Bachelorarbeit

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

Nick Garczorz

3. Februar 2026

Heinrich-Heine-Universität Düsseldorf

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

Selbstständigkeitserklärung

Hiermit erkläre ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum Unterschrift

Zusammenfassung

Bounding Volume Hierarchies (BVHs) gehören zu den wichtigsten Beschleunigungsdatenstrukturen zur effizienten Berechnung von Ray-Szenen-Schnittpunkten in der Bildprojektion. Während Wide BVHs (Verzweigungsgrad $k > 2$) bereits im Zusammenhang mit Optimierung durch GPU-Berechnung untersucht werden, ist es weniger klar, ob sie auch einen praktischen Vorteil durch ihre Struktur bieten, die über die Parallelisierbarkeit hinausgeht. Falls auch eine Verbesserung bei serieller Berechnung auf der CPU gesehen werden kann, ergibt das neue Einblicke in die Optimierungsmöglichkeiten durch Wide BVH.

Diese Arbeit untersucht Top-Down konstruierte BVH in statischen dreidimensionalen Szenen für Verzweigungsgrade $k \in \{2, 4, 8, 16\}$. Dies geschieht unter Berücksichtigung von drei Splitting-Verfahren (Median, Surface Area Heuristic und Binned Surface Area Heuristic), sowie zwei Methoden zur Erzeugung von Wide BVHs (k-way splitting und binary BVH collapse). Dabei steht die Traversal-Time als primäres und die Construction-Time als sekundäres Qualitätsmerkmal im Fokus. Zudem wird durch eine Kombination der beiden Qualitätsmetriken ein Szenario mit dynamischer Szene modelliert.

Für alle untersuchten statischen Szenen zeigt $k = 4$ eine statistisch signifikante Verbesserung gegenüber der vergleichbaren binären Variante. Dabei liegt der Traversal-Speedup-Faktor im Mittel zwischen 1,03 und 1,08, wobei der Effekt für größere Szenen abzuflachen scheint. Für höhere Verzweigungsgrade ($k = 8$ und $k = 16$) liegt der Speedup-faktor deutlich und statistisch signifikant unter 1. Es konnte zwischen k-way splitting und der collapse Methode keine signifikanten Unterschiede in der Traversal-Time gefunden werden. Auch im dynamischen Modell scheint der durch $k = 4$ entstehende Vorteil mit steigender Szenengröße abzunehmen und kann bei den größeren untersuchten Szenen, unabhängig vom Konstruktionsverfahren, sogar ausbleiben bzw. zum Nachteil werden. Insgesamt deuten die Ergebnisse darauf hin, dass bei serieller Berechnung auf der CPU ein Verzweigungsgrad von $k = 4$ einen kleinen aber klaren Gewinn liefert. Größere Verzweigungsgrade sind nicht zu empfehlen. Das Verhalten in größeren Szenen bietet Raum für zukünftige Forschung.

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 [En2]. Im Fall von Animation und Bildgebung im Allgemeinen handelt es sich vor allem um Kollisionen einfacher geometrischer Formen, häufig Geraden oder Strahlen, 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 Primitives 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 [MTre1] 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 Primitives 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) [GCnr2]. 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 Primitives 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 typischerweise bei geeigneter Ray-Verteilung logarithmisch von der Anzahl an Primitives in der Szene abhängt, was die Zeit für ein fertig gerendertes Bild drastisch sinken lassen kann. Diesen Vorteil erkauft man sich durch das Einführen des zusätzlichen Konstruktionsschrittes, der in Abwägung dazu steht. 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

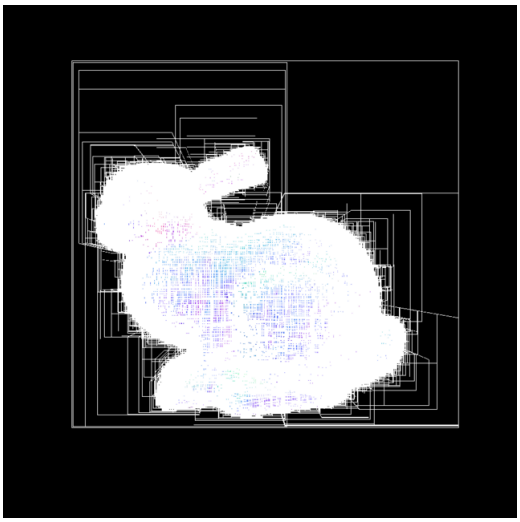

Abbildung 1.1: Stanford Bunny mit sichtbaren Bounding Boxes

eine BVH, da sich durch niedrigere Zeiten höhere Auflösungen und Bildwiederholraten, 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. Diese Art von BVH ist bereits in der parallelisierten Berechnung mit GPUs weiter verbreitet, hier wird allerdings eine serielle Berechnung auf der CPU untersucht und damit die algorithmischen Vor- und Nachteile von Wide BVH. 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 Umfang 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 $k \in \{2, 4, 8, 16\}$?

Die Analyse soll dabei auf einer selbstimplementierten Testbench stattfinden und anschließend mit unabhängigen T-Tests ausgewertet werden. Mittelwerte, sowie Konfidenzintervalle dienen zur späteren Diskussion der Ergebnisse.

1.4 Abgrenzung zu vorherigen Arbeiten

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

werden nicht untersucht.

Zudem werden ausschließlich Top-Down-Konstruktionsmethoden verwendet, da diese in Forschung und Lehre weit verbreitet und einfacher zu implementieren sind [Mr2].

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, da der Fokus auf Gewinn durch strukturelle Eigenschaften der Hierarchien liegen soll.

Die bei der Traversal-Time verwendeten Rays sind lediglich *Primärstrahlen*, also direkte Strahlen zwischen virtueller Kamera und Objekt. Dadurch fallen Verfahren wie Reflektion, 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.

Die Besonderheit dieser Arbeit ist, dass Wide BVH komplett außerhalb des Kontext der Parallelisierbarkeit analysiert werden und alle Berechnungen seriell auf der CPU ausgeführt werden. Es wird explizit die die gesamte serielle Zeit der Algorithmen untersucht, um Veränderungen durch Struktur der BVHs zu ermitteln.

Ziel ist es einzuschätzen, ob die Möglichkeit der parallelisierten Berechnung besser genutzt werden kann, durch z.B. mehr untersuchte Rays oder höhere Auflösungen, statt ineffiziente Erhöhung des Verzweigungsgrads.

1.5 Aufbau

Zuerst wird in die benötigen Grundlagen und Terminologie des Raytracings eingeführt. Darunter zählen Rays, AABBs und BVH-Konstruktion, sowie Traversal. Im folgenden werden die implementierten Algorithmen und der experimentelle Aufbau, wie auch das dazugehörige Datenmodell beschrieben. Nach der Vorstellung der Messergebnisse, werden diese anschließend diskutiert, Limitationen aufgezeigt und in den Literaturkontext einfügt. Zuletzt wird ein Fazit ausgeführt, welches die zentralen Erkenntnisse zusammenfasst.

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 *closest hit test* [Mr2].

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 [KKya1]. Hierbei wird ein Ray achsenweise mit den durch die AABB definierten Begrenzungsebenen getestet. Für jede Raumachse existieren paarweise zwei parallele Ebenen, die eine obere und untere Schranke darstellen. Für jede Achse wird ein Parameterintervall für die Ein- und Austrittsparameter des Rays bestimmt. Eine Kollision liegt nur dann vor, wenn sich die Intervalle über alle Achsen überlappen.

2.3 Bounding Volume Hierarchy

Eine BVH ist im Kern eine Baumstruktur, die eine sinnvolle Partitionierung der Primitives 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 konzeptionell Nodes, deren Menge an Primitives kleiner gleich dem festgelegten Verzweigungsgrad sind.

Qualitätsmerkmale einer BVH sind in etwa wenig Überschneidungen von AABBs, AABBs mit möglichst kleinen Flächeninhalten und möglichst wenig Kollisionsüberprüfungen mit Primitives [AKLase2].

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 untersuchten Algorithmen immer eine Referenzmethode zur Erstellung von Binärbäumen. Die untersuchten Verzweigungsgrade sind vielfache von Zwei. Dadurch wird die Vergleichbarkeit zwischen den verschiedenen Konstruktionsmethoden gewährleistet, da das collapse-Verfahren in dieser Implementierung 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 Hauptaufgabe. 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 Tiefensuche 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 Primitives 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* [MBrr2]. 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 gemessenen 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 Primitives 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)* [Mr2], um eine Auswahl zu nennen, deren Untersuchung nicht Teil dieser Arbeit ist.

