Christian Woizischke
christian@woizischke.de

Betreuer Martin Eisemann
eisemann@cg.tu-bs.de
Institut für Computergraphik, TU Braunschweig

Erstgutachter Prof. Dr. Ing. Marcus Magnor
magnor@cg.tu-bs.de
Institut für Computergraphik, TU Braunschweig

Zweitgutachter Prof. Dr.-Ing. Friedrich M. Wahl
f.wahl@tu-bs.de
Institut für Robotik und Prozessinformatik, TU Braunschweig

SIMD-Raytracing mittels Single Slab Hierarchy

Bachelorarbeit

22. September 2008

Institut für Computergraphik, TU Braunschweig

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, 22. September 2008

Christian Woizischke

Zusammenfassung

Raytracing ist ein Verfahren zur Generierung synthetischer Bilder, dessen Hauptproblem das Finden von Schnittpunkten zwischen Strahlen und 3D-Objekten ist. Die Laufzeit dieses Vorgangs lässt sich mit Beschleunigungsstrukturen verringern, sodass auf aktueller Hardware sogar interaktive Anwendungen möglich werden. Gegenstand dieser Arbeit ist eine neue Beschleunigungsstruktur, die Single Slab Hierarchy, und ein bereits existierender einfacher Raytracer, der diese benutzt. Die Single Slab Hierarchy ist von der Bounding Volume Hierarchy abgeleitet und soll durch die Repräsentation der Knotenvolumen durch nur eine Seite der Axis Aligned Bounding Box einen geringeren Speicherverbrauch und eine höhere Geschwindigkeit als die Bounding Volume Hierarchy aufweisen. Ziel dieser Arbeit ist die Überprüfung dieser Annahmen anhand eines praktischen Vergleichs, sowie die Erweiterung des Raytracers um eine iterative Variante der Traversierung beider Beschleunigungsstrukturen, gängige Beleuchtungseffekte und einen interaktiven Modus, in dem die Szene mit einer Kamera durchflogen werden kann. Die schriftliche Ausarbeitung der Arbeit stellt die Single Slab Hierarchy und die Implementierung der Änderungen bzw. Erweiterungen des Raytracers vor und präsentiert die Ergebnisse des Vergleichs beider Beschleunigungsstrukturen in tabellarischer, schriftlicher und grafischer Form.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Raytracing ist ein physikbasiertes Verfahren mit dem synthetische fotorealistische Bilder ohne Grafik-Hardwarebeschleunigung (d.h. auf der CPU) erzeugt werden können. Das hauptsächlich serielle Design der Prozessoren führt meistens dazu, dass Raytracing für vergleichbare Szenen eine wesentlich geringere Bildrate erreicht als die Rasterisierung mit Grafikkarten, jedoch liegt diese für einfache Szenen seit einigen Jahren in einem Bereich, der eine gewisse Interaktivität ermöglicht[WBWS01]. Dies ist nicht nur auf die Entwicklung der Computerhardware zurückzuführen, sondern auch auf die Entwicklung der Beschleunigungsdatenstrukturen. Mit ihnen lässt sich die Szene in sinnvolle Einheiten unterteilen und dadurch die Korrespondenzsuche zwischen den Bildschirmpunkten und der Geometrie beschleunigen.

Bislang gibt es keine Beschleunigungsstruktur, die in den Bereichen Geschwindigkeit und Speicherverbrauch bestmögliche Werte liefert. Es existiert eine Beschleunigungsstruktur, die aufgrund ihrer schnellen Schnitttests oft für interaktive Raytracer verwendet wird (kd-tree)[WBWS01], jedoch lässt sich ihr Speicherverbrauch nicht vorhersehen und er kann sogar deutlich höher sein als bei anderen Beschleunigungsstrukturen. Eine einfache Beschleunigungsstruktur mit vorhersehbarem Speicherverbrauch ist die Bounding Volume Hierarchy (BVH). Bei dieser Beschleunigungsstruktur lassen sich der Speicherverbrauch und die Geschwindigkeit aber noch verbessern, indem sie leicht angepasst wird. Diese Arbeit beschäftigt sich mit einer solchen Beschleunigungsstruktur, der Single Slab Hierarchy (SSH), und einem Raytracer aus einer früheren Arbeit, der mit der SSH und einer normalen BVH einfache Bilder erzeugt. Diese Arbeit soll den Raytracer zunächst erweitern und dann damit einen praktischen Vergleich zwischen der SSH und der BVH durchführen.

Kapitel 2 vermittelt zunächst die theoretische Grundlage des Raytracings, der klassischen Beschleunigungsstrukturen und gängiger Beleuchtungseffekte. Kapitel 3 stellt dann kurz einige Verfahren mit ähnlichem Ansatz oder ähnlicher Zielsetzung wie bei der SSH vor und zeigt deren Nachteile auf. Da-

nach werden in Kapitel 4 der Aufbau und die Arbeitsweise der SSH erklärt. In Kapitel 5 werden die Erweiterungen und Änderungen an dem Raytracer inklusive einiger Implementierungsdetails vorgestellt. Kapitel 6 gibt schlussendlich Auskunft über den Testaufbau, d.h. verwendete Einstellungen des Raytracers und verwendete Testszenen, und präsentiert die Ergebnisse in schriftlicher, tabellarischer und grafischer Form.

Kapitel 2

Grundlagen

Dieses Kapitel vermittelt die Grundlagen über Raytracing (2.1), Beschleunigungsstrukturen (2.2) und die klassischen Beleuchtungseffekte (2.3).

2.1 Raytracing Algorithmus

Der klassische Raytracing Algorithmus nach Whitted[Whi80] basiert auf der Annahme, dass sich das Licht als Photonengruppe auf einer geraden Linie im Raum ausbreitet und dabei aufgespalten, reflektiert und gebrochen werden kann. Weil nicht jedes Photon von der Lichtquelle zum Betrachter gelangt, bedient man sich eines Tricks, um die Anzahl der Strahlverfolgungen zu minimieren: Das Licht wird nicht von der Lichtquelle zum Betrachter verfolgt, sondern rückwärts d.h. vom Betrachter zur Lichtquelle. Die so entstandenen Strahlen werden Sehstrahlen genannt und lassen sich anhand eines einfachen Kameramodells, z.B. dem Lochkameramodell, leicht berechnen. Bei einer Lochkamera, verlaufen die Lichtstrahlen von der Szene durch ein Loch auf die Projektionsfläche. Die Sehstrahlen sind also jene Strahlen, welche von den Pixeln der Projektionsfläche durch das Loch verlaufen. Ein solcher Sehstrahl, welcher auch Primärstrahl genannt wird, kann nun mathematisch mit der Geometrie der Szene geschnitten werden. Wird ein Schnittpunkt erkannt, so kann anschließend ein Farb- und ein Helligkeitswert zu dem zugehörigen Pixel berechnet werden. Die gängigen Verfahren für die Berechnung dieser Werte werden in Sektion 2.3 vorgestellt. Wenn die Oberfläche der Geometrie lichtdurchlässig ist oder spiegelnd reflektiert, werden neue Strahlen, sog. Sekundärstrahlen, erstellt, die ebenfalls mit der Geometrie der Szene geschnitten werden. Für die Schattenberechnung werden ebenfalls weitere Strahlen, die sog. Schattenstrahlen, erzeugt und verfolgt. Diese Schattenstrahlen beginnen im Schnittpunkt zwischen dem Primärstrahl und der Oberfläche und laufen in Richtung Lichtquelle. Trifft ein solcher Strahl auf ein Objekt zwischen dem Schnittpunkt des Primärstrahls und der Lichtquelle, liegt der Schnittpunkt des Primärstrahls im Schatten.

Der Aufwand der Strahlverfolgung liegt in O(n), weil der Strahl mit jedem Objekt (z.B. Dreiecke) geschnitten werden muss. Dies führt bei großen Szenen zu einer nicht akzeptablen Laufzeit. Beschleunigungsstrukturen, von denen einige in 2.2 vorgestellt werden, können die Strahlverfolgung beschleunigen, indem sie Zeitkomplexität des Problems verringern.

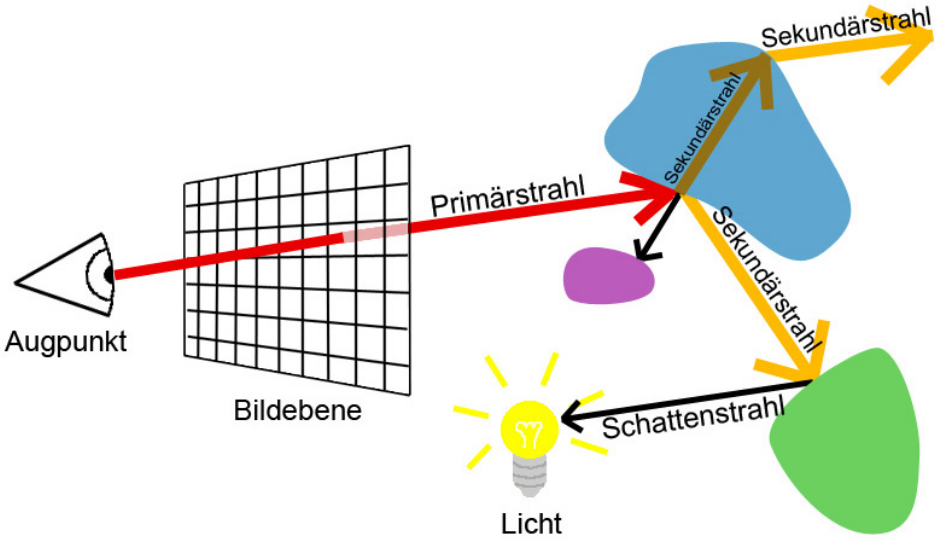

Abbildung 2.1: Beispiel zum Raytracing. Der Primärstrahl verläuft vom Projektionszentrum (Hier *Augpunkt*) der Lochkamera durch die am Projektionszentrum gespiegelte Projektionsfläche (*Bildebene*) und trifft ein blaues halbtransparentes Objekt. Ein Teil des Lichts wird gebrochen und ein Teil reflektiert. Die Oberflächenpunkte sind im Schatten, wenn der Schattenstrahl (Hier in schwarz) die Lichtquelle nicht erreicht.)

