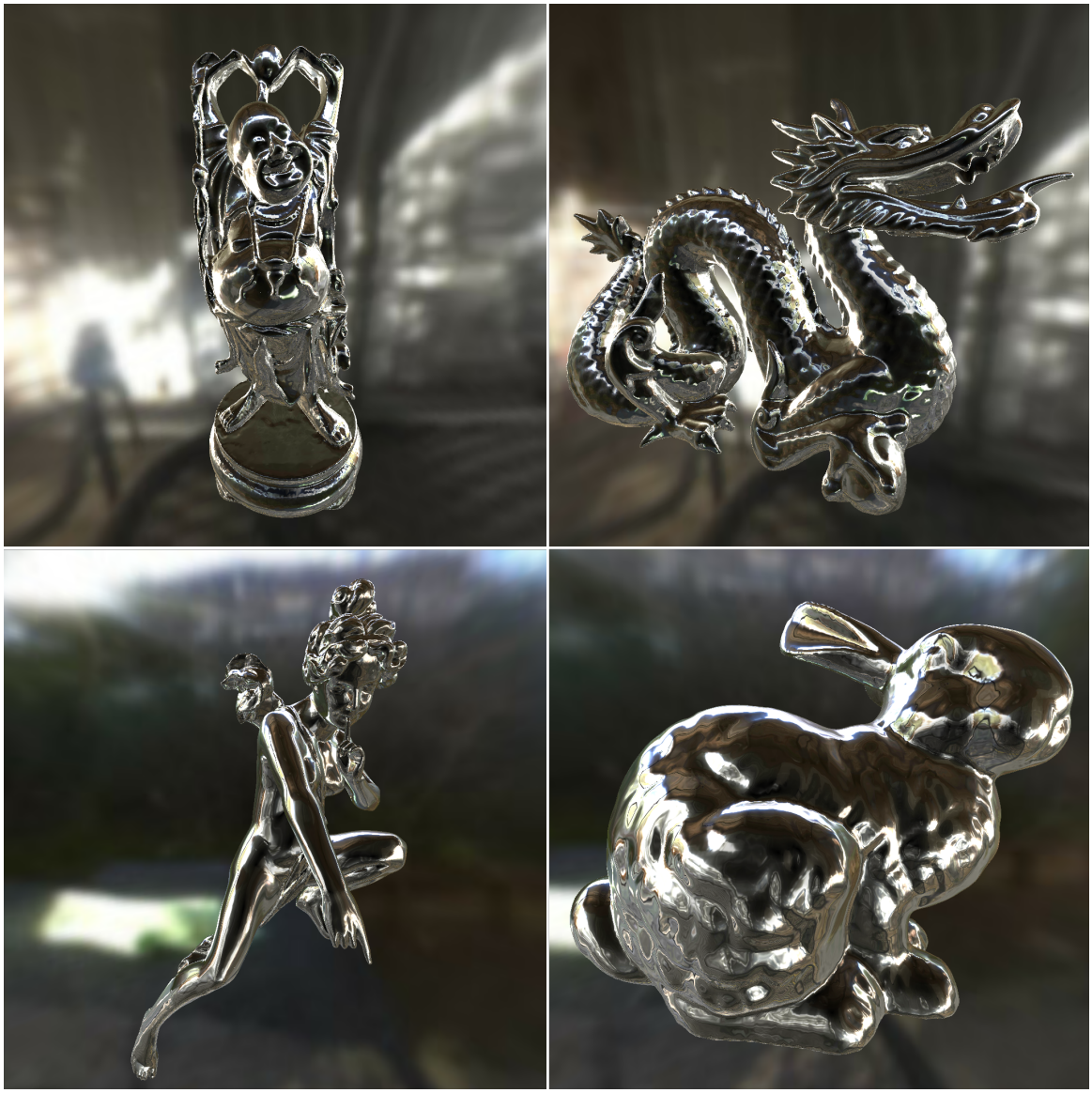

Abbildung 7.1: Happy, Dragon, Angel und Bunny; 1024 × 1024 Pixel (eigene Screenshots)

7.2 Beispiel einer dynamischen Szene

Mit dem für diese Arbeit programmierten Framework (Rekursiver Echtzeit-Raytracer und entwickelter SAH KD-Tree) wird zum Abschluss eine dynamische Szene (Ring aus Kugeln rotiert um das Objekt *Bunny*) berechnet. Hierbei ist einmal das *Bunny* aus einem metallischen Material (Abb. 7.2) und einem gläsernen Material (Abb. 7.3), zu sehen auf den nächsten zwei Seiten, modelliert. Der Raum, in dem sich die Objekte befinden, ist zusätzlich verspiegelt.