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 Primitives am Median geteilt. Das wird rekursiv für die neu entstandenen Teilmengen wiederholt, bis ein festgelegtes Abbruchkriterium erreicht wird. Sie basiert damit nicht auf einem Kostenmodell und eignet sich vor allem wegen ihrer Einfachheit für den Vergleich. Sie besitzt *de facto* den kleinsten Overhead für die Konstruktion, da keine Abwägung der Teilmengen nötig ist. Die Idee ist, dass dadurch strukturelle Änderungen der BVH klarer herausstechen.

Surface Area Heuristic Split

Dagegen stehen Traversal-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 [MBrr2]. Daraus ergibt sich, dass kleinere Flächeninhalte die Kostenfunktion minimieren, also als besser gesehen werden. Die SAH berücksichtigt dabei nicht die tatsächliche Ausrichtung und verteilung von Rays.

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

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 Konstanten, 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, allerdings 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 Heuristic Split* und *Binned Surface Area Heuristic Split*. Als Referenz 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. Jeder erzeugt Frame hat eine Auflösung von $500 * 500$ Pixeln, was einer Menge von 250.000 Primärstrahlen pro berechneten Bild bzw. Traversalschritt entspricht. 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 stets 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. Alle Modelle verwenden Lizenzen zu freien Benutzung [Br; Nl; Ly] 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 sichergestellt werden, dass die untersuchte BVH eine gültige Form besitzt. Unabhängig vom Testrun sollen die erzeugten Bilder pro Testszene immer gleich sein.

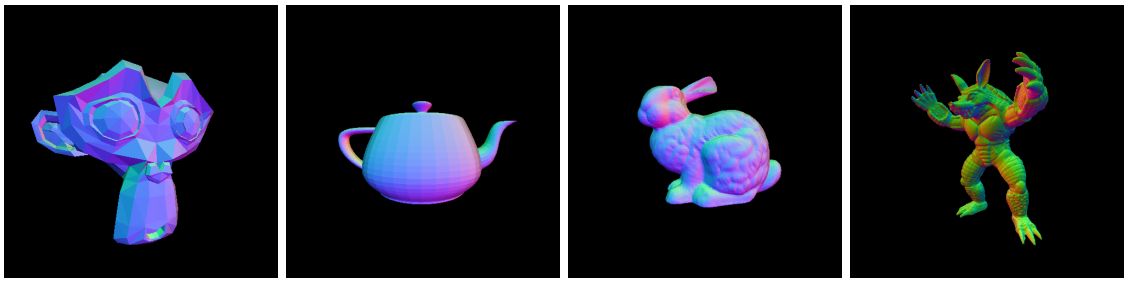

Abbildung 3.1: Suzanne, Utah Teapot, Stanford Bunny. Stanford Armadillo

3.1.2 Reproduzierbarkeit

Die verwendete Testbench hat folgende Systemeigenschaften:

- CPU: AMD Ryzen 7 7800X3D
- RAM: 32 GB DDR5 6000 MHz
- OS: Ubuntu 22 (WSL)
- Compiler: GCC
- Timer: C++ Standard Library (std::chrono)

Das System war während der Experimente nach besten Möglichkeiten von äußeren Störfaktoren getrennt und lief offline.

3.1.3 Datenmodell

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

Die implementierte BVH verwendet nicht direkt Polygone, sondern ist in der Lage Primitives 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 Aligned Bounding Box

Die BVH selbst besteht aus aufeinander referenzierende Nodes. Ein Node wird dabei durch seine Bounding Box, Referenzen zu den eingeschlossenen Primitives 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
    Primitive [] 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 Primitives 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 den Fall, dass es sich um einen Leaf-Node handelt, werden die Primitives 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
```

```
15              for primitive in node.primitives do
16                  intersection := primitive.intersect(ray)
17                  if intersection then
18                      distance := distance(ray.origin, intersection)
19                      if distance < closestDistance then
20                          closestDistance := distance
21                      end if
22                  end if
23              end for
24          end if
25
26          for child in node.children do
27              push(stack, child)
28          end for
29      end while
30      return closestDistance
31
32  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 der Primitives für die BVH erstellt und sichergegangen wird, dass überhaupt Primitives 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 Primitives des Nodes und der Achse aufgerufen. Die daraus resultierenden Indizes werden verwendet um die Primitives aufzuteilen. Falls keine Indizes 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 Primitives zurückgegeben.

```
1  function build(inputPrimitives, partitionFunction):
2      // move input primitives into a single owned list inside the BVH
3      ownedPrimitives := copy of inputPrimitives
4
5      if ownedPrimitives is empty then
6          return BVH(root = emptyRoot, primitives = ownedPrimitives)
7      end if
8
9      // compute global bounds of all primitives
10     box := computeBoundingBox(ownedPrimitives)
11
12     // create BVH with a single root node
13     rootNode := BVHNode(box       = box,
14                         primitives = ownedPrimitives
15                         children   = empty list)
16     bvh      := BVH(root = rootNode, primitives = ownedPrimitives)
17
18     // stack of nodes to process
19     nodes := empty list
20     push(nodes, bvh.root)
21
22     while nodes is not empty do
23         node := pop(nodes)
24
```

```
25          // choose splitting axis based on longest extent
26          axis := calculateLongestAxis(node.box)
27
28          // Find right subsets according to partitionFunction
29          splitIndices      := partitionFunction(primitives, axis)
30          if splitIndices is empty then
31              continue //Leaf Node
32          end if
33          primitiveSubsets  := calculateSubsets(primitives, splitIndices)
34
35          for each primitiveSubset in primitiveSubsets do
36              boundingBox  := computeBoundingBox(primitiveSubset)
37              child        :=  BVHNode(box       = boundingBox,
38                                       primitives = primitiveSubset
39                                       children   = empty list)
40              append child to node.children
41          end for
42
43          // push children onto stack for further splitting
44          for each child in node.children do
45              push(nodes, reference to child)
46          end for
47      end while
48
49      return bvh
50  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 Primitives wird in, vom Verzweigungsgrad abhängige, gleichgroße Teile gespalten. Dafür wird die Menge der Primitives n durch den Verzweigungsgrad geteilt um die Größe einer Teilmenge zu bestimmen. Vielfache dieser Zahl innerhalb des Intervalls $(0, n)$ werden als Indizes für das Splitting verwendet. Allerdings muss dafür noch die Menge an Primitives, entlang der gegebenen Achse, partiell Sortiert werden, sodass für alle Indizes gilt, dass der Mittelpunkt der Position der Primitives links des Index kleiner sind als die rechts des Indexes.

```
1   function medianSplit(primitives, axis, degree):    //O(n)
2       count := primitives.size()
3       splitIndices := empty list
4
5       for i in degree do
6           split := (count * i) / degree
7           if split == 0 or split >= count then
8               break
9           end if
10          push(splitIndices, split)
11      end for
12
13      // partially sort so that element at split is in its median position in O(n)
14      for split in splitIndices do
15          nth_element(set    = primitives,
16                      middle = split,
17                      compare(primitiveA, primitiveB):
18                          return center(primitiveA).component(axis) <
        center(primitiveB).component(axis))
19      end for
20      return splitIndices
21  end function
```

Listing 3.6: Median Partition

Surface Area Heuristic Split

Beim SAH-Split wird zunächst die Menge an behandelten Primitives entlang der längsten Achse nach Position der Centroids sortiert. Vorweg werden für jeden möglichen Split-Index i die Bounding Boxes der linken und rechten Teilmenge berechnet. Für eine vereinfachte Kostenfunktion $c(i)$ ergibt sich daraus:

$$c(i) = Area(\text{AABB}(P_L)) \cdot |P_L| + Area(\text{AABB}(P_R)) \cdot |P_R|$$

Dabei beschreibt $Area(\cdot)$ den Oberflächeninhalt der AABB der Teilmenge. Gewählt wird Split-Index i für den $c(i)$ minimal ist. Das errechnen der kosten geschieht in $O(n)$

Für Wide BVHs wird dieser Vorgang greedily mehrfach wiederholt, bis der gewünschte Verzweigungsgrad erreicht ist. Dafür wird das Segment, bzw. die Teilmenge an Primitives für den Algorithmus berücksichtig, welches die aktuell höchsten SAH-Kosten entstehen lässt. Insgesamt entstehen dadurch $k-1$ Split-Indizes, die k Teilmengen definieren.

