Bachelorarbeit

Analyse der Qualität von Top-Down erzeugten Wide BVHs in statischen 3D-Szenen

Nick Garczorz

12. Januar 2026

Heinrich-Heine-Universität Düsseldorf

Erstgutachter:in: Dr. Andreas Abels
Zweitgutachter:in: Dr. Markus Brenneis

Zusammenfassung

tbc

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Effiziente Kollisionserkennung hat grundlegenden Nutzen in diversen Feldern, über physikbasierte Simulation und Robotik, bis hin zu Darstellungen in Animation und Videospielen [1]. Im Fall von Animation und Bildgebung im Allgemeinen handelt es sich vor allem um Kollisionen einfacher geometrischer Formen, häufig Geraden oder Strahle, Dreiecke und Quader. Unter Zuhilfenahme dieser einfachen Werkzeuge lassen sich komplexe dreidimensionale Szenen mit einfachen Mitteln berechnen und darstellen. Für jeden Bildpunkt werden Strahlen, genannt Rays, auf die digitale Szene geworfen. Diese besteht in gängiger Praxis aus einer Ansammlung von Dreiecken, genannt Polygone, oder anderen fundamentalen Formen, die allgemein als Primitive bezeichnet werden. Um herauszufinden, ob und wo es eine Kollision zwischen einem Ray und einem bestimmten Polygon gibt, kann man Beispielsweise den weitverbreiteten Algorithmus von Möller und Trumbore [2] verwenden. Wenn es einen Treffer gibt, lässt sich aufbauend auf Position und Abstand des Schnittpunktes der korrekte Farbwert für den Pixel berechnen. Diese Rechnung müsste man naiv gesehen für jede Kombination aus Rays und Primitive wiederholen, um ein vollständiges Bild zu generieren.

1.2 Thema und fachliche Einordnung

Um den Vorgang zu beschleunigen, kommen zwei Datenstrukturen ins Spiel: Die Axis Aligned Bounding Box (AABB) und darauf aufbauend die Bounding Volume Hierarchy (BVH) [3]. Eine AABB definiert achsenorientierte obere und untere Schranken für Teilmengen von Primitives in Form eines Quaders, der diese im Raum einschließt. Durch die Einführung von AABB lassen sich die nötigen Ray-Object-Intersection-Tests verringern, indem man für jeden Ray erst die Kollision mit den AABBs überprüft. Wenn es bereits keine Kollision mit der umschließenden Bounding Box gibt, kann von vornherein ausgeschlossen werden, dass es eine Kollision mit den innenliegenden Primitiven gibt. Um den Vorteil dieses Verfahrens effektiv zu nutzen, kann man nun rekursiv die eingeschlossenen Teilmengen wieder aufteilen und mit AABBs umschließen. Dadurch lassen sich die Bounding Boxes in eine hierarchische Reihenfolge bringen, die eine Baumstruktur aufweist. Diese Struktur ist eine Bounding Volume Hierarchy. Durch eine BVH wird die Laufzeit auf ein Niveau gesenkt, das logarithmisch von der Anzahl an Primitives in der Szene abhängt, was die Zeit für ein fertig gerendertes Bild drastisch sinken lassen kann. Die sinnvolle Konstruktion einer BVH ist ein Kernpunkt dieser Arbeit.

1.3 Zielsetzung

Es existieren Konstruktionsverfahren verschiedener Komplexitätsklassen, deren Endprodukt sich auf die Traversal Time (Renderzeit) auswirkt. Von der Construction Time (Konstruktionszeit) lässt sich die Nutzbarkeit einer BVH in dynamischen Szenen, also Szenen mit bewegten Objekten, ableiten. Die Traversal Time ist das Hauptqualitätsmerkmal für eine BVH, da sich durch niedrigere Zeiten höhere Auflösungen und Bildwiederholraten,

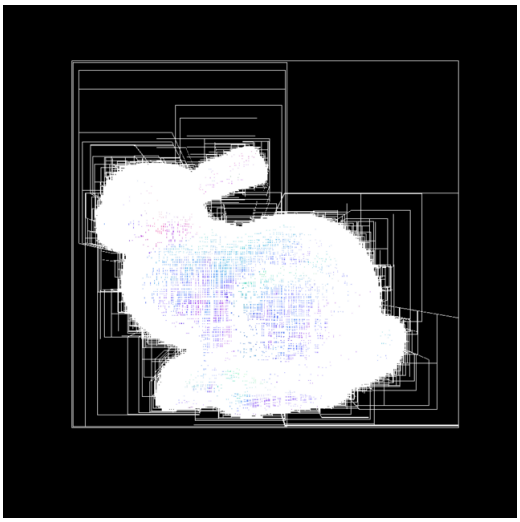

Abbildung 1.1: Stanford Bunny mit sichtbaren Bounding Boxes

sowohl in statischen wie auch dynamischen Szenen, realisieren lassen. Ein genutzter Vorteil der Baumstruktur ist, dass sich sowohl Construction (Konstruktion) als auch Traversal (Durchsuchung) auf GPUs parallelisieren lassen. Dabei spielt auch die konstruktion mit höheren Verzweigungsgraden eine Rolle, wobei man auch dies durch unterschiedliche Konstruktionsverfahren erreichen kann, wie dem *k-way splitting* oder durch *collapse* (Kollabierung) der Hierarchie. Diese Art von BVH wird Wide BVH (Weite BVH) oder auch BVHk genannt. Ziel der Arbeit ist die Analyse verschiedener Konstruktionsverfahren von Wide BVHs mit Hinblick auf Einsparung in Traversal- aber auch Construction-Zeit in statischen Szenen. Dabei sollen die Ergebnisse abhängig von dem verwendeten konstruktionsalgorithmus, dem Verzweigungsgrad der Hierarchie und der Primitive-Anzahl der Szene beleuchtete werden. Dabei haben sich drei Problemfragen ergeben, die den Scope dieser Arbeit definieren:

