UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Darstellung von Halbschatten durch Verwendung des "Cascaded Linespace"-Verfahrens

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Nicolas Klee

Erstgutachter:	Prof. Dr.-Ing. Stefan Müller (Institut für Computervisualistik, AG Computergraphik)
Zweitgutachter:	M.Sc. Kevin Keul (Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Januar 2016

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Zusammenfassung

Diese Arbeit präsentiert einen Ansatz zur Optimierung der Berechnung von Halbschatten flächiger Lichtquellen. Die Lichtquelle wird durch Sampling uniform abgetastet. Als Datenstruktur wird ein N-tree verwendet, durch den die Strahlen als Paket traversiert werden. Der N-tree speichert in seinen Knoten einen Linespace, der Informationen über Geometrie innerhalb eines Schaftes bietet. Diese Sichtbarkeitsinformation wird als Kriterium für eine Terminierung eines Strahles genutzt. Zusätzlich wird die Grafikkarte (kurz GPU – engl. „graphics processing unit“) zur Beschleunigung durch Parallelisierung benutzt. Die Szene wird zunächst mit OpenGL gerendert und anschließend der Schattenwert für jedes Pixel auf der GPU berechnet. Im Anschluss werden die CPU- und GPU-Implementationen verglichen. Dabei zeigt die GPU-Implementation eine Beschleunigung von 86% gegenüber der CPU-Implementation und bietet eine gute Skalierung mit zunehmender Dreieckszahl. Die Verwendung des Linespace beschleunigt das Verfahren gegenüber der Durchführung von Schnitttests und zeigt für eine große Anzahl an Strahlen keine visuellen Nachteile.

Abstract

This thesis presents an approach to optimizing the computation of soft shadows from area lights. The light source is sampled uniformly by traversing shadow rays as packets through an N-tree. This data structure stores an additional line space for every node. A line space stores precomputed information about geometry inside of shafts from one to another side of the node. This visibility information is used to terminate a ray. Additionally the graphics processing unit (short GPU) is used to speed up the computations through parallelism. The scene is rendered with OpenGL and the shadow value is computed on the GPU for each pixel. Evaluating the implementation shows a performance gain of 86% by comparison to the CPU, if using the GPU implementation. Using the line space instead of triangle intersections also increases the performance. The implementation provides good scaling with an increasing amount of triangles and has no visual disadvantages for many rays.

1 Einleitung

Die physikalisch korrekte Berechnung von Licht in der Computergrafik, stellt seit den Anfängen ein Kernproblem dar. Dabei sind indirekte Beleuchtung, sowie Verschattung zwei essentielle Probleme. Eine realitätsnahe Beleuchtung einer Szene sorgt für mehr Glaubwürdigkeit, sowie eine bessere räumliche Einordnung einzelner Elemente der Szene.
Lichtquellen treten in der realen Welt als Körper auf. Beispielsweise emittiert eine Glühbirne nicht aus einem Punkt Photonen in die Welt, sondern von einem glühenden Draht. Ein anderes Beispiel bietet eine Leuchtstoffröhre, die auf ihrer gesamten Länge Photonen in die Welt emittiert.
In der Computergrafik werden Lichtquellen meistens als Punktlichtquellen dargestellt. Raytracing von diesen führt beispielsweise zu harten Schattenkanten, da für jeden Strahl nur die Bedingung „Schatten“ oder „Licht“ erfüllt sein kann. Diese Repräsentation von Lichtquellen ist demnach für eine realistische Beleuchtung nicht ausreichend. Zur Berechnung von Halbschatten oder auch weichen Schatten (engl. „Soft Shadows“) werden flächige Lichtquellen benötigt. Diese können anders als Punktlichtquellen nicht mit einem Strahl abgetastet werden, sondern benötigen eine Abtastung der gesamten Fläche der Lichtquelle. Hierzu kann die Lichtquelle mit einer bestimmten Anzahl von Strahlen abgetastet werden und der Schattenwert anschließend gemittelt werden (engl. Sampling).