2.2 Klassische Beschleunigungsstrukturen

Es gibt verschiedene Arten von Beschleunigungsstrukturen. Grundsätzlich unterscheidet man zwischen räumlicher Aufteilung der Szene in disjunkte Voxel und Aufteilung der Objektliste in disjunkte Teillisten. Bei der räumlichen Aufteilung werden Objekte (z.B. Dreiecke) aufgeteilt, wenn sie nicht eindeutig zu einem der Voxel zugeordnet werden können. Bei der Aufteilung der Objektliste entstehen Volumen, die sich überschneiden können, jedoch in der Regel einen großen disjunkten Teil besitzen. Die Objekte werden dafür nicht aufgeteilt.
Die wohl einfachste Variante der räumlichen Aufteilung ist das gleichförmige dreidimensionale Raster (uniform grid) von Fujimoto *et al.* [FTI86], in dessen Zellen (Voxel) die Geometrie eingeordnet wird. Mithilfe eines sog. 3DDDA-

Algorithmus kann ermittelt werden, welche Voxel der Strahl schneidet. Für diese Voxel wird dann der aufwendige Schnitttest zwischen dem Strahl und der enthaltenen Geometrie ausgeführt. Für die anderen Voxel müssen die Schnitttests nicht durchgeführt werden. Der Nachteil dieses Verfahrens zeigt sich bei Szenen, deren Geometrie sich auf lokale Stellen im Raum konzentriert, z.B. bei dem *teapot in a stadium*. Der Raum zwischen diesen Stellen ist leer. Trotzdem müssen die Voxel durchlaufen werden, die den Strahl schneiden. Mit hierarchischen Strukturen kann dieses Problem gelöst werden. Die bekanntesten hierarchischen Beschleunigungsstrukturen sind der Octree, der kd-Tree und die Bounding Volume Hierarchy.
Der Octree von Glassner [Gla84] ist wie das uniform grid eine raumaufteilende Beschleunigungsstruktur. Hier wird die gesamte Szene in acht gleich große Teile aufgeteilt, indem die Szene auf jeder Achse halbiert wird. Die Axis Aligned Bounding Box (AABB) der gesamten Szene bildet die Wurzel eines Baums und die acht Teile der Szene bilden die Kindknoten dieser Wurzel. Jeder Kindknoten wird wiederum in acht gleich große Teile unterteilt, welche die nächsten Kindknoten bilden. Zum Schnitttest eines Strahls mit der Szene wird dieser zunächst gegen einen Knoten getestet und der Schnitttest beim Treffer rekursiv für die acht Kindknoten weitergeführt. Der Vorteil ist, dass große leere Teile der Szene weggeschnitten werden können. Nachteilig ist aber der aufwendige Schnitttest und dass selbst kleine Objekte einen hohen Baum zur Folge haben können, denn die Knoten werden unterteilt, solange das Objekt in einen Kindknoten passt.
Der kd-Tree von Bentley [Ben75, Kap85], ebenfalls eine raumaufteilende Beschleunigungsstruktur, ist ein binärer Baum, dessen Knoten jeweils durch eine Ebene, die senkrecht auf einer Achse des Koordinatensystems steht, geteilt werden. Beim Schnitttest wird der Strahl lediglich mit der Ebene des aktuellen Knotens geschnitten. Dies ist viel effizienter als ein Schnitttest mit einer Bounding Box. Für jeden Pfad durch den Baum wird ein Strahlintervall erzeugt und verändert, welches den Teil des Strahls kennzeichnet, der innerhalb des zu testenden Volumens liegt. Bei jedem Schnitt mit einer Ebene wird eine Seite dieses Strahlintervalls verändert, so dass es immer kleiner wird. Liegt der Anfang des Strahlintervalls nach dem Schnitt mit dem aktuellen Knoten in Strahlrichtung hinter dem Ende des Strahlintervalls, bedeutet dies, dass der Strahl den aktuellen Knoten nicht schneidet und der Schnitttest für den Teilbaum beendet ist. Für statische Szenen ist die Traversierungsgeschwindigkeit eines kd-Trees sehr hoch, jedoch ist der Speicherverbrauch wie bei allen raumaufteilenden Beschleunigungsstrukturen durch die Spaltung bzw. Mehrfachreferenzierung der Objekte im Voraus nicht berechenbar.
Die Bounding Volume Hierarchy (BVH) [KK86, RW80] ist, ähnlich wie der kd-Tree, ein binärer Baum, dessen Knoten für gewöhnlich auf einer Achse unterteilt werden. Hier werden die Teile jedoch nicht als Voxel angesehen, sondern genutzt, um die Objektliste aufzuteilen. In den Knoten wird die

AABB der Geometrie gespeichert. Ein Strahl wird beim Durchlaufen der Hierarchie wie bei den anderen hierarchischen Verfahren zuerst gegen die AABB eines Knotens getestet und beim Schnitt wird der Schnitttest für beide Kindknoten rekursiv weitergeführt. In den Blattknoten befindet sich für gewöhnlich genau ein Geometrieobjekt (z.B. ein Dreieck). Die BVH und auf ihr basierende Methoden sind aufgrund des vorhersehbaren Speicherverbrauchs und der akzeptablen Geschwindigkeit bei Aufbau und Traversierung häufig benutzte Beschleunigungsstrukturen.

2.3 Gängige Beleuchtungseffekte

Das bekannteste Beleuchtungsmodell ist das Phong-Modell. Es produziert glaubhafte Ergebnisse und ist einfach zu berechnen. Das an der Oberfläche reflektierte Licht setzt sich dabei aus drei Teilen zusammen:

Ambient Der ambiente Lichtanteil I_a soll eine Grundhelligkeit simulieren, die dadurch zu Stande kommt, dass das Licht einer Lichtquelle in der Szene unendlich oft reflektiert wird und somit auch in Regionen gelangt, die nicht direkt von der Lichtquelle bestrahlt werden. Dies ist eine sehr einfache Approximation der echten indirekten Beleuchtung und zeichnet sich durch eine extrem schnelle Berechnung aus, da es ein konstanter Wert ist, welcher auf das Beleuchtungsergebnis aufaddiert wird.

Diffuse Der diffuse Reflexionsanteil resultiert aus der Streuung von Lichtstrahlen auf der rauen Oberfläche eines Objekts. Er hängt lediglich vom Winkel des Lichteinfalls ab und nicht vom Betrachter. Für die normalisierte Richtung des Lichtstrahls L und die normalisierte Oberflächennormale N berechnet sich der diffuse Anteil als

$$I_d = \cos \angle(N, -L) = -N \cdot L. \tag{2.1}$$

Specular Der spiegelnde (engl. specular) Reflexionsanteil simuliert die Lichtreflexionen an einer glatten Oberfläche. Das Licht wird abhängig vom Winkel zwischen Blickrichtung und Lichtrichtung mehr oder weniger stark an der Oberfläche gespiegelt. Für die normalisierte Richtung des Lichtstrahls L und die normalisierte Oberflächennormale N ist die Richtung der gespiegelten Lichtstrahlen

$$R = 2 \cdot \cos \angle(N, -L) \cdot N + L = -2 \cdot (N \cdot L) \cdot N + L. \tag{2.2}$$

Mit der normalisierten Richtung des Sehstrahls V und einem Maß für die Schärfe der Spiegelung k ist der spiegelnde Reflexionsanteil

$$I_s = \cos^k \angle(R, -V) = -(R \cdot V)^k. \tag{2.3}$$

Alle drei Komponenten des Phong-Modells werden in der Regel mit jeweils einem Reflektanzwert der Oberfläche $k_{ambient}$ bzw. $k_{diffuse}$ bzw. $k_{specular}$ und der Stärke des einfallenden Lichts I_{in} multipliziert. Seien I^i_{in}, I^i_d und I^i_s die Werte für die Lichtquelle i, dann ist das Beleuchtungsergebnis I für N Lichtquellen:

$$I = k_{ambient} \cdot I_a + \sum_{i}^{N} (I^i_{in} \cdot (k_{diffuse} \cdot I^i_d + k_{specular} \cdot I^i_s)). \tag{2.4}$$

Der spiegelnde Anteil beim Phong-Modell bezieht sich nur auf das Licht, welches direkt von der Lichtquelle auf die Oberfläche strahlt. Licht, welches von anderen Oberflächen reflektiert wird und in einer spiegelnden Oberfläche sichtbar sein soll, muss aufwendiger berechnet werden. Dafür werden im Schnittpunkt neue Strahlen, sog. Sekundärstrahlen, erzeugt. Die Richtung der Sekundärstrahlen entspricht der Reflexion der Primärstrahlrichtung an der Oberfläche. Ein Sekundärstrahl kann beim Schnitt mit der Geometrie erneut Sekundärstrahlen erstellen, so dass die Reflexion zwischen zwei Spiegeln simuliert werden kann. Für transparente Materialien verwendet man die gleiche Strategie, jedoch werden die Primärstrahlen anhand des Snelliusschen Brechungsgesetz gebrochen. Für den Winkel α des einfallenden Lichtstrahls, den Winkel β des gebrochenen Lichtstrahls und den Brechzahlen der jeweiligen Medien gibt das Snelliussche Brechungsgesetz folgende Gleichung vor:

$$n_1 \cdot \sin\alpha = n_2 \cdot \sin\beta. \tag{2.5}$$

Für die normalisierte Richtung V vom Schnittpunkt zum Betrachter und die normalisierte Oberflächennormale N erhält man die Richtung R des gebrochenen Lichtstrahls durch folgende Rechnung (Vgl. Abb. 2.2):

$$\cos\alpha = V \cdot N \tag{2.6}$$

$$\sin\alpha = \sqrt{1 - \cos^2\alpha} \tag{2.7}$$

$$S_1 = N \cdot \cos\alpha - V \tag{2.8}$$

$$||S_1|| = \sin\alpha \tag{2.9}$$

$$\sin\beta = \frac{n_1}{n_2} \cdot \sin\alpha \tag{2.10}$$

$$S_2 = \frac{\sin\beta}{\sin\alpha} \cdot S_1 \tag{2.11}$$

$$||S_2|| = \sin\beta \tag{2.12}$$

$$\cos\beta = \sqrt{1 - \sin^2\beta} \tag{2.13}$$

$$R = S_2 - N \cdot \cos\beta \tag{2.14}$$

Durch Einsetzen und Vereinfachen erhält man:

$$R = \frac{n_1}{n_2} \cdot (N \cdot (V \cdot N) - V) - N \cdot \sqrt{1 - \frac{n_1^2}{n_2^2} \cdot (1 - (V \cdot N)^2)}. \tag{2.15}$$

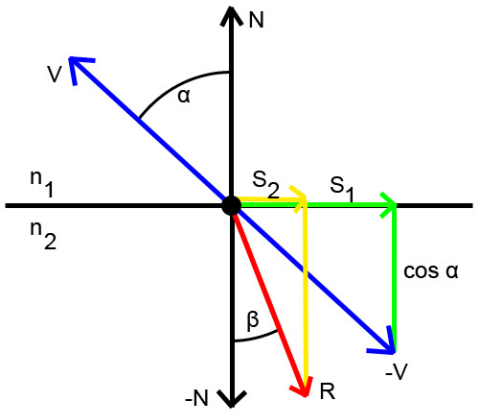

Abbildung 2.2: Zur Veranschaulichung der Rechnung 2.6 - 2.15 für die Lichtbrechung nach Snell.