- Wie verändert sich die Traversal-Time bei Top-Down-Konstruierten Wide BVHs für Verzweigungsgrade $k \in \{2, 4, 8, 16\}$ in statischen Szenen?
- Unterscheidet sich die Traversal-Time der BVHs zwischen direktem k-way splitting und collapse einer binären BVH abhängig von der gewählten statischen Szene?
- Wie verändern sich Construction- und Traversal-Time als kombinierte Qualitätsmetrik für die modellierung dynamischer Szenen mit Bezug auf das gewählte Splitting-Verfahren jeweils über $k \in \{2, 4, 8, 16\}$?

1.4 Abgrenzung zu vorherigen Arbeiten

In der zugrundeliegenden Arbeit werden lediglich statische dreidimensionale Szenen untersucht. Vefahren die darauf abzielen die BVH dynamisch anzupassen (Refitting etc.) werden nicht untersucht.

Zudem werden ausschließlich Top-Down-Konstruktionsmethoden verwendet, da dies der Industriestandard ist [?].

Die Verglichenen Datenstrukturen sind lediglich BVH bzw Wide BVH, es werden keine anderen Beschleunigungsstrukturen, wie k-d-Trees beachtet. Wide BVHs werden im Vergleich zu ihrer binären Referenzimplementation untersucht.

Es werden ganz explizit verschiedene Verfahren zur Konstruktion von Wide BVHs untersucht und verglichen. Das sogenannte k-way splitting, also das Partitionieren der Menge in k Teilmengen in einem Iterationsschritt entlang einer Achse, ist das erste Untersuchte Verfahren. Auf der anderen Seite steht das Collapse-Verfahren bei dem Nodes einer Binären Hierarchie nach der Konstruktion mit ihren Kindern verschmolzen werden um den Verzweigungsgrad zu verdoppeln.

Es werden lediglich Median-Split und SAH-Split, sowie die Optimierungsvariante binned SAH-Split untersucht, keine komplexeren Varianten wie Spatial-Splits. Zudem ist die Anzahl der Bins auf 16 festgelegt. Der Fokus soll nicht auf den verwendet Algorithmen, sondern auf den Auswirkungen der Verfahren bezüglich dem Verzweigungsgrad liegen.

Die gemessenen Metriken beziehen sich auf CPU-Zeit, also lineare Berechnungen. Spezielle Raytracing-Hardware oder parallelisierte Messergebnisse werden nicht untersucht, das der Fokus auf Gewinn durch strukturelle Eigenschaften der Hierarchien liegen soll.

Die bei der Traversal-Zeit verwendeten Rays sind lediglich *Primärstrahlen*, also direkte Strahlen zwischen virtueller Kamera und Objekt. Dadurch fallen Verfahren wie Refelktion, Refraktion o.Ä. aus der Bewertung, was die Vergleichbarkeit vereinfacht und die Komplexität verringert.

Die Qualität einer BVH wird rein durch die Traversal-Time definiert. Andere Aspekte wie zum Beispiel Bildqualität durch Verwendung verschiedener Shader sind nicht Teil der Untersuchung.

2 Grundlagen

2.1 Modell

Für das weitere vorgehen werden unter Primitives immer Polygone, also Dreiecke, verstanden. Polygone werden durch ihre drei Eckpunkte definiert, die jeweils durch dreidimensionale Vektoren beschrieben werden. zudem wird für die Vereinfachung von Berechnungen der Mittelpunkt des Polygons bestimmt, *Centroid* genannt. Eine zusammenhängende Struktur aus Polygonen wird als *Mesh* verstanden.

Wenn sich im Traversal-Schritt die Frage stellt, ob ein Ray ein Primitive schneidet, stellt sich eigentlich die Frage, ob ein bestimmtes Primitive das erste ist, was der untersuchte Ray schneidet. Man spricht hier von *first hit traversal*[?].

2.2 Axis Aligned Bounding Box

Um Konstruktion und Traversal von Nodes einer BVH möglichst einfach zu gestalten müssen einige Punkte gegeben sein:

- Sie müssen eine möglichst performante Möglichkeit bieten die Kollision mit einem Ray zu überprüfen.
- Ihre räumlichen Grenzen sollen möglichst einfach Berechnet werden können.
- Sie sollen möglichst wenig Speicherplatz verbrauchen, bzw. möglichst wenig Zusatzinformationen benötigen.