```
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 // default
        minCost := +∞
        for i from 1 to segCount - 2 do
            leftArea  := surfaceArea(prefixBoundingBoxes[i - 1])
            rightArea := surfaceArea(suffixBoundingBoxes[i])
            cost      := leftArea * i + rightArea * (segCount - i) //simplified cost function
            if cost < minCost then
                minCost := cost
                split   := i
            end if
        end for

        push(splitIndices, seg.begin + split)   //convert relative split to absolute
        erase(segments, seg)
        //push subsets
        push(segments, seg[seg.begin, split])
        push(segments, seg[split, seg.end])
    end do
    return splitIndices
end function
```

Listing 3.7: Surface Area Heuristic Partition

Binned Surface Area Heuristic Split

Der Binned-SAH-Split ist eine approximation des vorher beschriebenen SAH-Splits. Die Idee ist es statt jeden Punkt zwischen Primitives als Split-Index in betracht zu ziehen, zuerst die Primitives in $b = 16$ Bins entlang der Achse einzuteilen und dann Splits nur zwischen den Bins zuzulassen. Jeder Bin speichert die Anzahl an enthaltener Primitives und die zusammengefassten Bounds. Wieder werden die die Oberflächeninhalte und Menge an Primitives für jeden der möglichen der b Splits vorweg berechnet. Damit sind die Kosten:

$$c(j) = Area(\text{AABB}(Bins_{\leq j})) \cdot |P_{\leq j}| + Area(\text{AABB}(Bins_{>j})) \cdot |Bins_{>j}|$$

Die Kosten für alle möglichen Splits können in diesem Fall in $O(b)$ statt $O(n)$ ausgewertet werden. Analog zum SAH-Split wird dieser Vorgang greedily wiederholt und das teuerste Segment gesplittet. Zuletzt werden die errechneten Split-Indizes für die Bins in absolute Indizes umgerechnet und zurückgegeben.

```
1   function binnedSahSplit(primitives, axis, degree):   //O(n * log(n))
2
3       Bin_SIZE := 16  // b
4
5       count := primitives.size()
6       if isLeaf(count, degree) then
7           return {}
8       end if
9
10      //O(n * log(n))
11      sort(set = primitives,
12           compare(primitiveA, primitiveB):
13               return center(primitiveA).component(axis) < center(primitiveB).component(axis))
14
15      struct Bin {count; bounds}
16
17      bins := sortPrimitivesIntoBins(primitives, BIN_SIZE) //O(n)
18
19      splitIndices := empty list
20      segments := empty list
21      push(segments, bins) // [0, BIN_SIZE)
22      while segments.size() < degree do
23
24          seg := findSegmentToSplitGreedily(segments)   //highest cost segment has to be split
25          // bounds of primitives from segment from begin to index
26          prefixBoundingBoxes := calculatePrefix(seg)  //O(b)
27          // bounds of primitives from segment from index to end (included)
28          suffixBoundingBoxes := calculateSuffix(seg) // O(b)
29
30
31          segCount := seg.count()
32          split   := segCount / 2 // default
33          minCost := +∞
34          for i from 1 to segCount - 2 do
35              leftArea  := surfaceArea(prefixBoundingBoxes[i - 1])
36              rightArea := surfaceArea(suffixBoundingBoxes[i])
37              cost      := leftArea * i + rightArea * (segCount - i) //simplified cost function
38              if cost < minCost then
39                  minCost := cost
40                  split   := i
41              end if
42          end for
43
44          absoluteSplit := calculateAbsoluteSplit(bins, split)
45          push(splitIndices, absoluteSplit)
46          erase(segments, seg)
47          //push subsets
48          push(segments, seg[seg.begin, split])
49          push(segments, seg[split, seg.end])
50
```

```
51          end do
52          return splitIndices
53      end function
54
```

Listing 3.8: Binned Surface Area Heuristic Partition

3.2.3 Collapse

Für das collapse Verfahren wird eine bereits generierte BVH verwendet. Es wird mit einer binären BVH gestartet. Mit jeder Ausführung der Funktion wird der Verzweigungsgrad verdoppelt. Das bedeutet einmalige Ausführung für $k = 4$, zweimalige für $k = 8$ und dreimalige für $k = 16$. Es wird wieder mit einem Stack gearbeitet, angefangen bei der Root-Node. Für jedes Element auf dem Stack gilt: Falls der behandelte Node über Child-nodes verfügt, wird für jeden Child-Node überprüft, ob diese selbst Child-Nodes besitzen. Falls ja werden die Grandchild-Nodes zu den neuen Child-Nodes des behandelten Nodes. Diese werden zuletzt auf den Stack gelegt um in einem späteren Iterationsschritt behandelt zu werden.