Für das Raytracing (1024×768 Pixel) (110 ms, ca. 9 FPS) und die Erzeugung des KD-Trees (40 ms, ca. 25 FPS) werden in der Summe etwa 7 Frames pro Sekunde erreicht. Hierbei werden ca. 2 Mio. Rays (Schatten, Reflektion und Refraktion) erzeugt, die zugrundeliegende Szene besteht aus $\sim$ 100K Dreiecken.

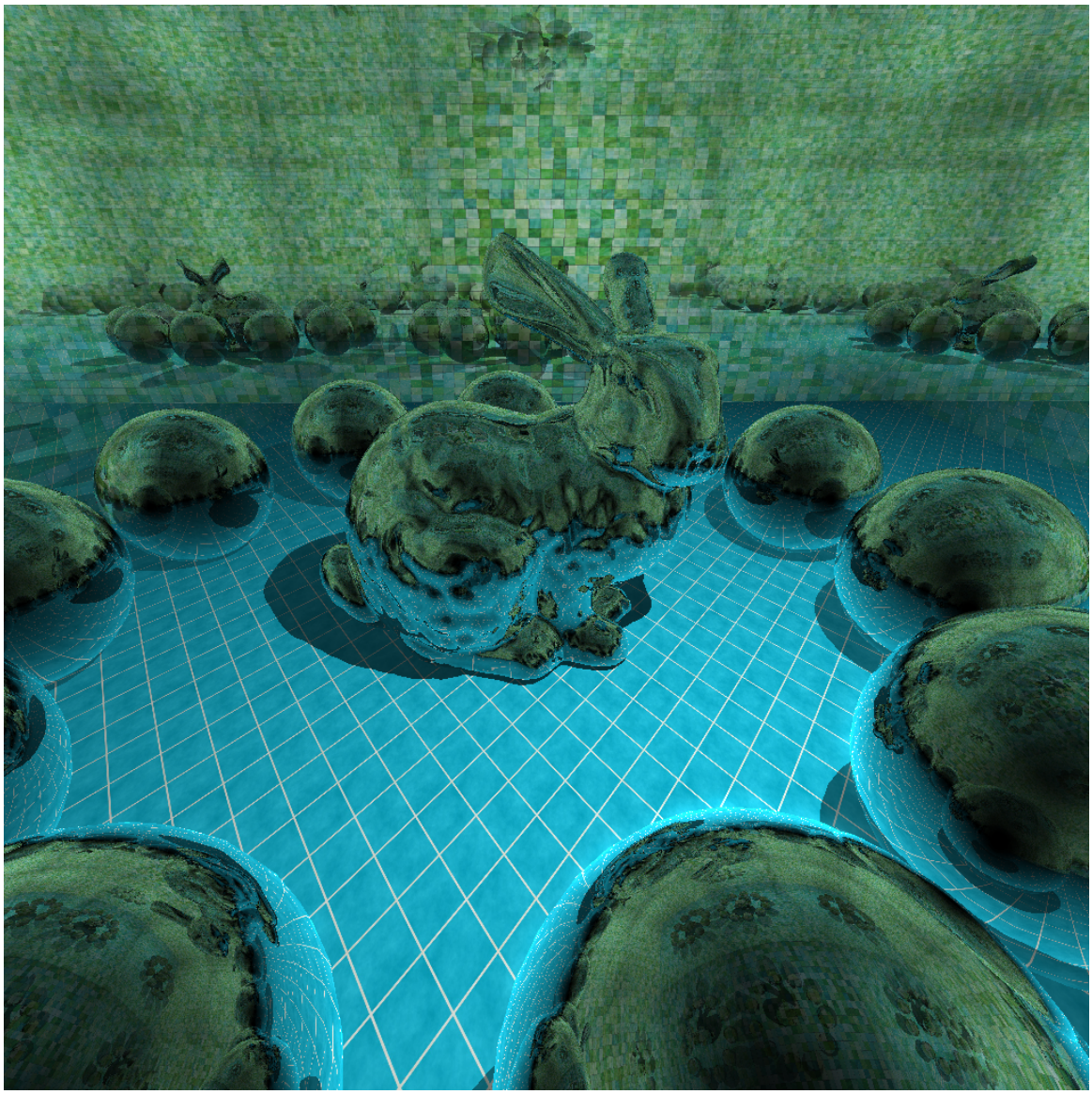

Abbildung 7.2: Rekursives Raytracing: Bunny aus Metall