Die am häufigsten verwendete Datenstruktur, die diese Punkte miteinander vereint ist die Axis Aligned Bounding Box. Man kann sie sich vorstellen wie ein an den Koordinatenachsen ausgerichteter Quader, der die darin enthaltenen Primitives möglichst Eng einschließt. Die räumlichen Grenzen lassen sich durch zwei Vektoren, die die oberen und unteren Schranken der Box definieren leicht definieren und durch eine lineare Suche in den enthaltenen Primitives relativ effizient berechnen.

Die Kollision mit Rays ist durch die *SLAB-Methode* sehr performant [?]. Hierbei ...

2.3 Bounding Volume Hierarchy

Eine BVH ist im Kern eine Baumstruktur, die eine sinnvolle Partiotionierung der Primitive in der Szene erwirkt. Um einen inneren Knoten zu definieren benötigt man eine AABB, Referenzen zu den eingeschlossenen Primitives und Referenzen zu ihren Child-Nodes. Die AABBs von Child-Nodes befinden sich immer innerhalb der grenzen der AABB ihres Parent-Nodes. Es ist allerdings möglich, dass die AABBS der Child-Nodes sich überschneiden. Die Wurzel ist gleich definiert. Die Blätter der Datenstruktur sind konzeptionel Nodes, deren Menge an Primitives kleiner gleich dem festgelegten Verzweigungsgrad sind. Qualitätsmerkmale einer BVH sind in etwa wenig Überschneidungen von AABBs oder AABBs mit möglichst kleinen Flächeninhalten [?].

Dabei hängt die Qualität einer BVH von der verwendeten Konstruktionsmethode ab, welche wiederum variierende Komplexität haben. Häufig existiert eine Abwägung zwischen Construction und Traversal Time.

2.3.1 Wide Bounding Volume Hierarchy

Wide BVH werden für das Arbeiten mit Grafikkarten immer interessanter, da sich durch die erweiterte Breite der BVH noch mehr Möglichkeiten für parallelisiertes Arbeiten ergeben. Dabei existiert für die in dieser Arbeit untersuchten Algorithmen immer eine Referenzmethode zur Erstellung von Binärbäumen. Die untersuchten Verzweigungsgrade sind, wie auch in Referenzarbeiten [?], vielfache von Zwei. Zudem wird so die Vergleichbarkeit zwischen den verschiedenen Konstruktionsmethoden gewährleistet, da das Collapse-Verfahren in dieser Implementation auf Binärbäumen basiert. Die direkte Konsequenzen von Wide BVH sind eine geringere Tiefe und mehr Kollisionschecks mit Child-Nodes pro Knoten. Vor- und Nachteile bezogen auf Construction und Traversal werden innerhalb dieser Arbeit untersucht.

2.3.2 Traversal

Der Traversal-Schritt ist der Kerngrund für die Verwendung einer BVH. Die Traversal-Time zu minimieren ist ihre Hauptauptaufgabe. Die Traversal geschieht dabei Top-Down durch die Hierarchie. Falls es eine Kollision zwischen Ray und AABB eines Knotens gibt, sollen auch die Child-Nodes, bzw. bei einem Blatt die Primitives, überprüft werden. Falls nicht kann der gesamte Teilbaum ausgelassen werden. Die Reihenfolge wird hier durch einen Stack gegeben. In diesem Fall handelt es sich um eine Art der Breitensuche mit frühem Abbruchkriterium pro Teilbaum. Die Zeit für einen AABB-Test und einen Primitive-Test unterscheiden sich, wodurch sich ein weiteres Abbruchkriterium ergeben kann. Dies geschieht indem man die Zeit für Ray-Primitive-Kollision, die Dauer des durchsuchens eines Teilbaums und die Menge an Primitiven im Teilbaum in eine Kostengleichung einträgt. Es existieren auch Vereinfachungen dieser Gleichung, diese werden im folgenden Erklärt.

2.3.3 Kostenmodelle

Es gibt eine Reihe von sogenannten *Traversal-Cost-Models* [?]. Hierbei gibt es eine *Cost- bzw. Loss-Function* die bei der Konstruktion des Baumes verwendet wird um den, für die Funktion, optimalen Split zu finden.

Das Grundgerüst der Kostenfunktion sieht in den meisten Fällen ähnlich aus und unterscheidet sich je nach Heuristik in der bemessenen Wahrscheinlichkeit, mit der der Teilbaum eines gegebene Nodes besucht wird. Dabei entsprechen die Kosten eines inneren Nodes N:

$$c(N) = c_T + \sum_{N_{child}} P(N_{child} \mid N)\ c(N_{child})$$

[?]

Dabei entspricht c_T der durchschnittlichen Zeit für einen Schritt während der Traversal und N_{child} wird als Iterator über alle Kinder des Knoten N verstanden.

Falls es sich nicht um einen inneren Node handelt ist die Kostenfunktion $c(N) = c_I|N|$ mit Ray-Primitive-Intersection-Time c_I und Anzahl an Primitiven in Node $|N|$.

Das bekannteste Kostenmodell zur Konstruktion einer BVH ist die *Surface Area Heuristic*, welche die in dieser Arbeit untersuchte Heuristik darstellt. Es existieren allerdings eine ganze Reihe an Modellen wie zum Beispiel die *Ray Distribution Heuristic (RDH)*[?], *Occlusion Heuristic (OH)* [?] oder *End-point Overlap Heuristic (EPO)* [?], um eine Auswahl zu nennen.