Das in dieser Arbeit vorgestellte Verfahren nutzt Sampling zum Abtasten einer flächigen Lichtquelle. Als Datenstruktur wird das „Cascaded Linespace“-Verfahren zur Beschleunigung der Berechnungszeiten genutzt. Dieses basiert auf einem hierarchischen N-tree und besitzt zusätzlich für jeden Knoten einen Linespace. Bei einem Linespace handelt es sich um vorberechnete Schäfte innerhalb des Knoten und bietet eine Möglichkeit Schnitttests zu vermeiden. Die Information auf Geometrie innerhalb eines Schaftes wird als Kriterium für eine Terminierung eines Strahles genutzt. Zusätzlich wird ein Paket-basierter Ansatz genutzt, um viele Strahlen gleichzeitig durch die Datenstruktur zu verfolgen.
Beschleunigt wird das Verfahren durch Verwendung von klassischem GPU-Rendering in Verbindung mit der Nutzung des Compute-Shader zur Berechnung der Schatten. Hierfür muss der N-tree und der Linespace als Datenstruktur auf der GPU zur Verfügung gestellt und traversiert werden.

Zunächst wird eine Einführung in die Besonderheiten weicher Schatten in Unterabschnitt 3.1 gegeben. Um das „Cascaded Linespace“-Verfahren einzuführen werden in Unterabschnitt 3.2 zuerst aktuelle Datenstrukturen miteinander verglichen und danach das „Cascaded Linespace“-Verfahren in Unterabschnitt 3.3 beschrieben.

Anschließend wird die Implementation in Abschnitt 4 beschrieben. Hierbei wird zunächst in Unterabschnitt 4.1 die Beschleunigung durch die GPU beschrieben, indem die hier genutzte Render-Pipeline beschrieben wird. Danach wird auf die Abbildung des N-tree und Linespace auf Texturen eingegangen und deren Eigenschaften in Unterabschnitt 4.2 beschrieben.
Die Traversierung dieser Datenstrukturen wird in Unterabschnitt 4.3 eingeführt und in Unterunterabschnitt 4.3.1 deren Implementation erläutert. Die Durchführung der Linespace-Überprüfung wird anschließend in Unterunterabschnitt 4.3.2 erläutert.

Um das Verfahren zu evaluieren werden die Ergebnisse verschiedener Tests in Abschnitt 5 zusammengefasst. Hierbei werden unter anderem Berechnungszeiten der Schatten gemessen und miteinander verglichen. Zusätzlich wird auf die Beschleunigung durch die GPU eingegangen und die Skalierbarkeit des Verfahrens mit zunehmender Dreieckszahl analysiert. Abschließend wird das Verfahren mit dem DDA-Algorithmus verglichen, um es in einen Kontext mit aktuellen Verfahren zu setzen.
Die Ergebnisse werden in Abschnitt 6 zusammengefasst und ein Ausblick auf weitere Optimierungsmöglichkeiten gegeben.

2 Andere Arbeiten

Halbschatten
Die Darstellung von flächigen Lichtquellen (engl. Area-Lights) stellt verschiedene Herausforderungen an die Computergrafik. Die Beleuchtung eines Pixels nach klassischem Phong-Beleuchtungsmodell ist nicht ausreichend, da ein Area-Light aus mehreren Vertices besteht. Um das Problem der Beleuchtung anzugehen, präsentierte Snyder 1996 in [Sny96] mehrere Möglichkeiten für Phong- und Lambert-Beleuchtungsmodelle für Area-Lights.

Das Problem der Schattenberechnung wird auf mehrere Weisen angegangen. Für eine exakte Darstellung von Halbschatten, ist das Sampling der Lichtquelle ein häufig verwendetes Verfahren. Dabei wird die Lichtquelle mit einer bestimmten Anzahl an Strahlen abgetastet und der Schattenwert aus dem Mittelwert der blockierten Strahlen gebildet. Ein großes Problem beim Sampling ist Rauschen, welches durch zu geringe Strahldichten entsteht. Ramamoorthi zeigt hierfür in [RAMN12] beispielsweise Optimierungsmöglichkeiten durch eine nicht-uniforme Verteilung der Schattenstrahlen über die Lichtquelle.
Da das Sampling durch das Verwenden vieler Strahlen aufwändig in der Berechnung wird, existieren verschiedene Ansätze zur Optimierung des Verfahrens. Ein Optimierungsansatz zur Verringerung des Berechnungs-

aufwands sind Packet-basierte Verfahren. Hierbei werden die Schattenstrahlen nicht einzeln überprüft sondern als Paket zusammen untersucht. Dadurch wird die Cache-Effizienz erhöht, da mehrere Strahlen gleiche Dreiecke schneiden können und somit die Traversierung durch die verwendete Datenstruktur einmalig stattfindet. Ein Beispiel für diesen Ansatz bietet das von Overbeck *et al.* in [ORM07] vorgestellte Beamtracing. Zusätzlich zu einem Paket-basierten Aufbau, werden in diesem Verfahren keine Strahlen sondern ein Schacht aus vier Strahlen verfolgt und dieser bei einem Treffer in kleinere Schächte aufgeteilt.