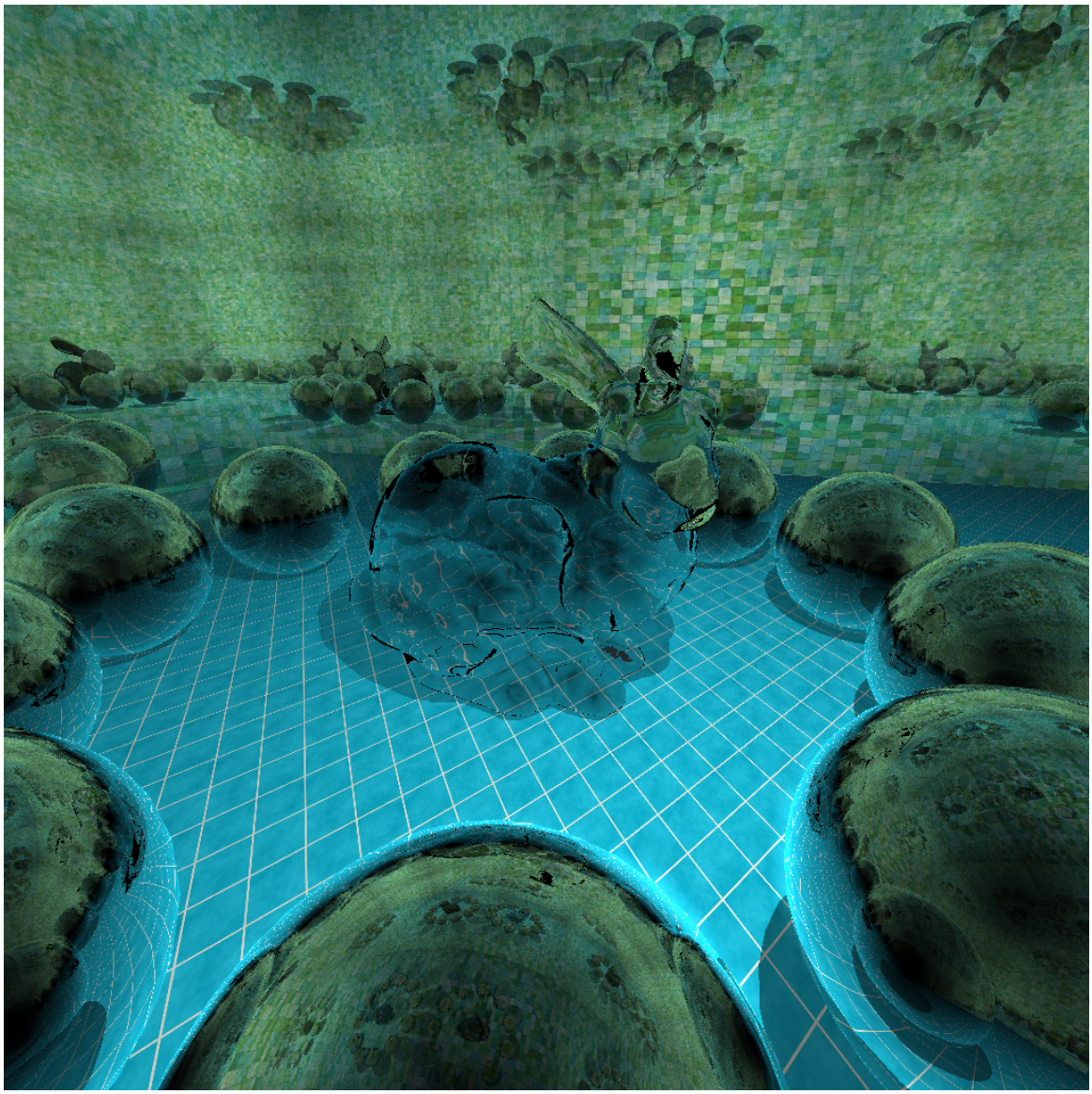

Abbildung 7.3: Rekursives Raytracing: Bunny aus Glass

7.3 Fazit

Ziel dieser Arbeit war die Formulierung eines parallelen Algorithmus zur Konstruktion eines SAH KD-Trees und die Implementierung auf einer modernen GPU, um das Raytracing dynamischer Szenen zu unterstützen. Dieses Ziel konnte erreicht werden.