Die Stärke der Reflexion r im Vergleich zur Transmission t muss in vielen Fällen für flache Winkel 100% und für steile Winkel 0% betragen. Ein solcher Wert lässt sich mit den Fresnelschen Formeln berechnen. Dabei wird zwischen parallel und senkrecht polarisiertem Licht unterschieden. Für den Winkel α des einfallenden Lichtstrahls und den Winkel β des gebrochenen Lichtstrahls werden die Reflexionskoeffizienten r_s für senkrecht polarisiertes Licht und r_p für parallel polarisiertes Licht nach folgenden Formeln berechnet:

$$r_s = \frac{n_1 \cdot \cos\alpha - n_2 \cdot \cos\beta}{n_1 \cdot \cos\alpha + n_2 \cdot \cos\beta}, r_p = \frac{n_2 \cdot \cos\alpha - n_1 \cdot \cos\beta}{n_2 \cdot \cos\alpha + n_1 \cdot \cos\beta} \tag{2.16}$$

Um einen Wert für das nicht polarisierte natürliche Licht zu erhalten, berechnet man den Mittelwert der beiden Werte.

$$r = \frac{r_s + r_p}{2} \tag{2.17}$$

Kapitel 3

Ähnliche Verfahren

Die Single Slab Hierarchy basiert auf der Idee, dass die Laufzeit und der Speicherverbrauch einer herkömmlichen BVH gesenkt werden kann, indem die Geometrie nicht mit Axis Aligned Bounding Boxes (AABBs) approximiert wird, sondern mit Ebenen. Es gibt bereits Verfahren, die das ebenfalls tun, wie z.B. der skd-tree von Ooi *et al.* [OSDM87] und die Bounding Interval Hierarchy von Wächter *et al.* [WK06]. Beide Verfahren teilen die Objektliste anhand einer Schnittebene im Raum und speichern die inneren Seiten der AABBs beider Teillisten, die parallel zur Schnittebene liegen. Man kann sich das so vorstellen, als ob die Schnittebene für beide Kindknoten parallel verschoben wird, so dass die eine Seite der AABB des jeweiligen Kindknotens, die sich nicht am Rand des aktuellen Knotens befindet, in ihr liegt. Zwischen der Ebene und der Außenseite des jeweiligen Kindknotens liegt die Geometrie. Zwischen den Ebenen beider Kindknoten kann sich leerer Raum befinden, aber auch Teile der Geometrie. Beide Volumen können sich also überschneiden. Die beiden Ebenen werden in dem Knoten, der mit der Schnittebene unterteilt wurde, gespeichert. Die Knoten der Hierarchie gleichen also eher einem kd-tree als einer Bounding Volume Hierarchie, aber im Gegensatz zum kd-tree wird beim skd-tree bzw. der Bounding Interval Hierarchy die Objektliste in disjunkte Listen geteilt und nicht der Raum in disjunkte Voxel. Der Nachteil des Verfahrens ist, dass die beiden Ebenen immer parallel zur Schnittebene liegen und dadurch in vielen Fällen weniger leeren Raum wegschneiden, als auf anderen Achsen des Koordinatensystems liegende Ebenen. Um dies auszugleichen erweitern Havran *et al.* den skd-tree mit ihrem H-tree um zusätzliche Knotenarten, die die Geometrie besser approximieren[HHS06]. Anhand einer Kostenfunktion wird entschieden, ob ein skd-tree-Knoten oder ein komplexerer Knoten verwendet wird. Diese gesteigerte Komplexität führt allerdings zu größerem Rechen- und Speicheraufwand, der sich nur deswegen lohnt, weil die Approximation im reinen skd-tree in einigen Fällen nicht gut genug ist. Eine weitere Beschleunigungsstruktur, die mit Ebenen arbeitet, ist der b-kd tree von Woop *et al.*

[WMS06], bei dem die Knoten durch zwei Ebenen repräsentiert werden, zwischen denen die Geometrie liegt. Diese Ebenen haben für beide Kindknoten die gleiche Orientierung.
Eine Verringerung des Speicherverbrauchs kann bei der BVH auch ohne große strukturelle Änderungen der Knoten erreicht werden, indem die Fließkommazahlen der AABB in diskrete Datentypen umgewandelt werden. Cline *et al.* beschreiben hierfür in [CSE06] eine einfache Methode, bei der die Float-Variablen der AABBs durch Short-Variablen ersetzt werden. Mit den diskreten Werten können auf jeder Achse $2^{15} - 1$ Positionen innerhalb der AABB B^{szene} der Szene angegeben werden. Für den Schnitttest zwischen Strahlen und Knoten werden die Strahlen in den diskreten Raum transformiert, indem sie um $-B^{szene}_{min}$ verschoben und entsprechend skaliert werden. Der Nachteil des Verfahrens ist die geringere Präzision und somit eine geringere maximale Baumtiefe. Außerdem verbrauchen die Transformationen wertvolle Rechenzeit.

Kapitel 4

Single Slab Hierarchy

Die Single Slab Hierarchy (SSH) ist eine von der Bounding Volume Hierarchy (BVH) abgeleitete Beschleunigungsstruktur. Der Unterschied zur BVH ist, dass für jeden Knoten lediglich eine Seite der AABB gespeichert wird. Auf den ersten Blick gehen Informationen verloren, jedoch werden alle Knoten von der Vereinigung ihrer übergeordneten Knoten umschlossen, so dass wie bei einem kd-tree anhand des aktiven Strahlintervalls berechnet werden kann, ob ein Schnitt des Strahls mit der enthaltenen Geometrie möglich ist. Es ist sogar unnötig, alle sechs Seiten der AABB zu speichern, denn für jeden Knoten K der BVH liegen mindestens die Hälfte aller Seiten der AABBs beider Kindknoten in einer Seite der AABB von K. Das bedeutet, dass der Strahl gegen einige Seiten der AABBs mindestens doppelt, wenn nicht sogar mehrfach, getestet wird. Bei der SSH kommt diese Redundanz nicht vor, denn die Ebenen sind innerhalb eines Pfads alle unterschiedlich.
Die Konstruktion einer SSH folgt grundsätzlich dem gleichen Algorithmus wie die einer BVH. Zunächst wird ein Wurzelknoten erstellt und als aktueller Knoten markiert. Dann werden folgende Schritte ausgeführt:

- Berechne für die Geometrieliste des aktuellen Knotens ein Volumen zur bestmöglichen Approximation des echten Volumens und speichere dieses im aktuellen Knoten.
- Wenn die Geometrieliste nur ein Objekt enthält, speichere dieses im aktuellen Knoten und beende die Konstruktion des Selbigen. Ansonsten führe die folgenden Schritte aus.
- Teile die Geometrieliste anhand einer Ebene im Raum. Füge die Objekte, die von der Ebene geschnitten werden, nur zu einer der beiden Listen hinzu.
- Erstelle für beide Teile je einen Knoten und speichere diese als Kindknoten des aktuellen Knotens.
- Wiederhole die Schritte für beide Kindknoten.

Für die Bestimmung der Ebene zur räumlichen Teilung der Geometrieliste kann ein beliebiger Algorithmus verwendet werden. Bekannte Algorithmen sind *Spatial Median Cut* [SM03], *Object Median Cut* [KK86] und *Surface Area Heurisitic (SAH)* [MB90]. Beim Spatial Median Cut, wird auf einer Achse des Koordinatensystems die Mitte der Szene berechnet und dort die Schnittebene gelegt. Beim Object Median Cut werden die Mittelpunkte der Geometrieobjekte für eine Achse des Koordinatensystems in eine Liste einsortiert und die Schnittebene beim mittleren Punkt dieser Liste angelegt. Bei der SAH wird eine Funktion für die Laufzeit der Traversierung abhängig von der Lage der Schnittebene aufgestellt und diese für jeden Knoten minimiert. Zur Approximation des echten Geometrievolumens wird bei der SSH eine Ebene verwendet. Sie wird so gewählt, dass alle vier Eckpunkte einer AABB-Seite auf ihr liegen. Die Auswahl der AABB-Seite wird so getroffen, dass die Ebene möglichst viel Freiraum von der Geometrie abtrennt. Dazu werden zwei AABBs benötigt. Sei B_1 die AABB der Geometrie und B_2 die AABB des entsprechenden Teilraums vom übergeordneten Knoten (vgl. Abb. 4.1). Sei $O_{min} = \infty$. Nun werden folgende Schritte für jede der sechs Seiten einer AABB durchgeführt:

- Sei B_k eine Kopie von B_2.
- Setze die Seite von B_k auf den Wert der Seite von B_1.
- Berechne die Oberfläche O_k von B_k.
- Setze $O_{min} = min(O_k, O_{min})$. Speichere die aktuelle Seite als Ebene E, falls $O_{min} = O_k$ ist.

Das Ergebnis ist eine Seite von B_1, die als Ebene E (E_1 in Abb. 4.1) den größten Freiraum von B_2 wegschneidet. Diese wird zusammen mit der Angabe der Seite, auf der sich die Geometrie befindet (1 Bit: links oder rechts), im Knoten der SSH gespeichert. Für die Speicherung einer Ebene werden eine Float-Variable und zwei Flag-Bits benötigt. In den zwei Bits speichert man die Achse, zu der die Ebene senkrecht steht.
Im Gegensatz zur Konstruktion gleicht die Traversierung der SSH eher der Traversierung eines kd-trees. Hierbei profitiert man von der Eigenschaft, dass ein Knoten von seinen übergeordneten Knoten umschlossen wird. Um zu erkennen, ob ein Strahl einen Knoten trifft, erstellt man bei der Traversierung des Baums für jeden Pfad ein Strahlintervall. Dieses Intervall gibt an, welcher Teil des Strahls durch den aktuellen Knoten verläuft. Für die Speicherung des Intervalls genügen zwei Gleitkommazahlen die üblicherweise als t_{near} und t_{far} bezeichnet werden. Sie entsprechen der Distanz vom Ursprung des Strahls zum Anfang bzw. Ende des Intervalls. Bei jedem Schnitttest mit einem Knoten wird entweder der Anfang oder das Ende des Intervalls angepasst, je nachdem von welcher Seite der Strahl auf die Ebene trifft. Trifft er

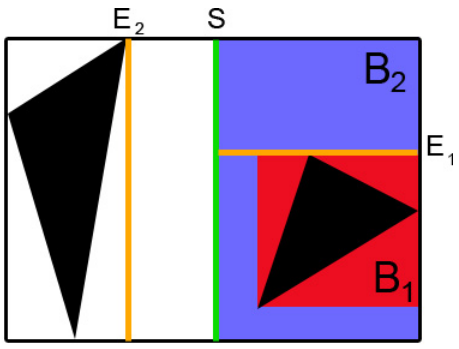

Abbildung 4.1: Beispiel für die Auswahl der Ebene eines SSH-Knotens. S: Schnittebene durch die der Knoten geteilt wird. B_1: AABB der Geometrie. B_2: AABB der rechten Knotenhälfte. E_1: Ebene des rechten Kindknotens. E_2: Ebene des linken Kindknotens. E_1 bzw. E_2 wird so gewählt, dass die resultierende Knotenoberfläche minimal ist.

von der Geometrieseite auf die Ebene, wird t_{far}, andernfalls t_{near} auf den Schnittpunkt gesetzt. Liegt t_{far} nach dem Schnitt in Strahlrichtung hinter t_{near}, d.h. $t_{far} > t_{near}$, so verläuft der Strahl durch den Knoten. Liegt t_{far} in Strahlrichtung vor t_{near}, so schneidet der Strahl den Knoten nicht. Falls der Strahl den Knoten schneidet und dieser ein innerer Knoten ist, wird der Schnitttest für beide Kindknoten fortgeführt. Dabei wird für beide Knoten eine Kopie des aktuellen Strahlintervalls erstellt, die unabhängig voneinander durch die Traversierung der Kindknoten verändert werden (vgl. Abbildung 4.3).

```
float plane;
union {
  int triangleID;    // für Blattknoten
  int firstChildID; // für innere Knoten

  // Bit 0..1: Orientierung der Ebene (x,y,z)
  // Bit 2:    Blattknoten Ja/Nein
  // Bit 3:    Geometrie links oder rechts
  // Bit 4..5: Orientierung der Schnittebene des Vaters
  int flags;
};
```

Abbildung 4.2: Datenstruktur eines SSH-Knotens. Die Orientierung der Schnittebene des Vaterknotens wird für die geordnete Traversierung benötigt (Siehe 5.3).

```
void intersectSSH(Ray& ray, SSHNode* node,
                  float t_near, float t_far)
{
  int axis = node->getSlabAxis();
  bool reverse = ray.dir[axis] < 0;

  // Schnittpunkt zwischen Strahl und Knoten
  float t = (node->plane - ray.origin[axis]) / ray.dir[axis];

  // Strahlintervall anpassen
  if (node->geometryIsLeft()){
    t_near = (reverse || (t<=t_near)) ? t_near : t;
    t_far = (reverse && (t<t_far)) ? t : t_far;
  } else {
    t_near = (reverse && (t>t_near)) ? t : t_near;
    t_far = (reverse || (t>=t_far)) ? t_far : t;
  }

  // Trifft der Strahl den Knoten?
  if (t_near < t_far) {
    if (node->isLeaf()) {
      [intersect geometry];
    } else {
      intersectSSH(ray, node->leftChild, t_near, t_far);
      intersectSSH(ray, node->rightChild, t_near, t_far);
    }
  }
}
```

Abbildung 4.3: Rekursive Traversierung der Single Slab Hierarchy.