Die Untersuchung dieser Modelle ist nicht Teil dieser Arbeit.

2.3.4 Construction

Die Konstruktion von BVH wird klassischerweise entweder Top-Down oder Bottom-Up durchgeführt. Der Großteil der verwendeten Algorithmen sind Top-Down-Construction Algorithmen, welche exklusiv in dieser Arbeit besprochen werden.

Median Split

Die naivste Partitionsstrategie ist der Median Split. Dabei wird die entlang einer Achse sortierte Menge an Primitiven am Median geteilt. Das wird rekursiv für die neu entstandenen Teilmengen wiederholt, bis ein festgelegtes Abbruchkriterium erreicht wird. Sie basiert damit nicht auf einm Kostenmodell und eignet sich vor allem für im Raum gleichverteilte Primitive [?]. Sie besitzt de facto den kleinsten Overhead für die Konstruktion, da keine Abwägung der Teilmengen nötig ist.

Surface Area Heuristic Split

Dagegen stehen Kostenmodelle wie die Surface Area Heuristic (SAH). Von ihr leiten sich einige spezialisierte Partitionierungsstrategien ab, wovon eine auch näher in dieser Arbeit besprochen wird.

Die Grundidee ist es, die Wahrscheinlichkeit, dass ein Ray auf die AABB eines Nodes trifft, als geometrische Wahrscheinlichkeit zu sehen. Das bedeutet, dass der Flächeninhalt der AABB einer Node mit den potenziellen Flächeninhalten ihrer Child-Nodes ins Verhältnis gesetzt werden [?]. Daraus ergibt sich, dass kleinere Flächeninhalte die Kostenfunktion minimieren, also als besser gesehen werden. Die SAH geht davon aus, dass die Ausrichtung von Rays gleichverteilt ist [?].

$$P(N_{child} \mid N)_{SAH} = \frac{Area(N_{child})}{Area(N)}$$

Um die Kostenfunktion zu minimieren muss jeder mögliche Split untersucht werden. Da das Kostenmodell verwendet wird um Vergleichswerte zu erstellen und nicht absolute Werte zu errechnen, können Konsatnten, wie die Traversal-Time des Teilbaums oder die Zeit für die Kollisionserkennung vernachlässigt werden, was die Funktion stark vereinfacht.

Theoretisch lassen sich durch die Verwendung dieser Konstanten und das Berechnen absoluter Werte ein neues Abbruchkriterium für das Erstellen einer BVH ermitteln, allerding müssten diese Konstanten im besten Fall vorher empirisch bestimmt werden. Da ein optimales Kostenmodell nicht der Schwerpunkt dieser Arbeit ist, wurde dies nicht behandelt.

Binned Surface Area Heuristic

Die Binned Surface Area Heuristic ist eine Abwandlung, der SAH. Sie verwendet das gleiche Kostenmodell, hat allerdings einen Implementationsspezifischen Unterschied. Statt Splits zwischen jedem Primitive zuzulassen, werden die Primitives vorerst entlang der Achse in eine beliebige Menge Bins, im diesem falle 16, einsortiert. Dann wird der optimale Split zwischen diesen Bins ermittelt. Der Vorteil ist, dass lediglich ein Bruchteil an Berechnungen gemacht werden muss, um einen Split zu berechnen. Dieser Gewinn in der Construction-Time geht allerdings gegebenenfalls auf Kosten der BVH-Qualität.

3 Methoden

3.1 Aufbau

3.1.1 Überblick

Verglichen werden BVH in verschieden großen Szenen, die sich in Konstruktionsalgorithmus und Verzweigungsgrad unterscheiden. Die Untersuchten Algorithmen sind der *Median Split, Surface Area Heurustic Split* und *Binned Surface Are Heuristic Split.* Als Baseline wird in jedem Fall ein Binärer Testdurchlauf mit Verzweigungsgrad $k = 2$ durchgeführt. Zudem werden $k = 4$, $k = 8$ und $k = 16$ untersucht. Dabei existiert jeweils eine Variante in der die binäre BVH collapsed wird um die höheren Grade zu erreichen und eine Variante mit direktem k-way-split. Innerhalb der Szenen wird eine einfache Kamerafahrt in 36 Frames unterteilt. Bei der Kamerafahrt handelt es sich um eine einfachen Pfad um den Ursprung der Szene, bei dem dieser stehts im Mittelpunkt steht. In den meisten Szenen rotiert damit die Kamera um ein Objekt. Die Frames unterscheiden sich damit lediglich in Kameraposition und -Ausrichtung. Dabei werden als Hauptmetriken die Construction-Time und die Traversal-Time erfasst. Insgesamt werden pro Testdurchlauf 10 Datenpunkte für die Construction-Time und dann pro Frame jeweils die Traversal-Time gesammelt. Alle Tests wurden um Messfehler auszuschließen 10 mal wiederholt. Damit kommt man für jede Permutation von Algorithmus, Verzweigungsgrad, Wide BVH Variante und Modell auf 84 verschiedene Testruns mit je 46 Datenpunkte und 10 Wiederholungen. Also gesamtheitlich auf *38.640 Datenpunkte*. Im ersten Schritt werden Unterschiede in der Traversal-Time untersucht um auf Veränderungen in der BVH-Qualität zurückzuschließen. Im zweiten Schritt wird die Summe von Construction- und Traversal-Time betrachtet, um die Performance in einer dynamischen Szene zu modellieren, da in dynamischen Szenen die BVH theoretisch nach jeder Änderung an der Szene neu berechnet werden muss.