Der Algorithmus besteht aus zwei Phasen. Die Initialisierungsphase (Phase I) ist für die Erzeugung der benötigten Datenstrukturen und das Sortieren der Events zuständig. Die darauf folgende Konstruktionsphase (Phase II) arbeitet in sequentiellen Schritten. Jeder Schritt bildet hierbei die Konstruktion einer Ebene des KD-Trees ab. Ein Schritt wird in vier Sub-Algorithmen unterteilt.

Der Sub-Algorithmus *"ComputeBestSplits"* berechnet parallel alle SAH-Werte der Events und bestimmt im Anschluss daran die besten Splits für alle Nodes. Im Sub-Algorithmus *"ClippingAndPartitioning"* werden die Events zuerst parallel an den gewählten Splits geclippt und dann auf die entstehenden Child-Nodes verteilt. Die Erstellung der Child-Nodes und Initialisierung ihrer Parent-Nodes erfolgt parallel über den Sub-Algorithmus *"CreateChildsAndInitParents"*. Zur Verarbeitung der Leaf-Nodes dient der Sub-Algorithmus *"MakeLeaves"*.

Der in dieser Arbeit verwendete Ansatz zum Clippen der Primitive unterscheidet sich von denen vorheriger Ansätze. Events werden nur auf der Split-Achse modifiziert, wodurch keine neue Sortierung der geclippten Events in jeder Ebene des Trees benötigt wird. Allerdings kann diese Methode dazu führen, dass die AABB eines Primitives dieses nicht mehr eng umschließt. Ein Qualitätsunterschied zwischen dem erzeugten KD-Tree und einem optimal geclippten KD-Tree konnte bei den getesteten Szenen nur minimal beobachtet werden.

In Kapitel 5 wird der formulierte Algorithmus auf seine theoretischen Eigenschaften hin untersucht. Der Algorithmus benötigt nicht mehr als $\mathcal{O}(n \log n)$ Operationen (theoretische untere Schranke eines sequentiellen Algorithmus) und ist somit *work-optimal*.

Die im Anschluss daran vorgestellten Optimierungen der Implementation für eine GPU konnten die Laufzeit einer ersten Version der Implementierung um den Faktor $3,32$ beschleunigen. Durch die Einführung der Clip-Mask wird die Implementation des Clipping und Partitioning optimiert. Vor allem mit Hilfe der in Kapitel 6.8 vorgestellten Kombination aus *Segmented-Reduction* und *Reduction* kann die Implementation des Sub-Algorithmus drastisch beschleunigt werden.

Laufzeitmessungen der Implementation werden im Kapitel 7.1 präsentiert. Da die Implementation von Wu et al. nicht öffentlich ist und auf eine Anfrage keine Antwort erhalten wurde, konnten die beiden Implementationen nicht auf demselben System

gemessen werden, um einen direkten Vergleich durchführen zu können. Die Implementation aus dieser Arbeit läuft auf einer NVIDIA GTX Titan 1.67 mal schneller als die Implementation von Wu et al. (NVIDIA GTX 280).

Abschließend bleibt zu sagen, dass der in dieser Arbeit formulierte parallele Algorithmus und seine Implementation dazu beitragen kann, in echtzeitkrischen Raytracing-Anwendungen das Rendern von dynamischen 3D Szenen zu beschleunigen.

Literaturverzeichnis

[1] Timo Aila, Tero Karras, and Samuli Laine. On Quality Metrics of Bounding Volume Hierarchies. *High-Performance Graphics*, 2013.

[2] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient Stream Compaction on Wide SIMD Many-Core Architectures. *In Proceedings of High Performance Graphics*, 2009.