Kapitel 5

Implementierung

Die Implementierung dieser Arbeit basiert auf einem Raytracer aus einer früheren Arbeit. Dieser wurde von mir um Beleuchtungseffekte, eine Testumgebung und einen interaktiven Modus erweitert. Dabei wurden auch große Teile des vorhandenen Raytracers umstrukturiert. Um einen Einblick in das Ausmaß der Änderungen und Erweiterungen zu erhalten, wird in 5.1 zunächst der bereits vor Beginn der Arbeit vorhandene Raytracer vorgestellt. Die darauf folgende Sektion 5.2 gibt einen Überblick über die allgemeinen Änderungen an den Klassen und am Kontrollfluss des Programms. In der Sektion 5.3 wird auf Änderungen an der Konstruktion und der Traversierung der Beschleunigungsstrukturen eingegangen. Die Sektion 5.4 gibt einen Einblick in die Implementierung der Beleuchtungseffekte.

5.1 Vorhandener Raytracer

Die Version des Raytracers, auf den diese Arbeit aufbaut, hat nach dem Start ein Bild in eine Datei ausgegeben und sich dann beendet. Zum Erzeugen des Bildes konnte zwischen zwei Beschleunigungsstrukturen gewählt werden: Single Slab Hierarchy (SSH) und Bounding Volume Hierarchy (BVH). Für beide gab es jeweils eine Klasse zur Speicherung der Knoten und zur Traversierung. Die beiden Klassen unterschieden sich recht stark, so wurde z.B. die SSH rekursiv und die BVH iterativ traversiert. Für die iterative Traversierung war die BVH, anders als die SSH, nach dem *depth-first order* Modell aufgebaut. Außerdem waren die Datenstrukturen der Knoten nicht einheitlich. Die Blattknoten der SSH enthielten (entgegen der Definition einer SSH) keine Ebene, wogegen die Blattknoten der BVH eine AABB enthielten. Das lag daran, dass der Speicherbereich für die Ebene in den Blattknoten für die Anzahl der enthaltenen Dreiecke genutzt wurde. Die Algorithmen zur Konstruktion der beiden Beschleunigungsstrukturen, welche in zwei anderen Klassen nach dem *spatial median cut* Verfahren durchgeführt wurde, waren sich daher nicht sehr ähnlich und in einem Test schlecht vergleichbar.

Bezüglich der Beleuchtungseffekte war der Raytracer sehr einfach gehalten. Es gab einen Eyelight-Shader, bei dem der Helligkeitswert aus dem Skalarprodukt der Blickrichtung und der Oberflächennormale berechnet wurde. Bei der Traversierung der Beschleunigungsstruktur wurden immer 4 Strahlen gleichzeitig mittels SIMD-Operationen verfolgt. Zur Beleuchtung wurden diese allerdings wieder getrennt betrachtet. Zur Speicherung der Helligkeitswerte (wohlgemerkt ohne Farbinformationen) wurde die Klasse `Image` genutzt, mit der ein zweidimensionales Array aus RGB-Daten im PPM-Dateiformat gelesen und geschrieben werden kann. Farbinformationen waren also zu keinem Zeitpunkt verfügbar. Die Klassen zum Laden der Szenen aus OBJ- und PLY-Dateien unterstützten dies auch nicht.
Die einzigen Messdaten, die der Raytracer erhob, waren die Dauer der Konstruktion, die Dauer des Raytracings (inkl. Beleuchtung) und die Anzahl der Schnitttests zwischen den Strahlen und den Knoten des Baums.

5.2 Softwarearchitektur

Zur Realisierung der geforderten Erweiterungen habe ich viele Änderungen an der Softwarearchitektur vorgenommen. In der vorherigen Version wurde innerhalb einer Funktion die Szene geladen, die Beschleunigungsstruktur aufgebaut, ein Bild gerendert und der Raytracer wieder beendet. In der neuen Version ist der Raytracingvorgang in einer Klasse namens `RayTracer` gekapselt, welche dafür sorgt, dass alle nötigen Objekte erzeugt, initialisiert und bei Bedarf wieder zerstört werden. Es können mehrere Bilder der selben Szene oder verschiedener Szenen nacheinander erzeugt werden. Der Benutzer legt dafür mit Startparametern eine Liste der 3D-Modelle fest, die nacheinander oder sogar gleichzeitig angezeigt werden sollen. Er kann zwischen drei Modi wählen:

Interaktiv Im interaktiven Modus wird eine Szene geladen, durch die der Benutzer mit Maus- und Tastatureingaben navigieren kann. Für die Art der Kamerasteuerung sind zwei Modi wählbar: Trackball und FirstPerson. Im Trackball-Modus wird die Kamera auf die Szene zentriert und der Benutzer kann die Kamera mit der Maus um die Szene rotieren, an sie heran zoomen oder die Kamera seitlich verschieben. Die Klasse zur Berechnung der Kameraposition und -rotation im Trackball-Modus wurde mir von meinem Betreuer zur Verfügung gestellt und durch mich um einen FirstPerson-Modus erweitert. Im FirstPerson-Modus kann die Kamera mit den Tasten `W,A,S,D` frei im Raum bewegt und mit der Maus um die eigenen Achsen gedreht werden. Die Kameraeigenschaften können mit der Taste `o` für jede Szene in einer eigenen Datei gespeichert und mit der Taste `i` wieder geladen werden. Die gespeicherten Kameraeinstellungen werden automatisch für den Video- und den Testmodus verwendet. Mit der Taste `p` kann ein Bild der Szene als

Datei abgespeichert werden.

Video Im Videomodus wird zu jeder Szene nur ein Bild erzeugt und in einer Datei abgelegt. Danach beendet sich das Programm automatisch. Wenn für die erste Szene gespeicherte Kamerainformationen gefunden werden, werden diese für alle Szenen verwendet, sodass die Kamera für alle Bilder gleich ist. Mit diesem Modus können Bildfolgen von animierten Szenen erzeugt werden, die anschließend beispielsweise mit externen Programmen in Videodateien umgewandelt werden können.

Test Im Testmodus können mehrere Szenen gleichzeitig oder nacheinander für eine bestimmte Anzahl an Bildern dargestellt werden. Danach werden detaillierte Informationen über die Konstruktion der Beschleunigungsstrukturen und den Raytracingvorgang in der Konsole und einer Datei ausgegeben. Abbildung 5.1 zeigt ein Beispiel für diese Ausgabe.

Neben der Ausführung der verschiedenen Modi ist die Klasse `RayTracer` auch für die Verwaltung des OpenGL-Fensters verantwortlich. Zum Erzeugen des Fensters wird GLUT verwendet. Einige Methoden der Klasse `RayTracer` werden von GLUT aufgerufen, um das Programm über Ereignisse (z.B. Tastatureingaben, Rendervorgang, etc.) zu informieren. Zum Anzeigen eines gerenderten Bildes auf dem Monitor kann der Benutzer zwischen drei Verfahren wählen: glDrawPixels, glTexImage2D und Pixel Buffer Objects (PBO). Tabelle 5.1 zeigt die Durchschnittswerte der Geschwindigkeiten dieser Verfahren bei einer Bildgröße von 640x480 Pixel.

Methode	Windows Vista	Ubuntu 8.04
glDrawPixels	0.0061	0.0037
glTexImage2D	0.0046	0.0029
PBO	0.0005	0.0028

Tabelle 5.1: Benötigte Rechenzeit in Sekunden zur Anzeige eines RGB-Bildes der Größe 640x480 Pixel mit OpenGL und einem aktuellen Grafikkartentreiber für NVIDIA GeForce 7800GT

Die Einstellung des Raytracers erfolgt beim Start über die Startparameter. Dabei können die Beschleunigungsstruktur, die Methode zum Anzeigen auf dem Bildschirm, der Modus (Interaktiv, Video, Test), die Bildgröße, das Lichtsetting und natürlich die Szene gewählt werden. Für den Testmodus kann zusätzlich eine Abfolge von Beschleunigungsstrukturen und Szenen angegeben werden. Der Raytracer erzeugt dann nacheinander alle Beschleunigungsstrukturen mit der zuerst angegebenen Szene. Dann schaltet er auf die nächste Szene um und geht wieder alle Beschleunigungsstrukturen durch.

```
--- Test ---
------------
acceleration method: SSH
model file: models/bunny.ply
number of triangles: 69451
scene extends: min(-0.0946899,0.0329874,-0.0618736)
               max(0.0610091,0.187321,0.0587997)
resolution: 640x480
threads: 1
number of frames: 30
construction time: 0.096297
tree height: 24
inner node count: 69450
leaf node count: 69451
computed memory usage of nodes: 1666812
average node surface ratio approx/real: 3.47727
traversal algorithm: iterative, ordered
-------------------------------
average node intersections per ray: 72.0644
average triangle intersections per ray: 5.26804
first traversal time: 0.647111
last traversal time: 0.646993
minimum traversal time: 0.646593
average traversal time: 0.646965
maximum traversal time: 0.647367
first raytrace time: 0.678597
last raytrace time: 0.678606
minimum raytrace time: 0.678499
average raytrace time: 0.678637
maximum raytrace time: 0.678936
```

Abbildung 5.1: Beispiel für die Ausgabe im Testmodus.

Dies wiederholt sich bis jede Szene getestet wurde. Es kann auch ein Ablauf erzeugt werden, bei dem jede Beschleunigungsstruktur mehrfach mit einer Szene getestet wird. Im Testmodus kann außerdem eine Anzahl an Bildern festgelegt werden, nach der die Szene bzw. die Beschleunigungsstruktur wechseln soll. Wenn keine Startparameter angegeben werden, gibt das Programm einen Hilfetext in der Konsole aus.

Der Ablauf des Programms kann Abbildung 5.2 entnommen werden.

Zur Beschleunigung des Raytracingvorgangs werden die Primär- und Sekundärstrahlenbündel parallel in mehreren Threads verfolgt. Hierfür werden die For-Schleifen durch eine OpenMP-Compilerdirektive vom Compiler automatisch parallelisiert. Auf Synchronisierungsmaßnahmen konnte komplett verzichtet werden, indem einige Datenstrukturen wie z.B. die Warteschlange für Sekundärstrahlen (Siehe 5.4) einfach für jeden Thread erzeugt werden.

```
Fenster initialisieren
Startparameter verarbeiten
Mit erster Szene und erster Beschleunigungsstruktur
 im geplanten Ablauf beginnen

WIEDERHOLE
   WENN Szene noch nicht geladen
      Listen für Dreiecke, Shader und Texturen leeren
      Anzahl der Dreiecke aus Szenendateien lesen
      Speicher für die Dreiecksliste reservieren
      Dreiecke und Oberflächeneigenschaften der Szenen laden
   Ggf. Datenstrukturen auf bevorstehende
    Testergebnisse vorbereiten
   Beschleunigungsstruktur erstellen
   Kamera aus Datei laden oder optimal positionieren
   Initialisiere Pixel mit der Farbe Schwarz

   FÜR JEDES SIMD-Primärstrahlenbündel
      Strahl mit der Szene schneiden
      WENN Schnittpunkt gefunden
         Shader ausführen
            Ggf. Sekundärstrahlen verfolgen oder in die
             Warteschlange einreihen
            Ergebnis auf Pixel addieren

   FÜR JEDEN Sekundärstrahl aus der Warteschlange
      Denselben Algorithmus wie für die
       Primärstrahlverfolgung ausführen
       (kann neue Sekundärstrahlen erzeugen)

   Zeige Bild auf dem Bildschirm an
   Ggf. Testergebnisse ausgeben

   WENN letzte Beschleunigungsstruktur im Ablauf
      WENN letzte Szene im Ablauf
         Programm beenden
      ANSONSTEN
         Mit nächster Szene und erster
          Beschleunigungsstruktur fortfahren
   ANSONSTEN
      Mit nächster Beschleunigungsstruktur fortfahren
```

Abbildung 5.2: Ablauf des Raytracers.