Die gewählten Szenen sind bekannte Testmodelle mit unterschiedlichen Größenordnungen im Bezug auf Polygonanzahl, die sich in einem ansonsten leeren Raum aufhalten. Die Szenengröße kann man folgender Tabelle entnehmen:

Szene	Polygonanzahl
Suzanne	968
Utah Teapot	6320
Stanford Bunny	69451
Stanford Armadillo	99976

Tabelle 3.1: Polygonanzahl untersuchter 3D-Modelle

Zu jedem Frame wird wie oben dargestellt ein Bild erzeugt, welches dazu dient die Korrektheit der Traversal zu untersuchen. Damit kann zudem sicehrgestellt werden, dass die untersuchte BVH eine gültige Form besitzt. Unabhängig von von Testrun sollen die erzeugten Bilder pro Testszene immer gleich sein.

3.1.2 Datenmodell

Punkte im Raum werden als Vektoren im Stil von 3er-Tupeln erfasst, hier *Vector3* genannt.

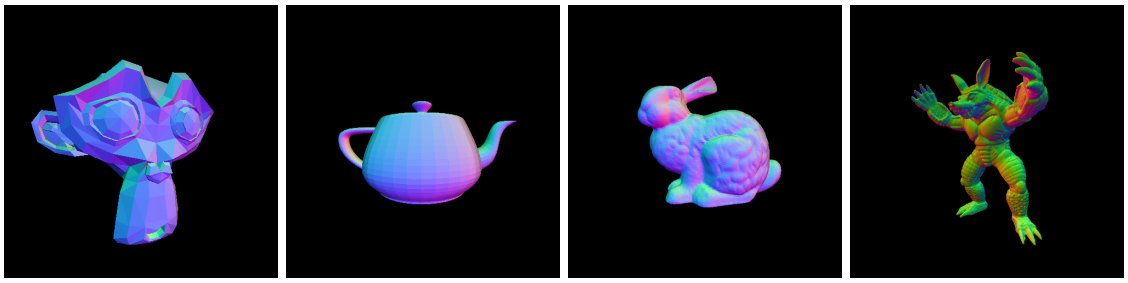

Abbildung 3.1: Suzanne, Utah Teapot, Stanford Bunny. Stanford Armadillo

Die implementierte BVH verwendet nicht direkt Polygone, sondern ist in der Lage Primitive verschiedener Arten zu verarbeiten, solange sie die benötigten Grundeigenschaften besitzen. Dazu gehören ein Mittelpunkt (Center oder auch Centroid), einen Minimal- und Maximalpunkt im Sinne einer Bounding Box, die nur das Primitive selbst einschließt und eine Funktion um Kollisionen (Intersections) mit Rays zu erkennen und gegebenenfalls den dazugehörigen Punkt im Raum zu bestimmen. Polygone verfügen über all diese Eigenschaften.

```
class Primitive
{
    Vector3 getCenter()
    Vector3 getMin()
    Vector3 getMax()
    Vector3 intersect(ray)
}
```

Listing 3.1: Primitive

AABBs verfügen über ähnliche Attribute. Sie benötigen allerdings keine referenz zu ihrem Mittelpunkt, da sie während der Laufzeit nicht entlang einer Achse sortiert werden müssen.

```
class AABB
{
    Vector3 getMin()
    Vector3 getMax()
    Vector3 intersect(ray)
}
```

Listing 3.2: Axis Alignet Bounding Box

Die BVH selbst besteht aus aufeinander referenzierende Nodes. Ein Node wird dabei durch seine Bounding Box, Referenzen zu den eingeschlossenen Primitiven und Referenzen zu seinen Child-Nodes definiert.

Innerhalb der BVH muss so nur eine Referenz auf die Wurzel (Root) gegeben sein, alle anderen Referenzen erfolgen implizit. Zudem liegt die Working-Copy aller Primitives innerhalb der BVH, auf die die jeweiligen Nodes jeweils ihre referenzen besitzen.

```
class BVHNode
{
    AABB aabb
    Primitve [] primitives
    BVHNode [] children
}

class BVH
{
    BVHNode root
    Primitive [] primitives
}
```

Listing 3.3: Bounding Volume Hierarchie

Das festgelegte Leaf-Kriterium ist wie folgt: Es existieren gleich oder weniger Primitive in der zu teilenden Menge, als vom Verzweigungsgrad verlangte splits nötig sind.

3.2 Algorithmenanalyse

3.2.1 Traversal

Für die Traversal wird ein Stack verwendet. Anfangs wird die Root der BVH auf den Stack gelegt. Im folgenden wird immer der oberste Node vom Stack genommen.