```
1       function collapse(bvh):
2
3           // stack of nodes to process
4           nodes := empty list
5           push(nodes, bvh.root)
6
7           while nodes is not empty do
8               node := pop(nodes)
9
10              if node.children is empty do
11                  continue
12              end if
13
14              new_children := empty list
15
16              for each child in node.children do
17                  if child.children is empty do
18                      push(new_children, child)
19                      continue
20                  end if
21                  push(new_children, child.children)
22              end do
23
24              node.children := new_children
25              push(nodes, node.children)
26          end while
27
28          return bvh
29      end function
30
```

Listing 3.9: BVH-Collapsing

4 Ergebnisse

4.1 Korrektheit

Um sicherzustellen, dass der BVH-Vergleich nicht durch falsche bzw. ungleiche Ausgaben verfälscht wird, wurde über alle Konfigurationen hinweg ein aus Primärstrahlen berechnetes Bild erzeugt. Die dabei entstandenen Bilder sind identisch. Für jede Szene und Kameraposition wurden die generierten Bilder mit denen der anderen Konfigurationen Bitweise verglichen, um eine gleiche Ausgabe zu garantieren.

Außerdem wurde die Anzahl an *Hitrays* pro Frame erfasst. Ein Hitray beschreibt dabei einen Primärstrahl, der mindestens ein Primitive in der Szene geschnitten hat. Auch die Anzahl an Hitrays ist über alle Konfigurationen pro Szene und Frame identisch, womit sichergestellt werden kann, dass über alle Konfigurationen auch die gleiche Menge an Traversalschritten erfolgreich war.

4.2 Traversal

Die erste Forschungsfrage beschäftigt sich damit, wie sich die Traversal-Time Top-Down konstruierter Wide BVHs abhängig vom Verzweigungsgrad $k \in \{2, 4, 8, 16\}$ verändert. Hierfür dient als Referenz für jede Szene und jedes Splitting-Verfahren eine binäre BVH ($k = 2$). Zum Zweck der Vergleichbarkeit wird das Feld *Speedup* definiert:

$$Speedup = \frac{T_{\mu,\text{traversal}(k=2)}}{T_{\mu,\text{traversal}(k)}}$$

Dabei bezeichnet $T_{\mu,\text{traversal}(k)}$ den Mittelwert der, wie im Kapitel Methoden gezeigt, aggregierten gemessenen Traversal-Times, abhängig vom Verzweigungsgrad k. Werte $Speedup > 1$ gelten als Verbesserung, also geringerer Traversal-Time, gegenüber der binären Referenz.

Die Abbildung 4.1 zeigt den Traversal-Speedup für $k \in \{4, 8, 16\}$ und Tabelle 7.1 enthält die dazugehörigen Mittelwerte, Konfidenzintervalle und p-Werte für jede Kombination von Szene, Splittingverfahren und Wide BVH Methode. Für $k = 4$ gibt es über alle Kombinationen eine konsistente und statistisch signifikante Verbesserung gegenüber der binären Referenz. Die Mittelwerte der Speedups liegen zwischen ungefähr $1,03$ und $1,08$. Für $k = 8$ verschlechtert sich die Traversal-Time in allen untersuchten Fällen klar mit Speedup-Mittelwerten zwischen ungefähr $0,63$ und $0,77$. Für $k = 16$ fällt die Performance nochmal deutlich ab mit Mittelwerten zwischen $0,10$ und $0,19$, was einer stark höheren Traversal-Time gegenüber $k = 2$ entspricht.

4.3 Wide BVH Konstruktionsmethode

Die zweiten Forschungsfrage bezieht sich darauf, inwiefern sich die Traversal-Time zwischen den beiden Untersuchten Verfahren zur Erzeugung einer Wide BVH unterscheidet. Zum Einen das direkte k-way-Splitting und zum Anderen das collapsing einer vorher konstruierten binären BVH. Wieder erfolgt der Vergleich über jede Szene und jedes Splittingverfahren. Speedup hat an dieser Stelle nicht $k = 2$, sondern die collapse-Methode als

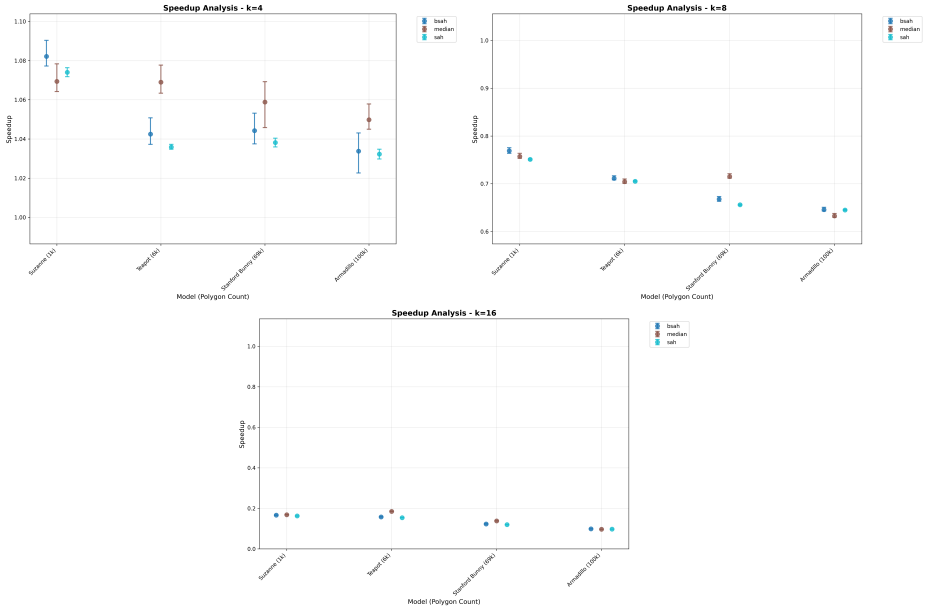

Abbildung 4.1: Abbildung Traversal-Speedup für statische Szenen

Referenz. Damit definiert sich ein neuer Vergleichswert:

$$Speedup_{\text{comparison}} = \frac{T_{\mu,\text{traversal}(k,collapse)}}{T_{\mu,\text{traversal}(k,k\text{-}way)}}$$

Wieder bezeichnet $T_{\mu,\text{traversal}}$ den Mittelwert der, wie im Kapitel Methoden gezeigt, aggregierten gemessenen Traversal-Times, diesmal allerdings abhängig von der gewählten Methode zur Erzeugung der Wide BVH bei gleichem Verzweigungsgrad k. Es gilt, dass $Speedup_{\text{comparison}} > 1$ bedeutet, dass k-way splitting eine schnellere Traversal-Time besitzt als collapsing, während $Speedup_{\text{comparison}} < 1$ auf einen Vorteil für collapsing hinweist.

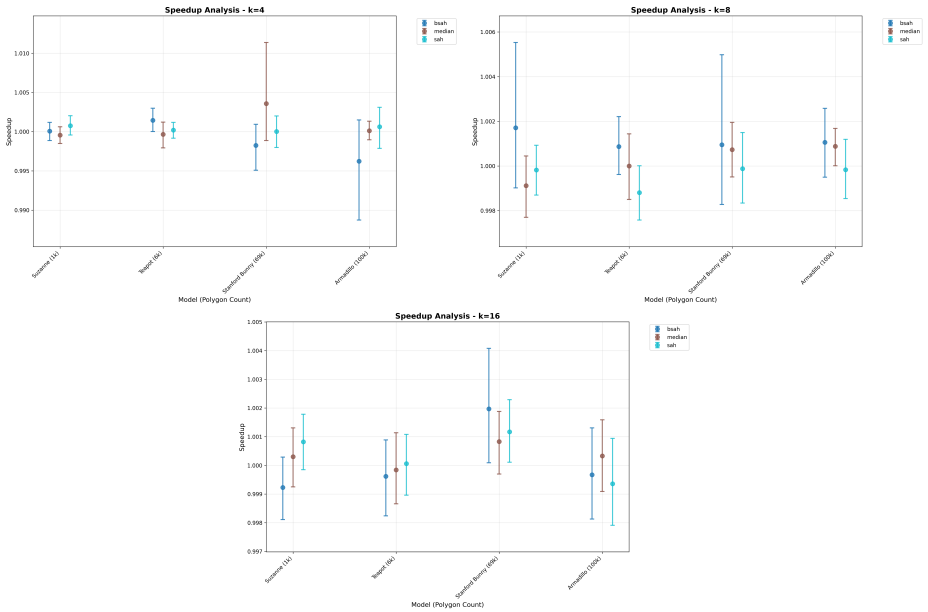

Abbildung 4.2: Abbildung Traversal-Speedup collapse- gegen k-way- Verfahren

Die Abbildung 4.2 zeigt den Traversal-Speedup für $k \in \{4, 8, 16\}$ und Tabelle 7.2 enthält die dazugehörigen Mittelwerte, Konfidenzintervalle und p-Werte für jede Kombination von Szene, Splittingverfahren und Wide BVH Methode. Im Vergleich liegen die Speedup-Werte für alle Kombinationen Szenen und Splitting-Verfahren für alle $k \in \{4, 8, 16\}$ jeweils sehr nah bei 1. Die Konfidenzintervalle reichen von $0,996$ bis $1,004$, beinhalten dabei aber immer den Wert 1. Dadurch lässt sich für die untersuchten statischen Szenen kein statistisch signifikanter und damit praktisch nutzbarer Unterschied in der Traversal-Time zwischen direktem k-way-Splitting und der collapse-methode feststellen. Dies gilt unabhängig von untersuchter Szene oder gewählten Splitting-Verfahren.