Dadurch schreibt kein Thread in den Speicherbereich eines Anderen. Der lesende Zugriff auf die Beschleunigungsstruktur und die Szene ist von allen Threads aus möglich, ohne andere Threads zu stören.

5.3 Beschleunigungsstrukturen

Zunächst musste das Datenlayout der SSH-Knoten geändert werden. Die Blattknoten der SSH haben in der vorherigen Version keine Ebene gespeichert, wogegen bei der BVH in den Blattknoten eine AABB gespeichert wurde. Die Algorithmen der Konstruktion und der Traversierung waren also grundlegend unterschiedlich und daher bei einem Test schlecht vergleichbar. Außerdem war die Höhe des Baums bei der SSH und bei der BVH auf 21 begrenzt, sodass die Blattknoten teilweise mehrere Dreiecke speichern mussten. Eine Begrenzung der Baumhöhe führt zu geringerer Geschwindigkeit bei großen Szenen und die Angabe der Anzahl der Dreiecke verbraucht zusätzlichen Speicherplatz. Daher habe ich die Begrenzung der Baumhöhe aufgehoben und die Angabe der Dreiecksanzahl entfernt. Die SSH habe ich außerdem so umstrukturiert, dass die Blattknoten, wie die inneren Knoten, eine Ebene speichern. Mit den Änderungen an der Traversierung beider Beschleunigungsstrukturen wurden sich beide Klassen so ähnlich, dass ich sie in einer generischen Klasse zusammenführen konnte. Diese Klasse nennt sich `XHierarchy` und ist eine *template class*. Sie enthält die Methoden zur Konstruktion und Traversierung, wobei der Hauptteil der Konstruktion in einer Implementierung der Schnittstelle `XHierarchyConstructionStrategy` liegt. Ihre einzige Implementierung ist zurzeit die Klasse `XHierarchySpatialMedianCut`, welche die Beschleunigungsstruktur nach dem spatial median cut aufbaut. Diese Klasse basiert auf zwei Klassen aus dem vorherigen Raytracer. Sie wurden von mir so angepasst, dass sie die neuen Knotenklassen benutzen. Dann habe ich sie einander angeglichen, um sie zu einer generischen Klasse zusammenzuführen, die von `XHierarchy` angesteuert werden kann. Zur Erstellung einer SSH oder einer BVH muss die Klasse `XHierarchy` und die gewünschte `XHierarchyConstructionStrategy` mit der entsprechenden Knotenklasse `SSHNode` oder `BVHNode` parametrisiert werden. Außerdem besitzen die beiden generischen Klassen abstrakte Methoden, d.h. von den parametrisierten Klassen muss eine Klasse abgeleitet werden, die diese Methoden implementiert. Für die SSH und die BVH sind dies z.B. die Klassen `SingleSlabHierarchy` und `BoundingVolumeHierarchy`. Weitere auf der BVH basierende Beschleunigungsstrukturen können also durch die Erstellung einer Knotenklasse und den entsprechenden Ableitungen von den Klassen `XHierarchy` und `XHierarchySpatialMedianCut` hinzugefügt werden. Folgende Methoden müssen für jede Beschleunigungsstruktur implementiert werden: `updateActiveRaySegment`, `setupRootNode` und `setNodeVolume`. `updateActiveRaySegment` wird dazu verwendet, den Strahl mit dem Knoten

zu schneiden und das Strahlintervall entsprechend anzupassen. Die Methoden `setupRootNode` und `setNodeVolume` werden bei der Konstruktion der Hierarchie zur Speicherung der Daten in den Knoten benötigt. Damit die generischen Klassen auf Eigenschaften der Knoten zugreifen können, gibt es jetzt Methoden in den Knotenklassen. Das vorherige Design, bei dem direkt auf Variablen mit Flags gearbeitet wurde, hatte den Nachteil, dass alle Algorithmen ständig mit dem Datenlayout konsistent gehalten werden mussten. Mit dem neuen methodenorientierten Design können beispielsweise die Flags jederzeit angepasst werden, ohne alle Traversierungs- und Konstruktionsalgorithmen ändern zu müssen.

5.3.1 Rekursive und iterative Traversierung

Wie bereits in 5.1 erwähnt waren die SSH und die BVH im ursprünglichen Raytracer unterschiedlich aufgebaut und wurden unterschiedlich traversiert. Beide Beschleunigungsstrukturen sollen aber gleich aufgebaut sein und sich nur durch die Schnitttests des Strahls mit den Knoten und die Repräsentation der Knoten unterscheiden. Daher hat es sich angeboten, den rekursiven Traversierungsalgorithmus der SSH aus dem ursprünglichen Raytracer für beide Verfahren zu verwenden und auf dessen Basis eine iterative Variante zu entwickeln. Die rekursive Variante besteht aus einer Funktion, die den Strahl mit einem Knoten schneidet und wenn der Strahl den Knoten trifft, die selbe Funktion für beide Kindknoten aufruft. Für die iterative Variante gilt es nun die Funktionsaufrufe zu vermeiden. Dafür benötigt die Traversierungsfunktion eine Schleife, in der für jeden Durchlauf der Strahl mit einem Knoten geschnitten wird und wenn es einen Schnittpunkt gibt, beide Kindknoten durchlaufen werden. Die Kindknoten können allerdings nicht beide sofort durchlaufen werden. Für den nächsten Schleifendurchlauf kann nur ein Knoten als Schnittpartner mit dem Strahl festgelegt werden. Der andere Knoten muss auf eine Warteliste gesetzt werden. Die einfachste und am wenigsten rechenintensive Realisierung einer solchen Warteliste ist ein Stack. Zusätzlich wird auch das aktuelle Strahlintervall auf den Stack gegeben, da es bereits im nächsten Schleifendurchlauf verändert wird und somit verloren gehen würde. Die auf dem Stack wartenden Knoten werden immer dann abgearbeitet, wenn für den nächsten Schleifendurchlauf kein Schnitttest geplant ist, weil entweder der Strahl am aktuellen Knoten vorbei geht und somit keine Kindknoten durchlaufen werden müssen, oder weil der aktuelle Knoten ein Blattknoten ist und keine Kindknoten besitzt. Eine weitere Ursache für das Ende eines Schnitttests innerhalb eines Pfads durch den Binärbaum ist die geordnete Traversierung, die im nächsten Absatz beschrieben wird. Die maximale Größe des Stacks entspricht aufgrund des LIFO-Prinzips der Höhe des Baums, weshalb ein aufwendiges dynamisches Speichermanagement vermieden werden kann. Bereits bei der Konstruktion der Beschleunigungsstruktur kann ein Stack erstellt werden, der die entspre-

chende Anzahl an Elementen fasst, so dass sich der zusätzliche Aufwand im Vergleich zur rekursiven Variante allein aus den Operationen des Stacks ergibt. Die Funktionsaufrufe verbrauchen aber offensichtlich mehr Rechenzeit als die Stack-Operationen, sodass die iterative Traversierung um einige Millisekunden schneller ist (Details: Siehe 5.2).

5.3.2 Geordnete Traversierung

Um die Traversierung zu beschleunigen bietet es sich an, die Knoten in einer Reihenfolge zu durchlaufen, die der Strahlrichtung entspricht. Dadurch werden Schnittpunkte mit der Geometrie, welche nahe der Strahlquelle liegen, zuerst gefunden. Alle Schnittpunkte mit der Geometrie, die in Strahlrichtung dahinter liegen, wären von dem zuerst gefundenen Schnittpunkt verdeckt und würden beim ungeordneten Traversieren sowieso wieder verworfen werden. Daher kann besonders bei großen Szenen mit viel Verdeckung ein großer Geschwindigkeitsvorteil aus der geordneten Traversierung gezogen werden. Es gibt allerdings auch einen kleinen Nachteil: Aus den gewöhnlichen Knoten der SSH und der BVH lässt sich nicht erkennen, welcher Kindknoten näher an der Strahlenquelle liegt. Die Knoten müssen also für die geordnete Traversierung mit dieser Information versehen werden. Dies lässt sich über die Speicherung der Schnittebene, die bei der Konstruktion verwendet wurde um den Knoten zu zerteilen, realisieren. Es reicht allerdings schon aus, die Orientierung der Ebene im Raum zu speichern und da diese immer orthogonal zu den Koordinatenachsen ist, werden dafür nur zwei Bits benötigt. Diese zwei Bits gehen also in die Flags der Knoten mit ein und reduzieren damit den Speicherplatz für die Referenzen auf die Kindknoten bzw. die Geometrie. Zusätzlich muss nun bei der Konstruktion darauf geachtet werden, dass sich der erste Kindknoten auf der entsprechenden Koordinatenachse immer vor dem zweiten Kindknoten befindet. Zusammen mit der Strahlrichtung kann dann bei der Traversierung ermittelt werden, welcher der beiden Kindknoten näher an der Strahlenquelle liegt und daher zuerst durchlaufen werden soll. Die geordnete Traversierung ist oftmals ungefähr doppelt so schnell wie die ungeordnete, weil die meisten Szenen aus geschlossenen Objekten bestehen, d.h. der Strahl schneidet die Objekte immer an mindestens zwei Stellen. (Details: Siehe 5.2).