Falls es eine Kollision zwischen Ray und Bounding Box des Nodes gibt, werden falls vorhanden alle Child-Nodes auf den Stack gelegt, andernfalls geschieht nichts. Für denn Fall, dass es sich um einen Leaf-Node handelt, werden die Primitive des Nodes näher angeschaut.

Kommt es zu einer Kollision zwischen Ray und Primitive, wird die Distanz zwischen Ray-Origin und Schnittpunkt berechnet. Ist nun die Distanz kürzer als die jeder zuvor gefundene Kollision, wird sie gespeichert. Für komplexere Raytracer würde man den Schnittpunkt, sowie Eigenschaften des Primitives, wie Material oder Oberflächenwinkel des Rays, abzuspeichern, dies ist in diesem Anwendungsfall allerdings nicht notwendig.

Die Funktion gibt letztendlich die kürzeste Distanz einer Kollision zurück. Diese Art von Traversal wird als *closest hit* bezeichnet.

```
function traverse(bvh, ray):
    stack := empty list
    push(stack, bvh.root)

    closestDistance := +∞

    while stack is not empty do
        node := pop(stack)

        if not node.aabb.intersect(ray) then
            continue
        end if

        if node.children is empty then  //Leaf node
            for primitive in node.primitives do
                intersection := primitive.intersect(ray)
                if intersection then
                    distance := distance(ray.origin, intersection)
                    if distance < closestDistance then
                        closestDistande := distance
                    end if
                end if
            end for
        end if

        for child in node.children do
            push(stack, child)
        end for
    end while
    return closestDistance

end function
```

Listing 3.4: BVH Traversal

3.2.2 Construction

Die Konstruktionsfunktion unterstützt eine variable Split-Funktion, hier *partitionFunction* mit variablem Verzweigungsgrad. Nachdem eine Kopie fer Primitive für die BVH erstellt und sichergegangen wird, dass überhaupt Primitive in der Szene existieren, wird die *Root-Node* erstellt. Diese wird auf einen Stack gelegt.

Solange Nodes auf dem Stack liegen, wird der oberste vom Stack genommen und behandelt. Nachdem die längste Achse der korrespondierenden AABB berechnet wurde, wird die

ausgewählte Split-Funtion mit den Primitiven des Nodes und der Achse aufgerufen. Die daraus resultierenden Indexe werden verwendet um die Primitive aufzuteilen. Falls keine Indexe zurückgegeben werden, wird der behandelte Node als Leaf angesehen und der Schleifendurchlauf abgebrochen. Für alle neuen Teilmengen werden dann neue AABBs berechnet und neue Nodes erstellt. Diese neue Nodes werden in den behandelten Nodes als Child-Nodes referenziert. Daraufhin werden sie auf den Stack gelegt und die Schleife beginnt von neuen.

Letztendlich wird eine BVH-Instanz mit dem zuerst berechneten Root-Node und der Kopie aller Primitive zurückgegeben.

```
function build(inputPrimitives, partitionFunction):
    // move input primitives into a single owned list inside the BVH
    ownedPrimitives := copy of inputPrimitives

    if ownedPrimitives is empty then
        return BVH(root = emptyRoot, primitives = ownedPrimitives)
    end if

    // compute global bounds of all primitives
    box := computeBoundingBox(ownedPrimitives)

    // create BVH with a single root node
    rootNode := BVHNode(box       = box,
                        primitives = ownedPrimitives
                        children   = empty list)
    bvh      := BVH(root = rootNode, primitives = ownedPrimitives)

    // stack of nodes to process
    nodes := empty list
    push(nodes, bvh.root)

    while nodes is not empty do
        node := pop(nodes)

        // choose splitting axis based on longest extent
        axis := calculateLongestAxis(node.box)

        // Find right subsets according to partitionFunction
        splitIndices      := partitionFunction(primitives, axis)
        if splitIndices is empty then
            continue //Leaf Node
        end if
        primitiveSubsets  := calculateSubsets(primitives, splitIndices)

        for each primitiveSubset in primitiveSubsets do
            boundingBox  := computeBoundingBox(primitiveSubset)
            child        :=  BVHNode(box       = boundingBox,
                                     primitives = primitiveSubset
                                     children   = empty list)
            append child to node.children
        end for

        // push children onto stack for further splitting
        for each child in node.children do
            push(nodes, reference to child)
        end for
    end while

    return bvh
end function
```

Listing 3.5: BVH-Konstruktion mit gegebener Partitionsfunktion

Median Split

Der Median-Split ist die einfachste variante einer Split-Funtion. Die Menge der Primitive wird in, vom Verzweigungsgrad abhängige, gleichgroße Teile gespalten. Dafür wird die Menge der Primitive n durch den Verzweigungsgrad geteilt um die Größe einer Teilmenge zu bestimmen. Vielfache dieser Zahl innerhalb des Intervalls $(0, n)$ werden als Indexe fürs Splitting verwendet. Allerdings muss dafür noch die Menge an Primitiven, entlang der gegebenen Achse, partiell Sortiert werden, sodass für alle Indexe gilt, dass der Mittelpunkt der Position der Primitive links des Index kleiner sind als die rechts des Indexes.

```
function medianSplit(primitives, axis, degree):    //O(n)
    count := primitives.size()
    splitIndices := empty list

    for i in degree do
        split := (count * i) / degree
        if split == 0 or split >= count then
            break
        end if
        push(splitIndices, split)
    end for

    // partially sort so that element at split is in its median position in O(n)
    for split in splitIndices do
        nth_element(set    = primitives,
                    middle = split,
                    compare(primitiveA, primitiveB):
                        return center(primitiveA).component(axis) <
     center(primitiveB).component(axis))
    end for
    return splitIndices
end function
```

