

Bachelorarbeit

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

Nick Garczorz

10. Januar 2026

Heinrich-Heine-Universität Düsseldorf

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

Zusammenfassung

tbc

Inhaltsverzeichnis

1	Einleitung	4
1.1	Motivation	4
1.2	Thema und fachliche Einordnung	4
1.3	Zielsetzung	4
1.4	Abgrenzung zu vorherigen Arbeiten	5
2	Grundlagen	7
2.1	Modell	7
2.2	Axis Aligned Bounding Box	7
2.3	Bounding Volume Hierarchy	7
2.3.1	Wide Bounding Volume Hierarchy	8
2.3.2	Traversal	8
2.3.3	Kostenmodelle	8
2.3.4	Construction	9
3	Methoden	10
3.1	Aufbau	10
3.1.1	Überblick	10
3.1.2	Datenmodell	10
3.2	Algorithmenanalyse	12
3.2.1	Traversal	12
3.2.2	Construction	12
4	Ergebnisse	15
5	Diskussion	16
6	Fazit	17
7	Appendix	18

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 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 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,

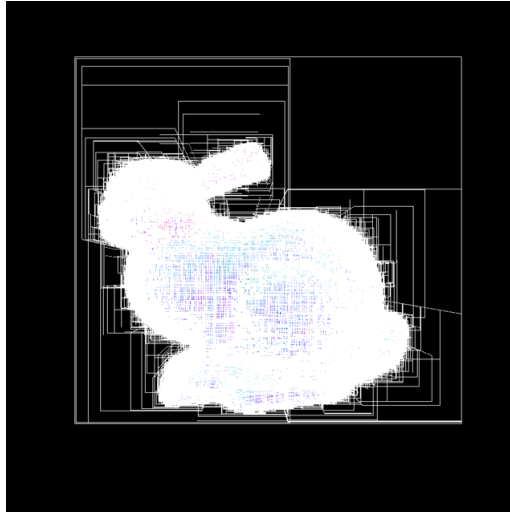


Abbildung 1.1: Stanford Bunny mit sichtbaren Bounding Boxes

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 beleuchtet 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 Qualität der BVHs zwischen direktem k-way splitting und collapse einer binären BVH abhängig von der gewählten statischen Szene?
- Wie groß ist der Trade-off zwischen Construction- und Traversal-Time mit Bezug auf das gewählte Splitting-Verfahren jeweils über $k \in \{2, 4, 8, 16\}$ in statischen Szenen?

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 dies der Industriestandard ist [?].

Die Vergleichenen 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

1.4 Abgrenzung zu vorherigen Arbeiten

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 verwendeten 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 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.

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 Partitionierung 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 direkten 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 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 gegebenen Nodes besucht wird. Dabei entsprechen die Kosten eines inneren Nodes N :

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

[?]

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 einem 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 [?].

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

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 fälle 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 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 Snapshots unterteilt. Bei der Kamerafahrt handelt es sich um einen 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. Dabei werden als Hauptmetriken die Construction-Time und die Traversal-Time erfasst und verglichen. Zudem wird die Anzahl an *Hit Rays*, also die Anzahl an Rays, die tatsächlich mit einem Polygon kollidieren, erfasst. Dies geschieht um die Modelle anhand ihrer Polygonanzahl zu vergleichen, obwohl diese unterschiedlich skaliert sind. Die Tests erfolgen lediglich auf der CPU.

3.1.2 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 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.

```
1 class Primitive
2 {
3     Vector3 getCenter()
4     Vector3 getMin()
5     Vector3 getMax()
6     Vector3 intersect(ray)
7 }
```

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.

```
1 class AABB
2 {
3     Vector3 getMin()
4     Vector3 getMax()
5     Vector3 intersect(ray)
6 }
```

Listing 3.2: Axis Aligned Bounding Box

3.1 Aufbau

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.

```
1 class BVHNode
2 {
3     AABB aabb
4     Primitive [] primitives
5     BVHNode [] children
6 }
7
8 class BVH
9 {
10    BVHNode root
11    Primitive [] primitives
12 }
```

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 den 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.

```

1 function traverse(bvh, ray):
2     stack := empty list
3     push(stack, bvh.root)
4
5     closestDistance := +∞
6
7     while stack is not empty do
8         node := pop(stack)
9
10        if not node.aabb.intersect(ray) then
11            continue
12        end if
13
14        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

```

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)

```

3.2 Algorithmenanalyse

```

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

```

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) <
19                  center(primitiveB).component(axis))