5.4 Beleuchtungseffekte

Ein wichtiges Ziel bei der Implementierung der Beleuchtungseffekte war die Verwendung von SIMD-Instruktionen im gesamten Beleuchtungsvorgang. Dadurch können die Strahlenpakete (4 Strahlen), welche zum Schnitt mit der Geometrie verwendet werden, in den meisten Fällen direkt bei der Beleuchtung benutzt werden, ohne sie aufzuspalten. Wenn das Strahlenpaket

Szene	R & U	I & U	R & G	I & G
Bunny	1.382110	1.32566	0.692806	0.655291
Dragon	2.004170	1.89597	0.957832	0.911306
Ben[0]	0.755908	0.719973	0.428967	0.406139
Toasters[110]	2.348210	2.22207	1.695740	1.595580
Hand[0]	0.951925	0.906014	0.579745	0.553083
FairyForest[0]	2.838110	2.69045	1.826500	1.733270
Powerplant	7.366730	6.90946	2.941820	2.799040

Tabelle 5.2: Minimale Traversierungsdauer der SSH in Sekunden für eines von 30 gleichen Bildern ohne Multithreading. Die Tests erfolgten unter den in Kapitel 6.1 angegebenen Bedingungen. R: rekursive Traversierung. I: iterative Traversierung. U: ungeordnete Traversierung. G: geordnete Traversierung.

allerdings auf die Kante eines Dreiecks trifft, muss es aufgespalten und jeder Schnittpunkt einzeln beleuchtet werden, denn mindestens einer der vier Strahlen geht an dem Dreieck vorbei und trifft ein anderes Dreieck oder gar nichts. Es gibt also für beide Fälle (Paket oder einzelne Strahlen) eine Beleuchtungsfunktion, wobei für die SIMD-Variante (für Strahlenpakete) größtenteils nur die Datentypen durch SIMD-Datentypen ersetzt wurden, die bereits im ursprünglichen Raytracer vorhanden waren.
Eine weitere Zielsetzung ergab sich während der Implementierung: Die Sekundärstrahlen sollen für folgende Arbeiten z.B. mittels Frustum Tracing gebündelt verfolgt werden können, um die Effizienz zu steigern. Das bedeutet, dass die Strahlen nicht sofort nach der Erzeugung verfolgt werden können, sondern vorerst in einer Warteschlange gesammelt werden müssen. Nachdem alle Primärstrahlen verfolgt wurden, wird diese Warteschlange abgearbeitet. Dabei können neue Sekundärstrahlen entstehen, die wiederum in die Warteschlange eingereiht werden sollen. Diese Strahlen sollen aber auch wieder gebündelt werden können, so dass für sie eine neue Warteschlange erstellt werden muss. Da aber zur gleichen Zeit nur eine Warteschlange abgearbeitet und eine gefüllt wird, reichen zwei Warteschlangen aus, die nach dem Abarbeiten der aktuellen Warteschlange ausgetauscht werden. In der aktuellen Version des Raytracers wird kein Verfahren zur Bündelung verwendet, d.h. dieser zusätzliche Rechenaufwand ist nicht notwendig und führt nur zu niedrigeren Bildraten und höherem Speicherverbrauch. Darum gibt es auch die Möglichkeit die Sekundärstrahlen sofort, d.h. rekursiv zu verfolgen. Bei beiden Varianten wird das Beleuchtungsergebnis Schrittweise aufgebaut. Jeder Pixel beginnt mit der Farbe Schwarz und erhält zunächst durch die Verfolgung des Primärstrahls einen vorläufigen Wert, z.B. den ambienten Teil des Phong-Modells. Der Wert, welcher durch die Lichtquellen entsteht, trägt nur dann zum endgültigen Wert bei, wenn das Objekt nicht

im Schatten liegt. Darum wird dieser nicht sofort auf das Ergebnis aufaddiert, sondern im Schattenstrahl gespeichert. Erst wenn dieser zusammen mit den anderen Sekundärstrahlen verfolgt wurde und kein Objekt zwischen der Lichtquelle und der Oberfläche trifft, wird der Wert auf den Pixel aufaddiert. Für reflektierte und gebrochene Strahlen funktioniert es so ähnlich. Das reflektierte oder gebrochene Licht, welches an der Oberfläche ankommt, ist erst bekannt, wenn der Sekundärstrahl verfolgt wurde. Daher wird den Strahlen eine Referenz auf den Pixel mitgegeben und das Ergebnis wird aufaddiert, sobald es bekannt ist.

Die Beleuchtungseffekte, wie z.B. Phong und Eyelight, basieren alle auf einer gemeinsamen Klasse, die die Reflexionen und Refraktionen berechnet. Das bedeutet, dass jede Oberfläche unabhängig vom verwendeten Beleuchtungsmodell die Umgebung zu einem bestimmten Anteil reflektieren oder durchscheinen lassen kann. Wenn der Raytracer also um ein Beleuchtungsmodell erweitert wird, müssen Reflexion und Refraktion nicht neu implementiert werden. Diese Funktion stellt dabei nur eine Grundfunktion der Beleuchtungseffekte dar, d.h. sie kann beim Erstellen eines neuen Effekts auch ignoriert werden, falls dies gewünscht ist.

Zur Beleuchtung kann beim Start des Raytracers aus sechs Lichtsettings gewählt werden, darunter Punktlichtquellen, bei denen das Licht von einem Punkt in alle Richtungen strahlt, Flächenlichtquellen, die von verschiedenen Stellen in alle Richtungen strahlen und paralleles Licht, das nur in eine Richtung strahlt. Die Flächenlichtquellen erzeugen einen weichen Schatten, denn für jeden Primärstrahl und jede Lichtquelle werden gleich mehrere Schattenstrahlen verfolgt, so dass die Lichtquelle von einem anderen Objekt auch teilweise verdeckt sein kann, anstatt nur ganz oder gar nicht. Auch zur Berechnung der auf der Oberfläche einfallenden Lichtstärke werden mehrere Strahlen benutzt, so dass die Schattierung und die Lichtspiegelung auf dem Objekt mit der Größe und Entfernung der Lichtquelle variieren.

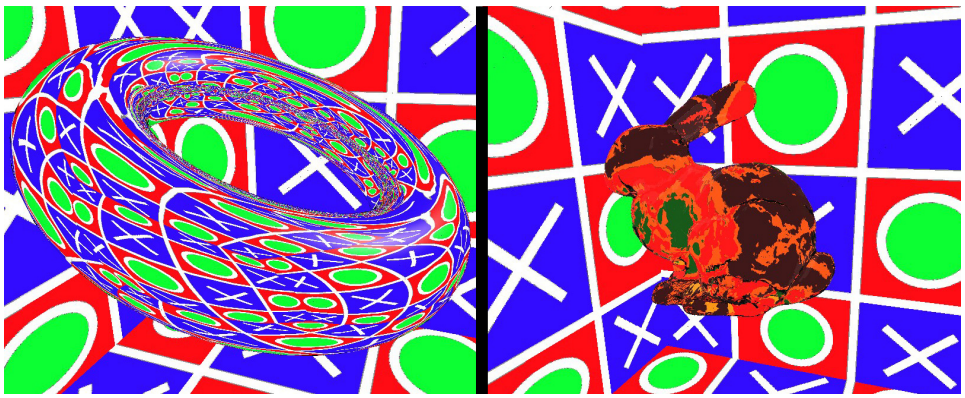

Abbildung 5.3: Beispiele für Reflexion und Refraktion. Links: Reflektierender Ring in einer Box. Rechts: Stanford Bunny aus rotem Glas in einer Box.

Kapitel 6

Tests

Dieses Kapitel beschreibt den praktischen Vergleich zwischen der Single Slab Hierarchy und der Bounding Volume Hierarchy mit dem erweiterten Raytracer, der in 5 beschrieben ist. In 6.1 ist der Testaufbau, d.h. die Einstellungen des Raytracers und die verwendeten Testdaten, beschrieben. 6.2 zeigt die Testergebnisse inkl. einer theoretischen Begründung.

6.1 Testaufbau

Für die Tests wurden 27 Szenen verschiedener Komplexität benutzt. Die Anzahl der Dreiecke pro Szene lag zwischen 3500 und 28 Millionen, wobei die Verteilung der Dreiecksanzahl unter den Szenen nicht gleichmäßig war, sondern exponentiell anstieg, d.h. 50% der Szenen hatten weniger als 100.000 Dreiecke und nur 6 Szenen hatten mehr als eine Millionen Dreiecke.
Der Raytracer lief vier Mal für jede Szene. Zwei Mal wurde eine SSH und zwei Mal eine BVH erstellt, um durch das Betriebssystem verursachte Verlangsamungen während der Konstruktion auszuschließen. Bei jedem Durchlauf wurden 30 Bilder erzeugt, um anfängliche Probleme durch Schreiben auf die Swap-Partition zu erkennen (Nach Konstruktion mit hohem Speicheraufwand bei großen Szenen).
Das Testsystem war mit einem Intel Core 2 Quad Q6600 und 4GB PC2-6400 CL4 RAM bestückt und lief mit dem Betriebssystem Ubuntu 8.04 64-Bit. Der Raytracer war mit den Compilerflags `-O3 -msse -march=native` optimiert und lief mit iterativer und geordneter Traversierung ohne Sekundärstrahlen.
Gemessen wurde die Konstruktionsdauer, die durchschnittliche Oberfläche der SSH-Knoten im Vergleich zur Oberfläche der BVH-Knoten, die durchschnittliche Anzahl an Schnitttests eines Strahls mit den Knoten, die durchschnittliche Anzahl an Schnitttests eines Strahls mit der Geometrie und die erste, letzte, minimale, maximale und durchschnittliche Traversierungsdauer (*Traversierung*: Schnitttest zwischen Strahl und Szene; Keine Beleuchtung).

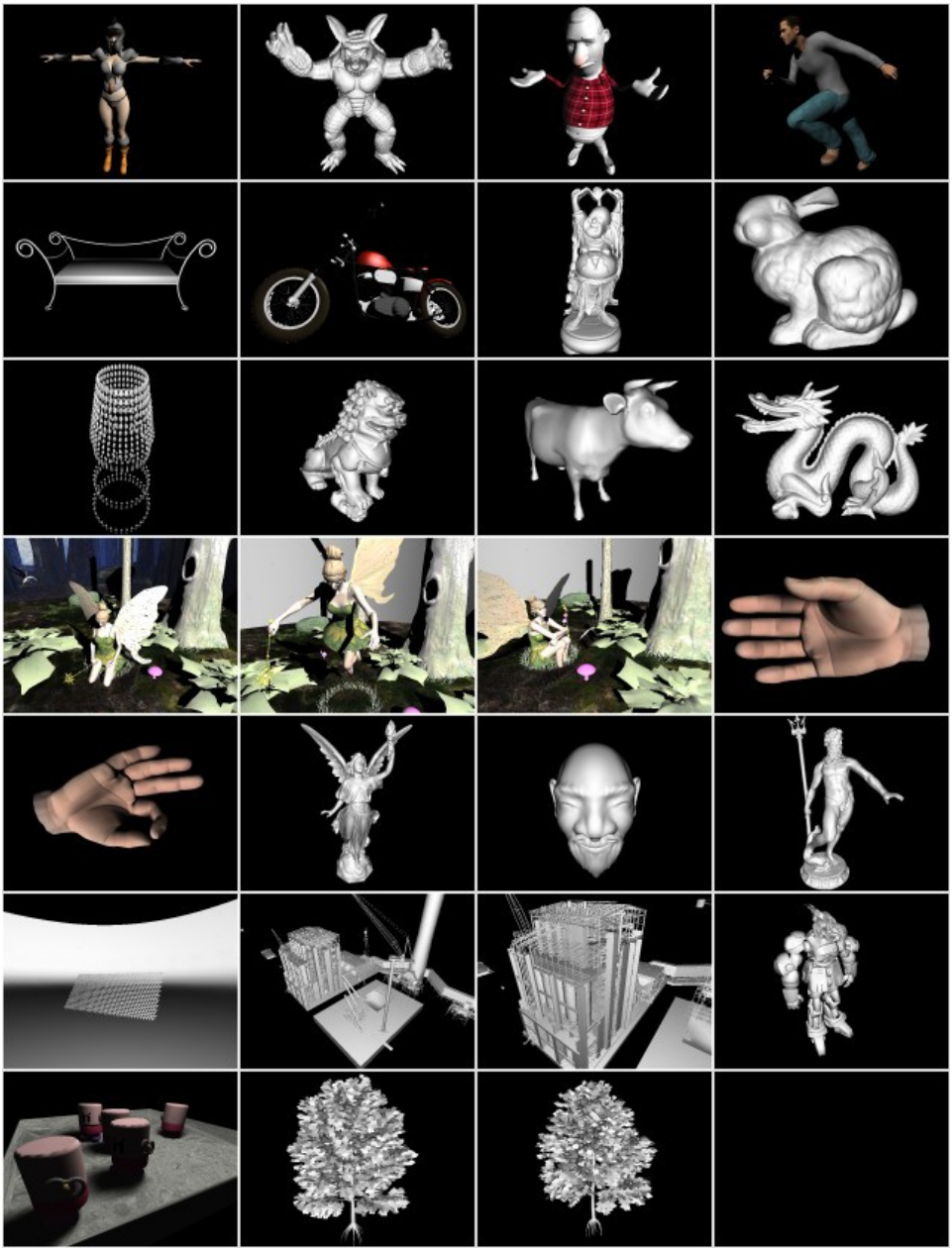

Abbildung 6.1: Verwendete Testszenen. Die Reihenfolge (in Leserichtung) entspricht der Reihenfolge in den Tabellen 6.1, 6.2 und 6.3