Listing 3.6: Median Partition

Surface Area Heuristic Split

```
function sahSplit(primitives, axis, degree):   //O(n * log(n))

    count := primitives.size()
    if isLeaf(count, degree) then
        return {}
    end if

    //O(n * log(n))
    sort(set = primitives,
         compare(primitiveA, primitiveB):
             return center(primitiveA).component(axis) < center(primitiveB).component(axis))

    splitIndices := empty list
    segments := empty list
    push(segments, primitives)
    while segments.size() < degree do

        seg := findSegmentToSplitGreedily(segments)   //highest cost segment has to be split
        // bounds of primitives from segment from begin to index
        prefixBoundingBoxes := calculatePrefix(seg)  //O(n)
        // bounds of primitives from segment from index to end (included)
        suffixBoundingBoxes := calculateSuffix(seg) // O(n)

        segCount := seg.count()
        split   := segCount / 2
        minCost := +∞
        for i from 1 to segCount - 2 do
            leftArea  := surfaceArea(prefixBoundingBoxes[i - 1])
            rightArea := surfaceArea(suffixBoundingBoxes[i])
```

```
30              cost      := leftArea * i + rightArea * (segCount - i) //simplified cost function
31              if cost < minCost then
32                  minCost := cost
33                  split   := i
34              end if
35          end for
36
37          push(splitIndices, seg.begin + split)   //convert relative split to absolute
38          erase(segments, seg)
39          //push subsets
40          push(segments, seg[seg.begin, split])
41          push(segments, seg[split, seg.end])
42      end do
43      return splitIndices
44  end function
```

Listing 3.7: Surface Area Heuristic Partition

Binned Surface Area Heuristic Split

3.2.3 Collapse Funktion

4 Ergebnisse

Alle Graphen: - Traversal-Speedup vs k (log-Skala): x = k, y = log2(Speedup) - Direct vs Collapse: Dumbbell, y = log2(Speedup)

4.1 Versuchssetup

Was wurde wie gemacht -> pipeline

4.2 Korrektheit

Die visuelle Ausgabe der Testruns ist für die jeweiligen Testszenen über alle möglichen Konstruktionsparameter konstant. Somit wird immer das gleiche Ergebnis geliefert. Zudem wird die reale Laufzeit mit der prognostizierten asympthotischen worst-case Laufzeit verglichen.

4.3 Traversal

- Hypothese Traversalzeiten ändern sich abhängig von k - H0 keine Änderung Die Darstellung der Ergebnisse orientiert sich an den Problemfragen.

Tabelle 4.1: Statistical Analysis Results (k=4)

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
armadillo	bsah	4	k-way	1.032	[1.023, 1.041]	3.40e-06***
stanford-bunny	bsah	4	k-way	1.043	[1.038, 1.052]	1.65e-07***
suzanne	bsah	4	k-way	1.082	[1.077, 1.090]	3.37e-09***
teapot	bsah	4	k-way	1.043	[1.039, 1.051]	4.93e-07***
armadillo	median	4	k-way	1.050	[1.045, 1.058]	3.11e-07***
stanford-bunny	median	4	k-way	1.061	[1.056, 1.069]	8.86e-08***
suzanne	median	4	k-way	1.069	[1.064, 1.078]	3.92e-08***
teapot	median	4	k-way	1.069	[1.063, 1.078]	2.01e-08***
armadillo	sah	4	k-way	1.033	[1.030, 1.035]	1.74e-14***
stanford-bunny	sah	4	k-way	1.038	[1.036, 1.040]	1.38e-18***
suzanne	sah	4	k-way	1.074	[1.073, 1.076]	2.92e-20***
teapot	sah	4	k-way	1.036	[1.035, 1.037]	1.73e-21***
armadillo	bsah	4	collapsed	1.036	[1.031, 1.043]	1.09e-06***
stanford-bunny	bsah	4	collapsed	1.045	[1.040, 1.053]	3.23e-07***
suzanne	bsah	4	collapsed	1.082	[1.077, 1.090]	3.73e-09***
teapot	bsah	4	collapsed	1.042	[1.037, 1.049]	5.79e-07***
armadillo	median	4	collapsed	1.050	[1.045, 1.058]	2.88e-07***
stanford-bunny	median	4	collapsed	1.057	[1.046, 1.068]	5.24e-09***
suzanne	median	4	collapsed	1.070	[1.065, 1.078]	3.65e-08***

Continued on next page

Tabelle 4.1: Statistical Analysis Results (k=4)

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
teapot	median	4	collapsed	1.069	[1.064, 1.078]	3.25e-08***
armadillo	sah	4	collapsed	1.032	[1.030, 1.034]	1.13e-14***
stanford-bunny	sah	4	collapsed	1.038	[1.036, 1.040]	5.44e-17***
suzanne	sah	4	collapsed	1.074	[1.072, 1.075]	9.59e-19***
teapot	sah	4	collapsed	1.036	[1.035, 1.037]	1.66e-19***