4.4 Dynamisches Modell

Die dritte und letzte Forschungsfrage untersucht die Modellierung einer dynamischen Szene, bei der die (Re-)Construction der Szene vor jeder Traversal-Phase erforderlich ist. Für die Modellierung wird eine kombinierte Qualitätsmetrik aus Construction- und Traversal-Time verwendet.

$$T_{dynamic}(k) = T_{\mu,\text{construction}(k)} + T_{\mu,\text{traversal}(k)}$$

Dabei ist $T_{\mu,\text{construction}(k)}$ den Mittelwert der Construction-Time und $T_{\mu,\text{traversal}(k)}$ den Mittelwert der Traversal-Time, wie im Kapitel Methoden beschrieben. Speedup wird analog zur ersten Forschungsfrage gegenüber $k = 2$ definiert:

$$Speedup_{dynamic} = \frac{T_{\mu,\text{dynamic}(k=2)}}{T_{\mu,\text{dynamic}(k)}}$$

Ebenso entspricht $Speedup_{dynamic} > 1$ einer Verbesserung der kombinierten Metrik.

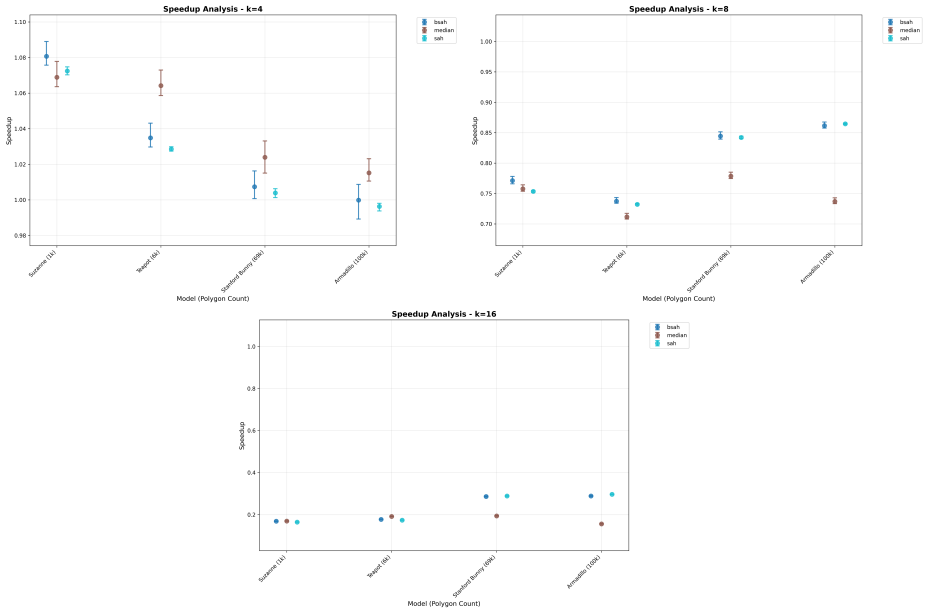

Abbildung 4.3: Abbildung kombinierter Speedup für Construction- und Traversal-Time

Die Abbildung 4.3 zeigt den Traversal-Speedup für $k \in \{4, 8, 16\}$ und Tabelle 7.3 enthält die dazugehörigen Ergebnisse. Für $k = 4$ gibt es in kleineren Szenen einen Vorteil gegenüber $k = 2$, der sich durch Speedup-Mittelwerten zwischen $1,03$ und $1,08$ ausdrückt. Für größere Szenen nimmt dieser Vorteil ab, womit der Speedup nahe an 1 liegt, oder sogar

teilweise leicht darunter fällt. Für $k = 8$ ist der Speedup für das dynamische Modell wieder deutlich unter 1 mit Mittelwerten, die sich typischerweise im Bereich von $0,71$ bis $0,86$ aufhalten. Für $k = 16$ gibt es weitere Geschwindigkeitseinbußen mit Mittelwerten für den Speedup zwischen $0,16$ und $0,30$. Zusammengefasst zeigt das dynamische Modell ähnliche Ergebnistrends, wie im Rahmen der ersten Forschungsfrage gezeigt. Insgesamt bleibt die BVH mit $k = 4$ die Performanteste Datenstruktur für die untersuchten Szenen, auch wenn ein Abfall für größere Szenen erkennbar ist.

5 Diskussion

5.1 Interpretation: Einfluss des Verzweigungsgrades auf die Traversal

Vorher wurde gezeigt, dass es Einsparung in der Traversal-Time für $k = 4$ gegenüber der binären Referenz gab, während es für $k \in \{8, 16\}$ starke Einbußen in der Performance gab.

Eine plausible Erklärung für die beobachteten Ergebnisse ist ein Abwägung zwischen der Anzahl an inneren Knoten und der Anzahl an auszuführenden AABB-Tests bzw. Kollisionsüberprüfungen zwischen Primärstrahl und Bounding Boxen. Bei höheren Verzweigungsgraden müssen nämlich pro Node mehr Kinder überprüft werden, während es insgesamt aber auch weniger innere Nodes gibt, die überhaupt überprüft werden können. Es lässt sich interpretieren, dass für $k = 4$ die Menge an weniger besuchten Nodes die größere Menge an AABB-Tests noch überkompensiert, während dies ab $k = 8$ nicht mehr der Fall ist. Zur Veranschaulichung dient Abbildung 5.1 in der man Beispielhaft die Menge an AABB-Tests und Triangle-Tests, bzw. Primärstrahl und Polygon Kollisionen, ablesen kann. Zudem ist wichtig, dass der für $k = 4$ beobachtete Effekt zwar statistisch signifikant

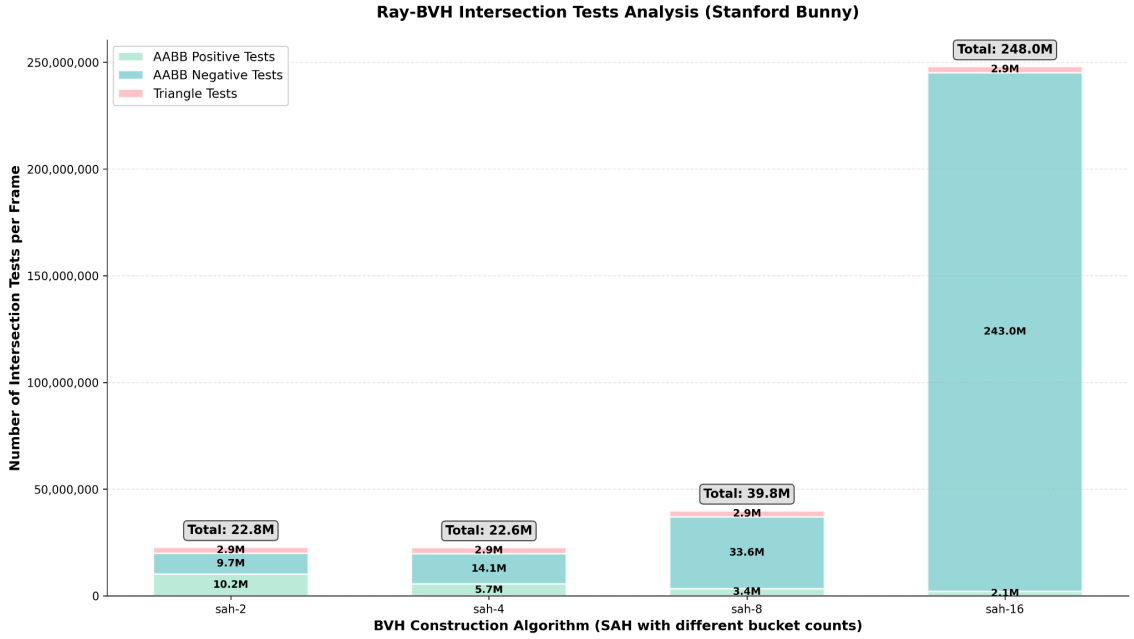

Abbildung 5.1: AABB- und Triangle-Tests, Stanford Bunny, SAH

ist, aber trotzdem nur einen Vorteil im einstelligen Prozentbereich liefert. Das heißt, dass Wide BVH in diesem linearen Zusammenhang eher als Optimierung mit kleinem Effekt zu sehen ist.

5.2 Interpretation: K-way splitting gegen collapsing

Im Rahmen der Ergebnissvorstellung wurde gezeigt, dass die Messergebnisse unabhängig von der gewählten Methode zur Erstellung der Wide BVH ähnliche Messungen ergaben.

Die untersuchten Messergebnisse zeigten keinen signifikanten Unterschied zwischen den Traversal-Times für die beiden Methoden.

Die naheliegendste Erklärung ist, dass die räumliche Partitionierung für beide verfahren sehr ähnlich ist. Vor allem deshalb, da beiden Methoden die gleichen Splitting-Algorithmen zugrunde liegen. Das zusammenfassen der binären BVH durch collapsing verändert zwar die Tiefe des Baumes, allerdings scheint es, dass sich die Wahrscheinlichkeit für die Kollision zwischen Ray und AABB nicht sonderlich zu der durch k-way splitting erzeugten BVH unterscheidet. In Abbildung 5.2 lässt sich anhand eines Beispiels ablesen, dass die Menge an AABB-Tests, positiv wie negativ, unabhängig vom gewählten Verfahren in der gleichen Größenordnung liegen. Für das Beispiel wurden wieder Stanford Bunny und SAH gewählt. Eine Konsequenz dieser Ergebnisse ist, dass die Wahl des Wide BVH Konstrukti-

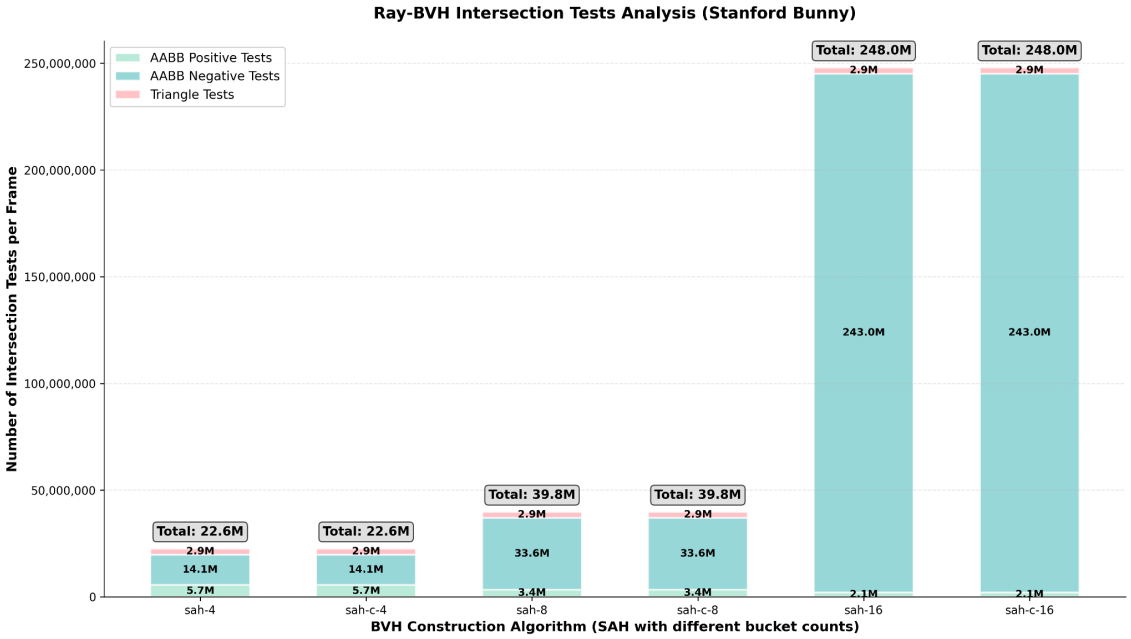

Abbildung 5.2: AABB- und Triangle-Tests k-way vs. collapse, Stanford Bunny, SAH

onsverfahren nach Implementierungsaufwand oder Construction-Time entschieden werden sollte, da es keine signifikanten einbußen durch die Wahl der untersuchten Verfahren gibt.