[3] Guy E. Blelloch. Prefix Sums and Their Applications. *School of Computer Science Carnegie Mellon University*, 1993.

[4] Laurence Boxer and Russ Miller. *Algorithms Sequential and Parallel: A Unified Approach*. 2013.

[5] Julien Demouth. Shuffle: Tips and Tricks, 2014.

[6] Tim Foley and Jeremy Sugerman. KD-Tree Acceleration Structures for a GPU Raytracer. *Graphics Hardware*, 2005.

[7] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7 (May), 14–20*, 1987.

[8] Takahiro Harada and Lee Howes. Introduction to GPU Radix Sort. *Heterogeneous Computing with OpenCL*, 2011.

[9] Vlastimil Havran. Heuristic Ray Shooting Algorithms. *Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague*, 2000.

[10] W. Daniel Hills and jr. Guy L. Steele. Data Parallel Algorithms. *Commun. ACM 29, 1170–1183*, 1986.

[11] Jared Hoberock and Nathan Bell. Thrust. https://code.google.com/p/thrust/, 2014.

[12] Daniel Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-D Tree GPU Raytracing. *I3D*, 2007.

[13] J. David MacDonald and Kellogg S. Booth. Heuristics for raytracing using space subdivision. *Vis. Comput. 6 (May), 153–166*, 1990.

[14] mikepan. BMW 3 Series Coupe. http://www.blendswap.com/blends/view/6917, 2011.

[15] Gordon Müller and Dieter W. Fellner. Hybrid Scene Structuring with Application to RayTracing. *International Conference on Visual Computing (ICVC'99), 19-26, February*, 1999.

[16] Kelvin Ng and Borislav Trifonov. Automatic Bounding Volume Hierarchy Generation Using Stochastic Search Methods. *In Proc. Mini-Workshop on Stochastic Search Algorithms*, 2003.

[17] NVIDIA. NVIDIA-Kepler-GK110-Architecture-Whitepaper. 2012.

[18] NVIDIA. CUDA 6.0 SDK. https://developer.nvidia.com/cuda-zone, 2014.

[19] NVIDIA. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, 2014.

[20] NVIDIA. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2014.

[21] NVIDIA. CUDA-Toolkit. https://developer.nvidia.com/cuda-zone, 2014.

[22] NVIDIA. Faster Parallel Reductions on Kepler. http://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/, 2014.

[23] NVIDIA. GeForce GTX TITAN. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications, 2014.

[24] NVIDIA. Parallel Prefix Sum (Scan) with CUDA. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html, 2014.

[25] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition.* 2010.

[26] Lawrence D. Stone. *Theory of Optimal Search.* 1975.

[27] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). *Symposium on Interactive Ray Tracing, 61–69*, 2006.

[28] Zhefeng Wu, Fukai Zhao, and Xinguo Liu. SAH KD-tree construction on GPU. *ACM SIGGRAPH Symposium on High Performance Graphics*, 2011.

[29] Kun Zhou, Rui Wand, Baining Guo, and Qiming Hou. Real-time kd-tree construction on Graphics Hardware. *ACM Transactions on Graphics*, 2008.

Erklärung zur selbstständigen Abfassung der Masterarbeit

Ich versichere, dass ich die eingereichte Masterarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als die von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Greven, den 5. Oktober 2014 _____

Node-Data	
Name	**Beschreibung**
pCount	Anzahl der Primitive in der Node
pBegin	Startindex der Primitive in der Node
aabb	AABB der Node
isLeaf	Node ist ein Blatt: true
split	Split-Ebene, auf der die Node geteilt wurde
axis	Split-Achse, auf der die Node geteilt wurde
leafId	Referenz auf Leaf-Data
leftChild	Referenz auf linke Child-Node
rightChild	Referenz auf rechte Child-Node

Event-Data	
Name	**Beschreibung**
type	START- / END-Type
plane	Schnittebene auf der Achse
node	Referenz auf dazugehörige Node
primId	Referenz auf dazugehöriges Primitive
aabb	AABB für das Clipping
axis	Split-Achse des Events
isLeaf	Event gehört zu einer Leaf-Node: true