```

3.2 Algorithmenanalyse

```

19     end for
20     return splitIndices
21 end function

```

Listing 3.6: Median Partition

Surface Area Heuristic Split

```

1 function sahSplit(primitives, axis, degree): //O(n * log(n))
2
3     count := primitives.size()
4     if isLeaf(count, degree) then
5         return {}
6     end if
7
8     //O(n * log(n))
9     sort(set = primitives,
10         compare(primitiveA, primitiveB):
11             return center(primitiveA).component(axis) < center(primitiveB).component(axis))
12
13     splitIndices := empty list
14     segments := empty list
15     push(segments, primitives)
16     while segments.size() < degree do
17
18         seg := findSegmentToSplitGreedy(segments) //highest cost segment has to be split
19         // bounds of primitives from segment from begin to index
20         prefixBoundingBoxes := calculatePrefix(seg) //O(n)
21         // bounds of primitives from segment from index to end (included)
22         suffixBoundingBoxes := calculateSuffix(seg) // O(n)
23
24         segCount := seg.count()
25         split := segCount / 2
26         minCost := +∞
27         for i from 1 to segCount - 2 do
28             leftArea := surfaceArea(prefixBoundingBoxes[i - 1])
29             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

4 Ergebnisse

- k = 4 als Sieger des statischen und

Tabelle 4.1: Collapsed vs k-way Algorithm Comparison Results

Model	Algorithm	k	k-way Time (s)	Collapsed Time (s)	Speedup	% Change	t-statistic
armadillo	bsah	4	1.034200	1.030300	1.004000	-0.400000	1.0530
stanford-bunny	bsah	4	0.953800	0.952200	1.002000	-0.200000	1.1170
suzanne	bsah	4	1.145300	1.145400	1.000000	0.000000	-0.1060
teapot	bsah	4	0.857100	0.858300	0.999000	0.100000	-1.8400
armadillo	median	4	1.246000	1.246100	1.000000	0.000000	-0.1810
stanford-bunny	median	4	1.109500	1.113500	0.996000	0.400000	-0.9610
suzanne	median	4	1.412700	1.412100	1.000000	-0.000000	0.7710
teapot	median	4	1.010800	1.010500	1.000000	-0.000000	0.3930
armadillo	sah	4	1.008800	1.009500	0.999000	0.100000	-0.4420
stanford-bunny	sah	4	0.937000	0.937000	1.000000	0.000000	-0.0210
suzanne	sah	4	1.095700	1.096600	0.999000	0.100000	-1.1540
teapot	sah	4	0.839400	0.839500	1.000000	0.000000	-0.3810
armadillo	bsah	8	1.650900	1.652600	0.999000	0.100000	-1.2860
stanford-bunny	bsah	8	1.489400	1.490800	0.999000	0.100000	-0.4990
suzanne	bsah	8	1.610500	1.613300	0.998000	0.200000	-0.9510
teapot	bsah	8	1.256400	1.257500	0.999000	0.100000	-1.2390
armadillo	median	8	2.066100	2.067900	0.999000	0.100000	-1.9510
stanford-bunny	median	8	1.644800	1.646000	0.999000	0.100000	-1.0950
suzanne	median	8	1.995600	1.993900	1.001000	-0.100000	1.1970
teapot	median	8	1.533600	1.533700	1.000000	0.000000	-0.0010
armadillo	sah	8	1.614600	1.614400	1.000000	-0.000000	0.2400
stanford-bunny	sah	8	1.482600	1.482400	1.000000	-0.000000	0.1420
suzanne	sah	8	1.567800	1.567500	1.000000	-0.000000	0.2940
teapot	sah	8	1.233900	1.232400	1.001000	-0.100000	1.8160
armadillo	bsah	16	10.835300	10.831700	1.000000	-0.000000	0.3880
stanford-bunny	bsah	16	8.074400	8.090300	0.998000	0.200000	-1.8420
suzanne	bsah	16	7.457200	7.451500	1.001000	-0.100000	1.3060
teapot	bsah	16	5.665300	5.663100	1.000000	-0.000000	0.5310
armadillo	median	16	13.528600	13.533200	1.000000	0.000000	-0.4940
stanford-bunny	median	16	8.519600	8.526600	0.999000	0.100000	-1.3990
suzanne	median	16	8.959000	8.961700	1.000000	0.000000	-0.5500
teapot	median	16	5.834500	5.833500	1.000000	-0.000000	0.2420
armadillo	sah	16	10.704200	10.697400	1.001000	-0.100000	0.7830
stanford-bunny	sah	16	8.134300	8.143800	0.999000	0.100000	-1.9910
suzanne	sah	16	7.245700	7.251600	0.999000	0.100000	-1.5850
teapot	sah	16	5.644400	5.644700	1.000000	0.000000	-0.0990

5 Diskussion

tbc

6 Fazit

tbc

7 Appendix