5.3 Interpretation: Dynamisches Modell

Für das dynamische Modell wurde gezeigt, dass auch hier $k = 4$ überwiegend bessere Ergebnisse gegenüber der binären Referenz aufzeigt, der Effekt allerdings mit steigender Szenengröße zu sinken bzw. sich sogar leicht umzukehren scheint. Für $k = 8$ und $k = 16$ wurden klar schlechtere Ergebnisse gezeigt.

Der Unterschied gegenüber der Ergebnisse der ersten Forschungsfrage ist dadurch erklärbar, dass die die Construction-Time über die verschiedenen Verzweigungsgrade hinweg für jeden Splitting-Algorithmus nahezu konstant zu sein scheint, wie man auch in Abbildung 5.3 ablesen kann. Die Vorteile in der Traversal-Time werden damit durch einen konstanten offset verschlechtert, was zur Folge hat, dass der Speedup für größere Szenen gegen 1 geht. Vorsichtig kann man beurteilen, dass der Effekt für größere Szenen zu einem immer größer werden Problem wird, sodass der Vorteil von $k = 4$ immer weiter schwindet oder möglicherweise sogar zum Nachteil wird. Allerdings ist die untersuchte Stichproben-

größe an Szenen nicht ausreichend um hierfür finale Aussagen liefern zu können, weshalb man hier von Indizien sprechen muss.

5.4 Literaturkontext

In der Literatur werden Wide BVH eher im Kontext mit der Nutzung von SIMD oder GPUs erwähnt, da höhere Verzweigungsgrade Chancen für die bessere Parallelisierbarkeit bieten [WBBdns2]. Der beobachtete Einbruch des Speedups für größere k ist daher mit der Literatur konsistent, da in diesem Falle der erhöhte Aufwand durch mehr AABB-Tests nicht durch parallele Berechnung amortisiert werden kann. Dadurch erscheint es so, dass ein kleinerer Verzweigungsgrad wie $k = 4$ als Optimum für die untersuchten Szenen glaubhaft wirkt.

5.5 Limitationen

Alle besprochenen Ergebnisse gelten innerhalb der folgenden Annahmen:

- Statische 3D Szenen
- Kamera außerhalb der Szenen
- Ausschließlich Untersuchung von Primärstrahlen
- Serielle Berechnung auf der CPU ohne spezielle Raytracing-Hardware
- Ausschließlich Top-Down-Konstruktion von BVH
- Keine echten dynamischen Verfahren, wie z.B. Refitting oder Rebuilding

Darüber hinaus sorgt die Menge an lediglich vier untersuchten Szenen, sowie ihre ähnliche Beschaffenheit für eine begrenzte Aussagekraft. Die Verteilung der untersuchten Rays ergibt sich aus der vorgegebenen Kamerafahrt um den Ursprung der Szene und nicht aus der Untersuchung von Sekundärstrahlen durch z.B. die Berechnung von Schatten. Metriken wie AABB- und Triangle-Tests wurde nur für ausgewählte Beispiele erhoben und bieten keine Grundlage für eine ausgiebige Analyse.

5.6 Ausblick

Fortlaufend bietet es sich an die Stichprobe an Szenen zu erweitern. Mehr und vor allem größere Szenen, sowie komplexere Szenen mit Kameraursprung innerhalb der Szene bieten Forschungsspielraum. Zudem wäre eine Untersuchung, die nicht nur Primärstrahlen überprüft, sondern auch Shadingeffekte, durch z.B. *Ambient Occlusion*, denkbar. Auch wäre eine quantitative Untersuchung der AABB- und Triangle-Tests sinnvoll, um die hier genannten Erklärungen zu untermauern.

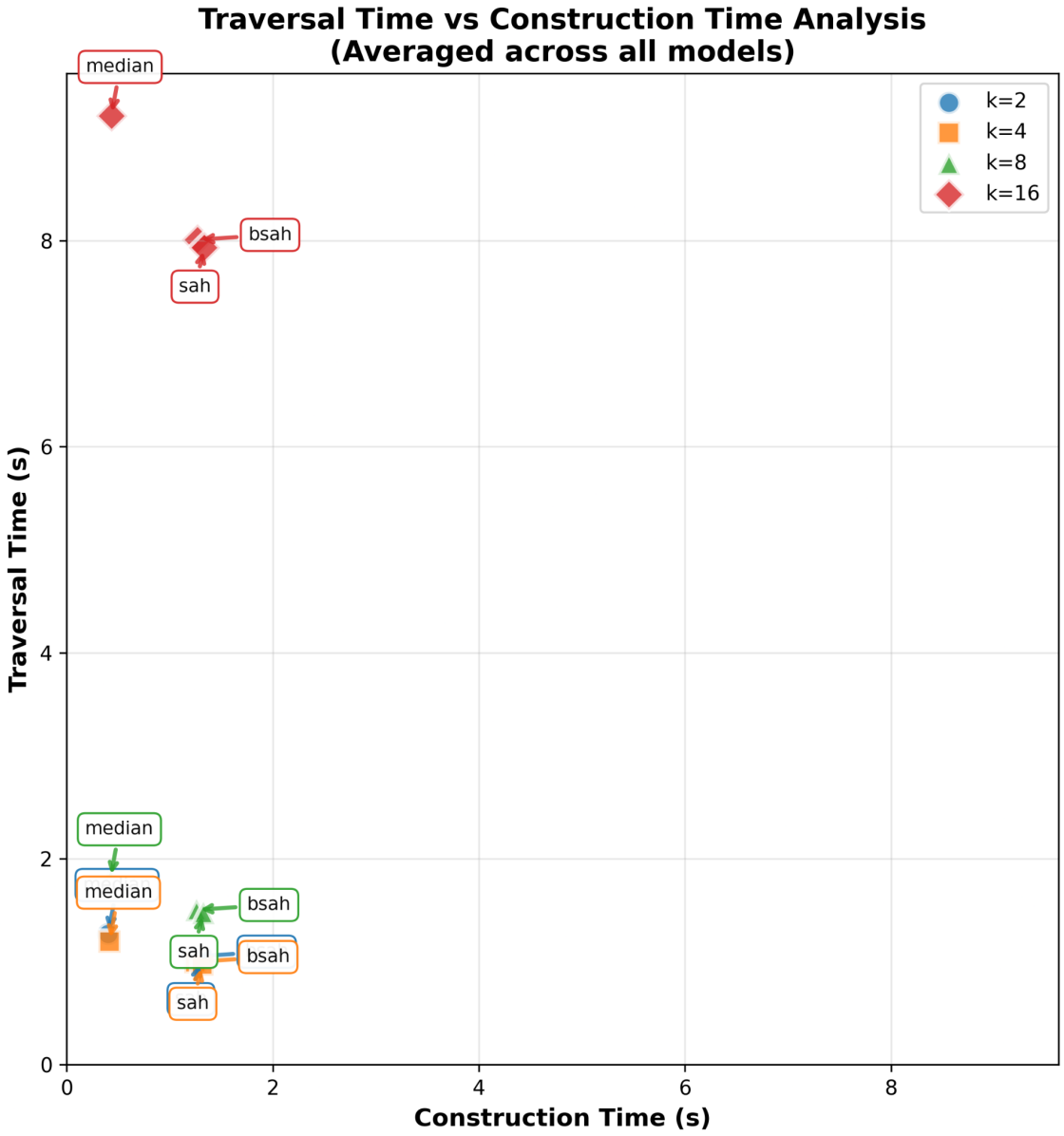

Abbildung 5.3: Vergleich Construction- gegen Traversal-Time

6 Fazit

Der Fokus dieser Arbeit war die analytische Bewertung der Qualität von Top-Down konstruierten Wide BVHs in statischen 3D Szenen aufbauend auf serieller Berechnungen. Dabei gab es das Ziel herauszufinden, ob Wide BVH alleinig auf struktureller bzw. algorithmischer Ebene einen Vorteil gegenüber binärer BVHs innehaben, auch wenn der Vorteil der parallelisierten Berechnung wegfällt.

Im Rahmen der ersten Forschungsfrage zeigten die Messungen einen statistisch signifikanten, wenn auch kleinen Vorteil durch Verwendung einer BVH mit Verzweigungsgrad $k = 4$, gegenüber $k = 2$, in statischen Szenen. Für alle untersuchten Szenen und Algorithmen liegt der Speedup zwischen $1,03$ und $1,08$ für Mittelwerte. Die Qualität der BVH nimmt für $k \geq 8$ deutlich ab mit Speedup-Mittelwerten von $0,63$ bis $0,77$ für $k = 8$ und fällt für $k = 16$ sogar auf Werte zwischen $0,10$ und $0,19$. Damit lässt sich für das gewählte Szenario unter gleichen Voraussetzungen eine Empfehlung für die Nutzung von Wide BVH mit $k = 4$ aussprechen.

Die Untersuchung der zweiten Forschungsfrage hat ergeben, dass es keinen praktischen Unterschied zwischen der Nutzung der untersuchten Arten der Erzeugung von Wide BVHs in den untersuchten Szenen gibt. Über alle Konfigurationen hinweg ist der Speedup zwischen k-way splitting und collapsing ohne signifikante Abweichung nahe 1. Damit ist die Empfehlung die gewählte Methode nach Implementierungsaufwand und Construction-Time zu wählen.