Tabelle 4.2: Statistical Analysis Results (k=8)

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
armadillo	bsah	8	k-way	0.646	[0.644, 0.651]	1.10e-19***
stanford-bunny	bsah	8	k-way	0.668	[0.665, 0.673]	1.46e-17***
suzanne	bsah	8	k-way	0.770	[0.766, 0.775]	2.59e-15***
teapot	bsah	8	k-way	0.712	[0.709, 0.717]	6.97e-17***
armadillo	median	8	k-way	0.633	[0.630, 0.638]	1.82e-17***
stanford-bunny	median	8	k-way	0.716	[0.712, 0.721]	2.15e-15***
suzanne	median	8	k-way	0.757	[0.753, 0.763]	6.21e-15***
teapot	median	8	k-way	0.704	[0.701, 0.710]	1.34e-16***
armadillo	sah	8	k-way	0.645	[0.644, 0.646]	2.96e-40***
stanford-bunny	sah	8	k-way	0.656	[0.655, 0.657]	1.15e-33***
suzanne	sah	8	k-way	0.751	[0.750, 0.752]	1.14e-34***
teapot	sah	8	k-way	0.705	[0.704, 0.706]	3.06e-37***
armadillo	bsah	8	collapsed	0.646	[0.643, 0.650]	1.74e-18***
stanford-bunny	bsah	8	collapsed	0.668	[0.663, 0.673]	7.15e-26***
suzanne	bsah	8	collapsed	0.768	[0.764, 0.775]	1.34e-20***
teapot	bsah	8	collapsed	0.711	[0.708, 0.716]	1.64e-16***
armadillo	median	8	collapsed	0.633	[0.630, 0.637]	3.06e-17***
stanford-bunny	median	8	collapsed	0.715	[0.712, 0.721]	3.32e-16***
suzanne	median	8	collapsed	0.758	[0.754, 0.764]	7.62e-15***
teapot	median	8	collapsed	0.704	[0.701, 0.710]	3.95e-16***
armadillo	sah	8	collapsed	0.645	[0.644, 0.646]	2.50e-34***
stanford-bunny	sah	8	collapsed	0.656	[0.655, 0.657]	4.46e-38***
suzanne	sah	8	collapsed	0.751	[0.750, 0.752]	3.19e-25***
teapot	sah	8	collapsed	0.706	[0.705, 0.707]	2.36e-33***

Tabelle 4.3: Statistical Analysis Results (k=16)

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
armadillo	bsah	16	k-way	0.098	[0.098, 0.099]	6.08e-38***
stanford-bunny	bsah	16	k-way	0.123	[0.123, 0.124]	4.96e-47***
suzanne	bsah	16	k-way	0.166	[0.165, 0.167]	2.04e-43***
teapot	bsah	16	k-way	0.158	[0.157, 0.159]	7.07e-44***

Continued on next page

Tabelle 4.3: Statistical Analysis Results (k=16)

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
armadillo	median	16	k-way	0.097	[0.096, 0.097]	3.42e-45***
stanford-bunny	median	16	k-way	0.138	[0.138, 0.139]	2.99e-45***
suzanne	median	16	k-way	0.169	[0.168, 0.170]	2.88e-36***
teapot	median	16	k-way	0.185	[0.184, 0.187]	3.51e-42***
armadillo	sah	16	k-way	0.097	[0.097, 0.097]	5.02e-27***
stanford-bunny	sah	16	k-way	0.120	[0.119, 0.120]	1.70e-27***
suzanne	sah	16	k-way	0.162	[0.162, 0.163]	7.05e-31***
teapot	sah	16	k-way	0.154	[0.154, 0.154]	1.03e-28***
armadillo	bsah	16	collapsed	0.099	[0.098, 0.099]	1.59e-34***
stanford-bunny	bsah	16	collapsed	0.123	[0.122, 0.124]	2.09e-30***
suzanne	bsah	16	collapsed	0.166	[0.166, 0.168]	1.21e-37***
teapot	bsah	16	collapsed	0.158	[0.157, 0.159]	2.76e-42***
armadillo	median	16	collapsed	0.097	[0.096, 0.097]	9.59e-43***
stanford-bunny	median	16	collapsed	0.138	[0.137, 0.139]	1.86e-40***
suzanne	median	16	collapsed	0.169	[0.168, 0.170]	1.91e-38***
teapot	median	16	collapsed	0.185	[0.184, 0.187]	2.17e-32***
armadillo	sah	16	collapsed	0.097	[0.097, 0.098]	9.06e-26***
stanford-bunny	sah	16	collapsed	0.119	[0.119, 0.120]	1.06e-31***
suzanne	sah	16	collapsed	0.162	[0.162, 0.163]	1.52e-36***
teapot	sah	16	collapsed	0.154	[0.154, 0.154]	6.23e-28***

4.3.1 Zusammenfassung

4.4 Dynamisches Modell

- Construction + Traversal Time

4.4.1 Zusammenfassung

5 Diskussion

tbc

6 Fazit

tbc Brueqidowk fwefe

7 Appendix