6.2 Ergebnisse

In Abbildung 6.2 zeigt die Kurve mit der Beschriftung „BVH Traversal Speedup" den Geschwindigkeitsunterschied zwischen SSH und BVH beim Traversieren für alle 27 Szenen. Der Wert wird als t_{SSH}/t_{BVH} berechnet, wobei t_{BVH} die minimale Traversierungsdauer der BVH und t_{SSH} die minimale Traversierungsdauer der SSH ist. Ein Wert über 1 bedeutet also, dass die Traversierung der BVH für die jeweilige Szene weniger Zeit gebraucht hat, als die Traversierung der entsprechenden SSH. Ein Wert unter 1 bedeutet, dass die Traversierung der SSH weniger Zeit in Anspruch genommen hat, als die der BVH. Die Werte formen ungefähr eine Gerade, die für die Hälfte der Szenen unter 1 und für die andere Hälfte über 1 liegt, d.h. die Geschwindigkeit beider Verfahren ist im Durchschnitt gleich. Der maximale Geschwindigkeitsunterschied liegt bei ca. 30%. Bei der Konstruktion der Hierarchie benötigte die BVH für nahezu jede Szene weniger Zeit als die SSH. Die einzige Ausnahme ist die Lucy-Szene aus dem Stanford 3D Scanning Repository, die so groß ist, dass sie bei der Konstruktion der BVH nicht in den Arbeitsspeicher passte und ein Teil auf die Swap-Partition ausgelagert wurde. Daher sollte dieses Ergebnis nicht gewertet werden. Der Geschwindigkeitsvorteil der BVH liegt bei der Konstruktion zwischen 0,372% und 9,447%.
Abbildung 6.2 zeigt neben den Geschwindigkeitsunterschieden noch eine weitere Kurve, die die durchschnittliche Anzahl der Schnitttests zwischen Strahl und SSH-Knoten im Vergleich zu den Schnitttest zwischen Strahl und BVH-Knoten angibt. Ein Wert von 2 bedeutet also, dass für die jeweilige Szene bei der SSH doppelt so viele Schnitttests zwischen Strahl und Knoten durchgeführt wurden wie bei der BVH. Die Werte liegen zwischen 1,58 und 2,78. Obwohl die SSH-Knoten nur $^1/_6$ der Informationen eines BVH-Knoten beinhalten und das Volumen eines BVH-Knoten nur durch mindestens sechs SSH-Knoten beschrieben werden kann, liegt der Wert deutlich unter 6. Dies liegt einerseits daran, dass durch den hierarchischen Aufbau der BVH viele Seiten der AABBs redundant sind und somit nicht sechs SSH-Knoten benötigt werden, um einen BVH-Knoten zu beschreiben, sondern höchstens drei. Andererseits ist der Baum bei der SSH genau so aufgebaut wie bei der BVH und nicht etwa sechs Mal so hoch. Es gibt also viele Strahlen, dessen Schnitttests in den Baum ähnlich tief vordringen, z.B. solche, die bis zu den Blattknoten vordringen und die Geometrie treffen.
Sieht man sich die Kurven vom Geschwindigkeitsunterschied der Traversierung und dem Verhältnis zwischen den Anzahlen der Schnitttests an, fällt auf, dass diese offenbar miteinander korrelieren. In Abbildung 6.2 sind die Szenen nach dem Geschwindigkeitsunterschied der Traversierungen geordnet. Beide Kurven bilden in etwa eine Gerade. Für jede der beiden Kurven ist deshalb in schwarzer Farbe eine approximierende Gerade und die dazugehörige Geradengleichung angegeben. Die Steigung der oberen Gerade ist

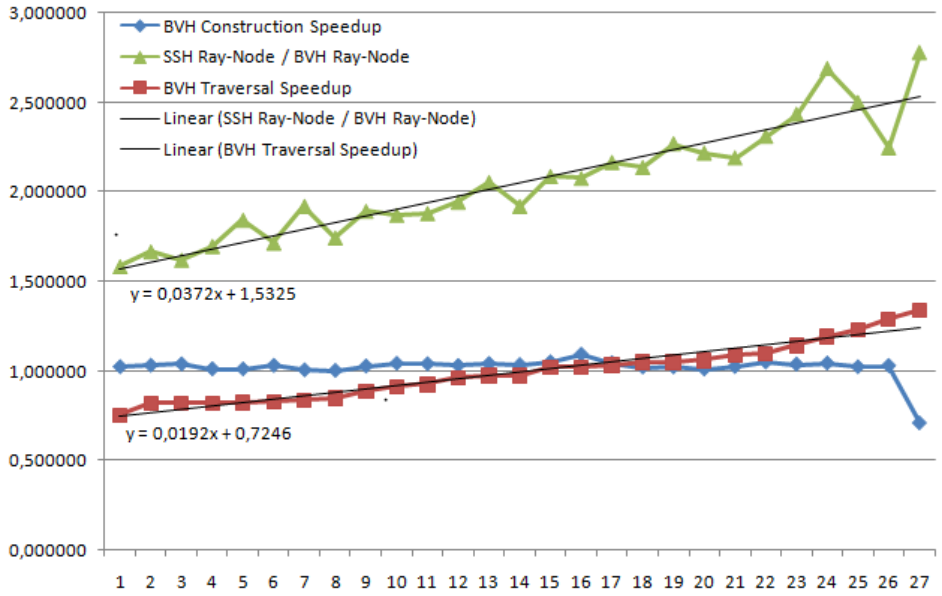

Abbildung 6.2: Geschwindigkeit im Vergleich zur Anzahl der Schnitttests für 27 Testszenen. Die dünnen schwarzen Geraden inkl. Geradengleichung sind lineare Approximationen der Kurven. *BVH Construction Speedup*: Geschwindigkeitsvorteil bei der Konstruktion der BVH. *BVH Traversal Speedup*: Minimale Traversierungsdauer der SSH geteilt durch die der BVH. *SSH Ray-Node / BVH Ray-Node*: Anzahl der Strahl-Knoten-Schnitttests bei der SSH geteilt durch die bei der BVH.

mit einer vernachlässigbaren Abweichung von 0,0012 ungefähr das Doppelte von der Steigung der unteren Gerade. Die Werte der oberen Gerade liegen mit einer Abweichung von 0,0833 ungefähr bei dem Doppelten der Werte der unteren Gerade. Für die Anzahl der Schnitttests bei der SSH i_{SSH}, die Anzahl der Schnitttests bei der BVH i_{BVH}, die Traversierungsdauer t_{SSH} der SSH und die Traversierungsdauer t_{BVH} der BVH gilt Formel 6.1.

$$\frac{i_{SSH}}{i_{BVH}} \approx 2 * \frac{t_{SSH}}{t_{BVH}} + 0,0833 \tag{6.1}$$

Formt man diese um und setzt die beiden Traversierungszeiten gleich, erhält man:

$$i_{SSH} \approx 2 * i_{BVH} + 0,0833. \tag{6.2}$$

Bei der Traversierung der SSH werden also in gleicher Zeit ungefähr doppelt so viele Schnitttests durchgeführt wie bei der Traversierung der BVH. Da der Strahl bei der SSH nur mit einer Ebene, bei der BVH aber mit einer AABB geschnitten werden muss, wäre die Annahme gewesen, dass der Schnitttest bei der SSH sechs Mal so schnell wäre. Die Schnitttests beider

Verfahren bestehen allerdings nicht nur aus dem Schnitttest zwischen Strahl und Ebene bzw. AABB, sondern auch aus der Überprüfung des Strahlintervalls, der geordneten Traversierung und den Rechenoperationen zum Durchlaufen der Kindknoten. Dieser zusätzliche Aufwand ist bei beiden Beschleunigungsstrukturen gleich, so dass der Geschwindigkeitsunterschied zwischen den Schnitttests mit der Ebene bzw. der AABB keinen allzu großen Einfluss mehr auf die Gesamtgeschwindigkeit hat und daher in gleicher Zeit nur doppelt so viele SSH-Knoten durchlaufen werden können wie BVH-Knoten. Das bedeutet, dass die Traversierung bei der SSH länger dauert als bei der BVH, wenn bei der SSH mehr als doppelt so viele Knoten durchlaufen werden müssen. Umgekehrt ist die Traversierung der SSH schneller als die der BVH, wenn bei der SSH weniger als doppelt so viele Knoten durchlaufen werden müssen. Die Annahme, dass die Anzahl der Schnitttests umso größer ist, je schlechter die AABB vom SSH-Knoten approximiert wird, hat sich in den Tests nicht bewahrheitet. Die Werte in Tabelle 6.3 in der Spalte *surface* entsprechen dem durchschnittlichen Verhältnis zwischen den SSH-Knotenoberflächen und den BVH-Knotenoberflächen. Ein Wert von 3 bedeutet zum Beispiel, dass die durchschnittliche Oberfläche eines SSH-Knotens dem Dreifachen des entsprechenden BVH-Knotens entspricht. Die Werte wurden mit 256-Bit Genauigkeit gemessen. Sie scheinen zu groß zu sein, jedoch ergab eine Messung mit 512-Bit Genauigkeit für einzelne Szenen (z.B. Dragon und Tree2) die gleichen Werte, d.h. bei der Messung sind keine numerischen Probleme aufgetreten.
Anhand der Knotenklassen kann der Speicherverbrauch berechnet werden. Ein SSH-Knoten besitzt einen Float-Wert und einen Verweis auf die Kindknoten bzw. die Geometrie. Dieser Verweis ist als Index auf eine Knoten- bzw. Dreiecksliste implementiert, d.h. es kann jeder Datentyp verwendet werden. Von der Variable für den Index werden 5 bzw. 3 Bits als Flags benutzt, je nachdem ob geordnete Traversierung benutzt wird oder nicht. Ein Datentyp der Größe 32 Bit bietet also im Fall der geordneten Traversierung 27 Bits zum Speichern des Index. Damit können 2^{27} Knoten bzw. Dreiecke adressiert werden, wobei die Anzahl der Dreiecke kleiner ist, als die Anzahl der Knoten. Die Frage ist also, wie viele Dreiecke sich aus 2^{27} Knoten ergeben. Die Anzahl der Knoten für n Dreiecke ist $2 \cdot n - 1$. Somit ist die maximale Anzahl an Dreiecken für einen 32 Bit Datentyp $\lfloor \frac{2^{27}+1}{2} \rfloor = 67.108.864$. Ein 32 Bit Datentyp sollte also für die meisten Szenen ausreichen. Der Speicherverbrauch eines SSH-Knotens liegt damit bei 8 Bytes. Für eine 64 Bit große Variable für Index und Flags liegt der Speicherverbrauch bei 12 Bytes. Ein BVH-Knoten beinhaltet sechs Float-Werte für die AABB und ebenfalls einen Verweis auf die Kindknoten bzw. die Geometrie. Damit ergibt sich ein Speicherverbrauch von 28 Bytes für eine 32 Bit große Variable für Index und Flags und 32 Bytes für eine 64 Bit große Variable. Somit liegt der Speicherverbrauch der SSH mit einer 32 Bit Variable bei 28,6% und mit einer 64 Bit Variable bei 37,5% von dem einer BVH.