Für die Modellierung einer dynamischen Szene durch eine kombinierte Metrik aus Construction- und Traversal-Time im Rahmen der letzten Forschungsfrage, bleiben die Ergebnisse ähnlich zur ersten Frage. Verzweigungsgrad $k = 4$ liefert insgesamt die besten Ergebnisse, während $k \geq 8$ auch in diesem Fall schlechtere Metriken liefern. Zudem lassen sich Indizien finden, dass der Vorsprung von $k = 4$ mit steigender Szenengröße abzunehmen scheint bzw. sogar komplett ausbleiben oder sich umkehren kann. Damit kann man sagen, dass auch in diesem Zusammenhang zumindest für die kleineren Untersuchten Szenen ein Vorteil für BVHs mit Verzweigungsgrad $k = 4$ existiert.

Zusammengefasst lässt sich sagen, dass falls Wide BVHs in der seriellen Berechnung auf der CPU eingesetzt werden, man sich für einen Verzweigungsgrad von $k = 4$ entscheiden sollte, während Verzweigungsgrade $k \geq 8$ vermieden werden sollten. Ein wichtiger nächster Schritt wäre die Messungen mit einer größeren Menge an Szenen mit höherer Polygonanzahl durchzuführen, um die Indizien eines abnehmenden Vorteils zu untersuchen und belegbare Aussagen treffen zu können. Zudem wäre es interessant das Verhalten der BVHs unter der Nutzung von anderen Ray-Verteilungen durch beispielsweise Shader-Berechnung mit Hilfe von Sekundärstrahlen oder Ähnlichem zu beleuchten.

7 Appendix

Tabelle 7.1: Statistical Analysis Results

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***
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***
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***

Continued on next page

Tabelle 7.1: Statistical Analysis Results

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
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***
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***
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***

Tabelle 7.2: Statistical Analysis Results

Model	Algorithm	k	Speedup μ	Speedup CI	p
armadillo	bsah	4	0.996	[0.989, 1.002]	3.17e-01
stanford-bunny	bsah	4	0.998	[0.995, 1.001]	2.82e-01
suzanne	bsah	4	1.000	[0.999, 1.001]	9.17e-01
teapot	bsah	4	1.001	[1.000, 1.003]	8.32e-02
armadillo	median	4	1.000	[0.999, 1.001]	8.59e-01

Continued on next page

Tabelle 7.2: Statistical Analysis Results

Model	Algorithm	k	Speedup μ	Speedup CI	p
stanford-bunny	median	4	1.004	[0.999, 1.011]	3.61e-01
suzanne	median	4	1.000	[0.998, 1.001]	4.51e-01
teapot	median	4	1.000	[0.998, 1.001]	7.00e-01
armadillo	sah	4	1.001	[0.998, 1.003]	6.64e-01
stanford-bunny	sah	4	1.000	[0.998, 1.002]	9.84e-01
suzanne	sah	4	1.001	[1.000, 1.002]	2.64e-01
teapot	sah	4	1.000	[0.999, 1.001]	7.09e-01
armadillo	bsah	8	1.001	[1.000, 1.003]	2.16e-01
stanford-bunny	bsah	8	1.001	[0.998, 1.005]	6.28e-01
suzanne	bsah	8	1.002	[0.999, 1.006]	3.62e-01
teapot	bsah	8	1.001	[1.000, 1.002]	2.32e-01
armadillo	median	8	1.001	[1.000, 1.002]	6.77e-02
stanford-bunny	median	8	1.001	[1.000, 1.002]	2.94e-01
suzanne	median	8	0.999	[0.998, 1.000]	2.47e-01
teapot	median	8	1.000	[0.999, 1.001]	9.99e-01
armadillo	sah	8	1.000	[0.999, 1.001]	8.13e-01
stanford-bunny	sah	8	1.000	[0.998, 1.002]	8.89e-01
suzanne	sah	8	1.000	[0.999, 1.001]	7.74e-01
teapot	sah	8	0.999	[0.998, 1.000]	8.66e-02
armadillo	bsah	16	1.000	[0.998, 1.001]	7.03e-01
stanford-bunny	bsah	16	1.002	[1.000, 1.004]	9.08e-02
suzanne	bsah	16	0.999	[0.998, 1.000]	2.10e-01
teapot	bsah	16	1.000	[0.998, 1.001]	6.03e-01
armadillo	median	16	1.000	[0.999, 1.002]	6.27e-01
stanford-bunny	median	16	1.001	[1.000, 1.002]	1.81e-01
suzanne	median	16	1.000	[0.999, 1.001]	5.89e-01
teapot	median	16	1.000	[0.999, 1.001]	8.13e-01
armadillo	sah	16	0.999	[0.998, 1.001]	4.44e-01
stanford-bunny	sah	16	1.001	[1.000, 1.002]	6.51e-02
suzanne	sah	16	1.001	[1.000, 1.002]	1.32e-01
teapot	sah	16	1.000	[0.999, 1.001]	9.22e-01

Tabelle 7.3: Statistical Analysis Results

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
armadillo	bsah	4	k-way	0.998	[0.989, 1.007]	6.59e-01
stanford-bunny	bsah	4	k-way	1.007	[1.001, 1.015]	1.18e-01
suzanne	bsah	4	k-way	1.081	[1.076, 1.089]	4.97e-09***
teapot	bsah	4	k-way	1.036	[1.031, 1.043]	3.03e-06***
armadillo	median	4	k-way	1.015	[1.011, 1.023]	2.36e-03**
stanford-bunny	median	4	k-way	1.025	[1.021, 1.033]	9.93e-05***
suzanne	median	4	k-way	1.069	[1.064, 1.077]	4.18e-08***
teapot	median	4	k-way	1.064	[1.059, 1.073]	4.49e-08***

Continued on next page

Tabelle 7.3: Statistical Analysis Results

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
armadillo	sah	4	k-way	0.996	[0.994, 0.998]	5.60e-03**
stanford-bunny	sah	4	k-way	1.004	[1.001, 1.006]	9.52e-03**
suzanne	sah	4	k-way	1.073	[1.071, 1.075]	1.45e-20***
teapot	sah	4	k-way	1.029	[1.027, 1.030]	9.72e-20***
armadillo	bsah	4	collapsed	1.002	[0.997, 1.009]	5.91e-01
stanford-bunny	bsah	4	collapsed	1.008	[1.003, 1.016]	5.53e-02
suzanne	bsah	4	collapsed	1.081	[1.076, 1.089]	5.50e-09***
teapot	bsah	4	collapsed	1.034	[1.030, 1.042]	3.99e-06***
armadillo	median	4	collapsed	1.015	[1.011, 1.023]	2.64e-03**
stanford-bunny	median	4	collapsed	1.023	[1.015, 1.032]	1.50e-04***
suzanne	median	4	collapsed	1.069	[1.064, 1.078]	3.90e-08***
teapot	median	4	collapsed	1.064	[1.059, 1.073]	6.75e-08***
armadillo	sah	4	collapsed	0.996	[0.995, 0.998]	6.33e-04***
stanford-bunny	sah	4	collapsed	1.004	[1.002, 1.006]	6.49e-03**