Split-Data	
Name	**Beschreibung**
pBelow	Anzahl der Primitive unter dem Split
pAbove	Anzahl der Primitive über dem Split
axis	Split-Achse des Splits
sah	Berechnete Surface Area Heuristic
plane	Schnittebene auf der Achse
Leaf-Data	
Name	**Beschreibung**
pCount	Anzahl der Primitive innerhalb des Leafs
pBegin	Startposition der Primitive innerhalb des Leafs
primIds	Referenz auf enthaltene Primitive

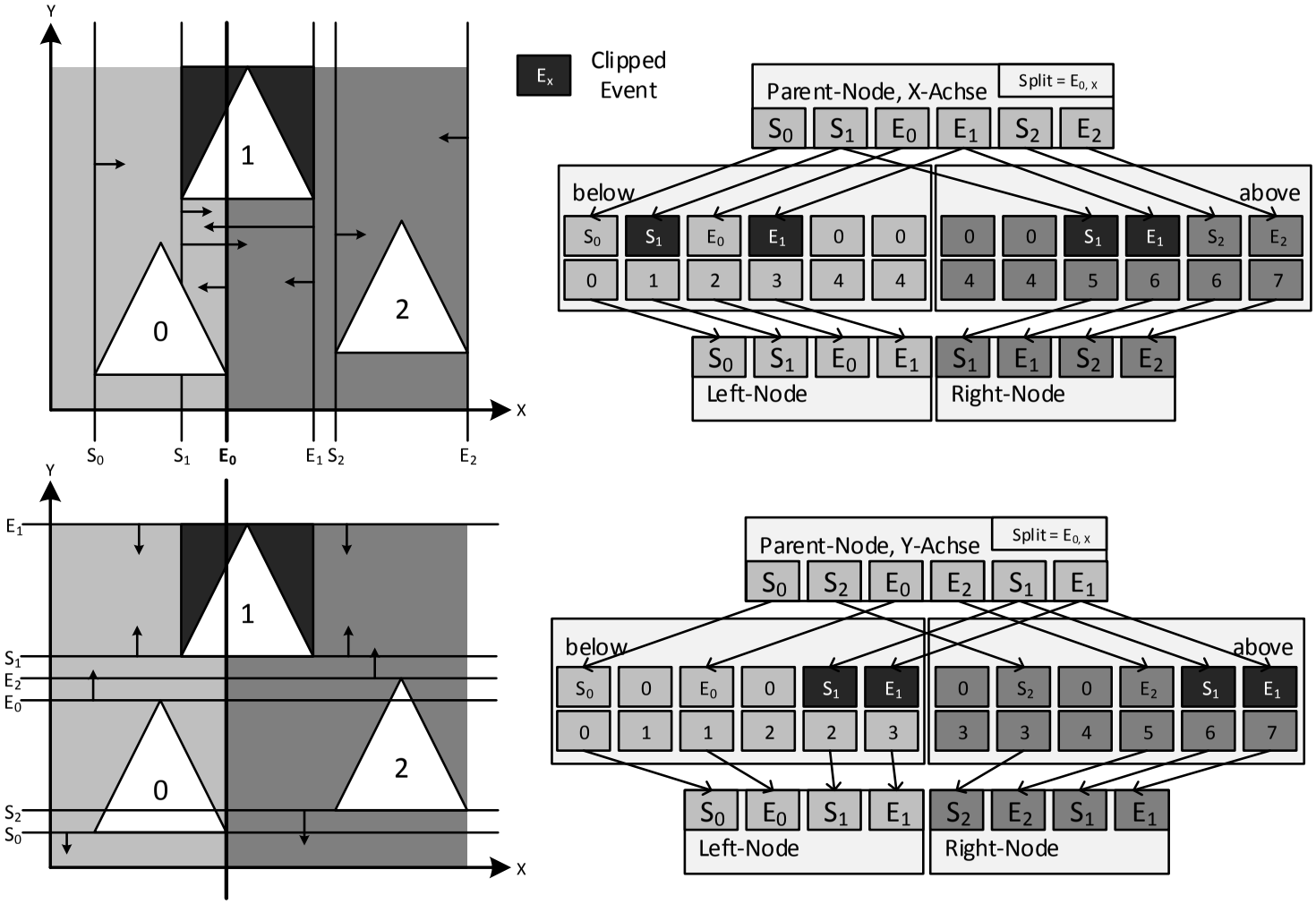

Abbildung 7.4: Clipping und Partitioning