Szene	Dreiecke	c_{SSH}	c_{BVH}	t_{SSH}	t_{BVH}
Amazon	3.508	0,004909	0,004703	0,185350	0,202638
Armadillo	345.944	0,639331	0,624774	0,629549	0,599122
Barney	9.402	0,013404	0,012801	0,345723	0,334425
Ben[0]	78.029	0,121912	0,117346	0,411001	0,359241
Bench	3.783	0,004778	0,004718	0,140580	0,170825
Bike	459.999	0,771929	0,747365	2,656140	3,245660
Buddha	1.087.716	1,978020	1,927930	0,588303	0,539906
Bunny	69.451	0,096297	0,092365	0,646593	0,663366
Chandelier	9.216	0,011527	0,011031	0,373339	0,313086
Chinese dragon	1.311.956	1,915230	1,857430	0,825183	0,639022
Cow	5.804	0,007489	0,007120	0,381752	0,374090
Dragon	871.414	1,569100	1,550240	0,890888	0,839874
FairyForest[0]	174.117	0,312926	0,303151	1,706400	2,061360
FairyForest[200]	174.117	0,316583	0,315409	2,039060	2,401560
FairyForest[80]	174.117	0,300989	0,292689	2,218070	2,501550
Hand[0]	15.855	0,022193	0,021510	0,547713	0,568779
Hand[15]	15.855	0,023530	0,021499	0,408236	0,399967
Lucy	28.055.742	51,015700	71,926100	1,040600	0,776955
Ming Head	61.103	0,794470	0,755870	0,435382	0,396335
Neptune	4.007.872	6,867220	6,703640	0,718727	0,582793
Net	33.136	0,062617	0,060205	1,236450	1,506120
Powerplant view1	12.748.510	28,922100	28,574000	2,726890	3,311150
Powerplant view2	12.748.510	28,763900	28,568200	5,573800	6,656930
Robot	16.906	0,025799	0,025213	0,375193	0,357063
Toasters[110]	11.141	0,016878	0,016469	1,571060	2,082410
Tree1	133.986	0,206257	0,198970	0,766461	0,786287
Tree2	95.641	0,145458	0,139742	0,605674	0,653996

Tabelle 6.1: Testergebnisse für 27 Szenen. c_{SSH} *bzw.* c_{BVH}: Konstruktionsdauer der SSH bzw. BVH in Sekunden. t_{SSH} *bzw.* t_{BVH}: Minimale Traversierungssdauer der SSH bzw. BVH in Sekunden.

Szene	in_{SSH}/ray	in_{BVH}/ray	it_{SSH}/ray	it_{BVH}/ray
Amazon	18,766240	10,032200	1,496328	0,564272
Armadillo	68,256800	30,089160	4,823840	2,318164
Barney	37,172960	17,171760	2,974336	1,085168
Ben[0]	44,218800	18,175280	3,167632	1,207344
Bench	13,418080	7,907960	0,938360	0,632720
Bike	269,518000	161,743600	35,242880	24,298160
Buddha	56,555600	25,816800	6,129040	2,525560
Bunny	72,064400	35,099200	5,268040	2,194064
Chandelier	42,844400	15,940480	2,550168	0,959780
Chinese dragon	69,060400	30,746240	7,907960	3,138632
Cow	40,308400	19,321160	3,292044	1,165988
Dragon	92,858800	41,884800	8,456640	3,784168
FairyForest[0]	182,031600	106,098000	16,947280	10,129600
FairyForest[200]	214,349200	122,935600	21,261200	12,472160
FairyForest[80]	243,500800	128,700800	20,892324	12,833720
Hand[0]	58,416400	30,059160	5,299760	1,947240
Hand[15]	42,642400	20,546000	3,905832	1,416564
Lucy	105,076400	37,842760	7,736160	2,375848
Ming Head	46,223200	20,012880	3,791668	1,486028
Neptune	71,145600	28,476400	6,523680	2,130532
Net	123,922800	76,471600	14,517000	8,185880
Powerplant view1	320,904800	174,208800	21,354080	16,980280
Powerplant view2	720,808000	375,392800	26,868760	18,732000
Robot	38,647960	18,085640	3,642124	1,328892
Toasters[110]	187,391600	118,288800	12,985480	5,249280
Tree1	80,548800	41,966000	6,854480	2,485692
Tree2	65,119600	34,641000	5,435240	1,924676

Tabelle 6.2: Testergebnisse für 27 Szenen. in_{SSH}/ray *bzw.* in_{BVH}/ray: Durchschnittliche Anzahl der Schnitttests zwischen Strahl und Knoten pro Strahl. it_{SSH} *bzw.* it_{BVH}: Durchschnittliche Anzahl der Schnitttests zwischen Strahl und Dreieck pro Strahl.

Szene	mem_{SSH}	mem_{BVH}	height	surface
Amazon	84.180	224.480	19	3,209970
Armadillo	8.302.644	22.140.384	25	4,426860
Barney	225.636	601.694	25	5,277090
Ben[0]	1.872.684	4.993.824	38	3,570200
Bench	90.780	242.080	23	2,887110
Bike	11.039.964	29.439.904	63	9,660140
Buddha	26.105.172	69.613.792	36	2269,41
Bunny	1.666.812	4.444.832	24	3,477310
Chandelier	221.172	589.792	17	2,722920
Chinese dragon	31.486.932	83.965.152	34	3,745280
Cow	139.284	371.424	20	3,131870
Dragon	20.913.924	55.770.464	35	4087,45
FairyForest[0]	4.178.796	11.143.456	42	445,355000
FairyForest[200]	4.178.796	11.143.456	43	412,658000
FairyForest[80]	4.178.796	11.143.456	41	524,159000
Hand[0]	380.508	1.014.688	26	6,463090
Hand[15]	380.508	1.014.688	25	3,488030
Lucy	673.337.796	1.795.567.456	40	497,418
Ming Head	1.466.460	3.910.560	136	3,607860
Neptune	96.188.916	256.503.776	35	3,711830
Net	795.252	2.120.672	29	3,467910
Powerplant view1	305.964.228	815.904.608	63	31,581700
Powerplant view2	305.964.228	815.904.608	63	31,581700
Robot	405.732	1.081.952	26	4,070380
Toasters[110]	267.372	712.992	29	7,287180
Tree1	3.215.652	8.575.072	33	47,137900
Tree2	2.295.372	6.120.992	30	921,324000

Tabelle 6.3: Testergebnisse für 27 Szenen. mem_{SSH} *bzw.* mem_{BVH}: Speicherverbrauch der SSH bzw. BVH in Bytes. *height*: Höhe des Baums (bei SSH und BVH gleich). *surface*: Verhältnis der Knotenoberflächen von den SSH-Knoten zu den BVH-Knoten. Die Werte wurden mit 256 Bit Genauigkeit gemessen. Eine vereinzelte Messung mit 512 Bit Genauigkeit (bei Tree2 und Dragon) ergab die selben Werte, d.h. numerische Probleme bei der Messung sind auszuschließen.

Kapitel 7

Zusammenfassung und Ausblick

Das Ergebnis dieser Arbeit ist ein interaktiver Raytracer, der Szenen aus Dateien verschiedenen Dateityps laden und sie dank SIMD-Instruktionen und Multithreading sehr effektiv mit verschiedenen Beleuchtungseffekten darstellen kann. Die ursprüngliche Implementierung der Beschleunigungsstrukturen wurde derart angepasst, dass die SSH und die BVH optimal verglichen werden können. Außerdem bietet die neue generische Basisklasse der beiden Beschleunigungsstrukturen die Möglichkeit, neue BVH-Derivate effizient zu implementieren. Die neuen Knotenklassen der SSH und der BVH sind leichter anpassbar, um z.B. den Speicherverbrauch durch Nutzung einer Heap-Struktur und dem Verzicht auf den Verweis der Kindknoten noch weiter zu senken. Über die Startparameter kann der Raytracer in verschiedene Modi versetzt werden, um z.B. eine Szene interaktiv mit einer Kamera zu durchfliegen oder automatische Tests mit einer wählbaren Abfolge von Beschleunigungsstrukturen und Szenen durchzuführen und deren Ergebnisse in eine Textdatei auszugeben. Mit der Unterstützung einer iterativen Verfolgung von Sekundärstrahlen ist die Grundlage für weitere Beschleunigungstechniken, wie z.B. Frustum Tracing, gelegt.
Der praktische Vergleich der SSH mit der BVH ergab, dass die SSH für einige Szenen mit bis zu 30% schneller traversiert werden kann als die BVH, jedoch trat auch der umgekehrte Fall auf. Der durchschnittliche Geschwindigkeitsunterschied liegt ungefähr bei 0%. Ein erhoffter Zusammenhang zwischen der Geschwindigkeit und dem Verhältnis der Knotenoberflächen konnte nicht festgestellt werden, so dass der Geschwindigkeitsunterschied im Voraus bislang nicht berechenbar ist. Der um 62,5% bzw. 71,4% geringere Speicherverbrauch im Vergleich zur BVH macht die SSH jedoch auch bei möglichen Geschwindigkeitseinbußen zu einer attraktiven Alternative. Mit dem in Kapitel 3 gezeigten Verfahren von Cline *et al.* zur Diskretisierung des Raums ließe sich der absolute Speicherverbrauch sogar noch weiter verringern.

Literaturverzeichnis

[AK89] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In Andrew S. Glassner, editor, *An Introduction to Ray Tracing*, pages 206–208. Academic Press, 1989.

[App68] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conference*, volume 32, pages 37–45, 1968.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[CSE06] David Cline, Kevin Steele, and Parris Egbert. Lightweight Bounding Volumes for Ray Tracing. *Journal of Graphics Tools: JGT*, 11(4):61–71, 2006.

[EWM08] Martin Eisemann, Christian Woizischke, and Marcus Magnor. Ray Tracing with the Single-Slab Hierarchy. In *Proc. Vision, Modeling, and Visualization (VMV'08)*, Konstanz, Germany, 10 2008.

[FTI86] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. *IEEE Compututer Graphics and Applications*, 6(4):16–26, April 1986.

[Gla84] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, October 1984.

[HHS06] Vlastimil Havran, Robert Herzog, and Hans-Peter Seidel. On fast construction of spatial hierarchies for ray tracing. Research Report MPI-I-2006-4-004, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, June 2006.

[Kap85] M. R. Kaplan. Space tracing a constant time ray tracer. In *State of the Art in Image Synthesis (Course Notes on ACM SIGGRAPH '85*, volume 11, pages 149–158, July 1985.

[KK86] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 269–278, New York, NY, USA, 1986. ACM Press.

[MB90] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(3):153–166, 1990.

[OSDM87] B.C. Ooi, R. Sacks-David, and K.J. McDonnel. Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE International Computer Software and Applications Conference (COMPSAC)*, pages 433–438, Tokio, Japan, October 1987.

[PH04] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[RW80] S.M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer graphics and interactive techniques*, pages 110–116, New York, NY, USA, 1980. ACM Press.

[SM03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition, 2003.

[WBWS01] I. Wald, C. Benthin, M. Wagner, and P. Slusallek. Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, 20(3):153–164, 2001.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.

[WK06] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In T. Akenine-Möller and W. Heidrich, editors, *Rendering Techniques 2006 (Proc. of 17th Eurographics Symposium on Rendering)*, pages 139–149, 2006.

[WMS06] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, pages 67–77, 2006.