suzanne	sah	4	collapsed	1.072	[1.070, 1.074]	9.14e-19***
teapot	sah	4	collapsed	1.029	[1.028, 1.030]	2.32e-17***
armadillo	bsah	8	k-way	0.862	[0.858, 0.867]	7.72e-13***
stanford-bunny	bsah	8	k-way	0.845	[0.841, 0.851]	7.65e-13***
suzanne	bsah	8	k-way	0.772	[0.768, 0.778]	4.45e-15***
teapot	bsah	8	k-way	0.738	[0.735, 0.743]	4.31e-16***
armadillo	median	8	k-way	0.737	[0.734, 0.743]	2.12e-15***
stanford-bunny	median	8	k-way	0.779	[0.776, 0.785]	4.09e-14***
suzanne	median	8	k-way	0.757	[0.754, 0.764]	6.49e-15***
teapot	median	8	k-way	0.712	[0.708, 0.717]	2.49e-16***
armadillo	sah	8	k-way	0.865	[0.864, 0.866]	1.18e-32***
stanford-bunny	sah	8	k-way	0.842	[0.840, 0.844]	7.80e-28***
suzanne	sah	8	k-way	0.753	[0.752, 0.755]	1.22e-34***
teapot	sah	8	k-way	0.732	[0.731, 0.733]	6.94e-38***
armadillo	bsah	8	collapsed	0.862	[0.858, 0.868]	2.03e-12***
stanford-bunny	bsah	8	collapsed	0.844	[0.839, 0.851]	3.52e-14***
suzanne	bsah	8	collapsed	0.771	[0.766, 0.777]	3.93e-20***
teapot	bsah	8	collapsed	0.737	[0.734, 0.743]	9.24e-16***
armadillo	median	8	collapsed	0.737	[0.733, 0.742]	2.54e-15***
stanford-bunny	median	8	collapsed	0.778	[0.775, 0.785]	7.53e-15***
suzanne	median	8	collapsed	0.758	[0.754, 0.764]	7.94e-15***
teapot	median	8	collapsed	0.712	[0.708, 0.717]	6.87e-16***
armadillo	sah	8	collapsed	0.865	[0.864, 0.866]	3.66e-32***
stanford-bunny	sah	8	collapsed	0.842	[0.840, 0.844]	1.08e-28***
suzanne	sah	8	collapsed	0.754	[0.753, 0.755]	7.40e-25***
teapot	sah	8	collapsed	0.733	[0.732, 0.734]	4.23e-35***
armadillo	bsah	16	k-way	0.288	[0.287, 0.290]	1.68e-34***
stanford-bunny	bsah	16	k-way	0.286	[0.285, 0.289]	7.67e-30***
suzanne	bsah	16	k-way	0.168	[0.167, 0.170]	1.11e-42***
teapot	bsah	16	k-way	0.177	[0.177, 0.179]	7.05e-44***

Continued on next page

Tabelle 7.3: Statistical Analysis Results

Model	Algorithm	k	Type	Speedup μ	Speedup CI	p
armadillo	median	16	k-way	0.156	[0.155, 0.157]	3.84e-43***
stanford-bunny	median	16	k-way	0.194	[0.193, 0.195]	1.75e-37***
suzanne	median	16	k-way	0.169	[0.168, 0.170]	3.76e-36***
teapot	median	16	k-way	0.191	[0.190, 0.193]	3.23e-41***
armadillo	sah	16	k-way	0.297	[0.296, 0.297]	2.98e-29***
stanford-bunny	sah	16	k-way	0.289	[0.288, 0.289]	3.77e-32***
suzanne	sah	16	k-way	0.165	[0.164, 0.165]	6.36e-31***
teapot	sah	16	k-way	0.174	[0.173, 0.174]	3.12e-29***
armadillo	bsah	16	collapsed	0.288	[0.287, 0.290]	2.27e-34***
stanford-bunny	bsah	16	collapsed	0.286	[0.284, 0.288]	4.74e-38***
suzanne	bsah	16	collapsed	0.168	[0.168, 0.170]	1.16e-36***
teapot	bsah	16	collapsed	0.177	[0.177, 0.179]	2.52e-38***
armadillo	median	16	collapsed	0.156	[0.155, 0.157]	2.77e-44***
stanford-bunny	median	16	collapsed	0.194	[0.193, 0.195]	4.42e-32***
suzanne	median	16	collapsed	0.169	[0.168, 0.170]	2.56e-38***
teapot	median	16	collapsed	0.191	[0.190, 0.193]	2.15e-31***
armadillo	sah	16	collapsed	0.297	[0.296, 0.297]	7.87e-27***
stanford-bunny	sah	16	collapsed	0.288	[0.288, 0.289]	5.11e-45***
suzanne	sah	16	collapsed	0.164	[0.164, 0.165]	1.86e-36***
teapot	sah	16	collapsed	0.174	[0.173, 0.174]	2.50e-28***

Literatur

[AKLase2] Timo Aila, Tero Karras und Samuli Laine. „On quality metrics of bounding volume hierarchies“. In: *Proceedings of the 5th High-Performance Graphics Conference*. HPG '13. Anaheim, California: Association for Computing Machinery, 2013, S. 101–107. ISBN: 9781450321358. URL: https://doi.org/10.1145/2492045.2492056.

[Br] Blender. *The Cycles Render Engine*. URL: https://github.com/blender/cycles/blob/main/examples/objects/suzanne.xml, Zugriff am: 2. Februar 2026.

[En2] C. Ericson. *Real-Time Collision Detection*. Taylor & Francis, 2004. ISBN: 9781558607323.

[GCnr2] Scott Vahl Gordon und John L. Clevenger. *Computer Graphics Programming in OpenGL with C++*. Mercury Learning und Information, 2021, S. 483. ISBN: 9781683926719.

[KKya1] Timothy L. Kay und James T. Kajiya. „Ray tracing complex scenes“. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, S. 269–278. ISBN: 0897911962. URL: https://doi.org/10.1145/15922.15916.

[Ly] Stanford University Computer Graphics Laboratory. *The Stanford 3D Scanning Repository*. URL: https://graphics.stanford.edu/data/3Dscanrep/, Zugriff am: 2. Februar 2026.

[MBrr2] Daniel Meister und Jiří Bittner. „Performance comparison of bounding volume hierarchies for GPU ray tracing“. In: *Journal of Computer Graphics Techniques (JCGT)* 11.4 (2022), S. 1–19.

[Mr2] Daniel Meister u. a. „A Survey on Bounding Volume Hierarchies for Ray Tracing“. In: *Computer Graphics Forum* 40.2 (2021), S. 683–712.

[MTre1] Tomas Möller und Ben Trumbore. „Fast, Minimum Storage Ray-Triangle Intersection“. In: *Journal of Graphics Tools* 2.1 (1997), S. 21–28.

[Nl] Martin Newel. *The Utah Teapot*. URL: https://graphics.cs.utah.edu/teapot/, Zugriff am: 2. Februar 2026.

[WBBdns2] Ingo Wald, Carsten Benthin und Solomon Boulos. „Getting rid of packets - Efficient SIMD single-ray traversal using multi-branching BVHs -“. In: *2008 IEEE Symposium on Interactive Ray Tracing*. 2008, S. 49–57.