Zum Erreichen echtzeitfähiger Schattenberechnungen, werden überwiegend Shadow-Map-basierte Verfahren verwendet. Diese können auch für die Darstellung von Halbschatten angewendet werden. So vergleichen Hasenfratz *et al.* in ihrer Studie [HLHS03], mehrere Shadow-Map-basierte Verfahren miteinander. Als Beispiel für ein solches Verfahren, zeigen Annen *et al.* in [ADM+08] eine Möglichkeit zur Berechnung weicher Schatten, durch Filterung mehrerer Shadow-Maps.
Neuere Verfahren nutzen einen neuen Ansatz zur Repräsentation der Szene. Die Szene wird, wie von Crassin in [CNE09] vorgestellt, zunächst voxelisiert. Über diese voxelisierte Szene kann durch Voxel-Cone-Tracing traversiert werden. Dieser Ansatz wird vorrangig für indirekte Beleuchtung eingesetzt, wie in [CNS+11] gezeigt, kann aber auch zur Darstellung von weichen Schatten verwendet werden, wie in [CNE09] beschrieben wird. Hierzu werden die Voxel innerhalb des Schatten-Kegels akkumuliert und anschließend der Schattenwert aus dem akkumulierten alpha-Wert ausgelesen.

Datenstrukturen
Zur Optimierung Raytracing-basierter Verfahren, existieren verschiedene Datenstrukturen, für eine effektive Speicherung der Szene. Unter anderem verwendete Datenstrukturen sind hierbei das Uniform-Grid, der Octree, kd-trees und Bounding-Volume-Hierarchien. Diese Datenstrukturen werden in Unterabschnitt 3.2 genauer erläutert. Um über diese Datenstrukturen auf der GPU zu traversieren, existieren verschiedene Ansätze. Einen Raycasting-Algorithmus über einen Octree auf der GPU zeigen Gobetti *et al.* in [GMG08]. Zudem zeigen Laine und Karras in [LK10] eine effiziente Speicherung eines Octree im Speicher der GPU, sowie einen Stack basierten Raycast-Algorithmus über diese Datenstruktur.
Foley und Sugerman zeigen in [FS05] eine Möglichkeit der GPU-basierten kd-tree Traversierung, welche von Horn *et al.* in [HSHH07] erweitert wurde. Hierbei werden Pakete von Strahlen verfolgt und ein short-stack-basierter Ansatz, zur Vermeidung einer Rekursion, verwendet.

Diese Arbeit baut auf dem Paket basierten Ansatz von Wald *et al.* in

[WIK+06] auf. Dieser führt das Coherent-Grid-Traversal-Verfahren ein, unter Verwendung eines Uniform-Grid. Die Verwendung dieses Verfahrens für Sekundärstrahlen, wie Schattenstrahlen, zeigen Gribble *et al.* in [GIK+07]. Die Möglichkeit der Erweiterung für hierarchische Datenstrukturen, wie dem Octree, zeigen Knoll *et al.* in [KWH09].

3 Grundlagen

Dieses Kapitel behandelt Methoden und Besonderheiten der Darstellung von Halbschatten. Weiter wird auf Datenstrukturen eingegangen, die die Berechnungszeiten dieser Methoden verringern und abschließend das „Cascaded-Linespace"-Verfahren beschrieben und dessen Vorteile erläutert.

3.1 Halbschatten

Im Gegensatz zu einer Punktlichtquelle, bietet eine flächige Lichtquelle einen Übergang vom Schatten in ein beleuchtetes Gebiet. Der Kernschatten wird hierbei, wie auch in [HLHS03], als „Umbra" bezeichnet und der Halbschatten als „Penumbra". Die Zusammenhänge werden in Abbildung 1 dargestellt.

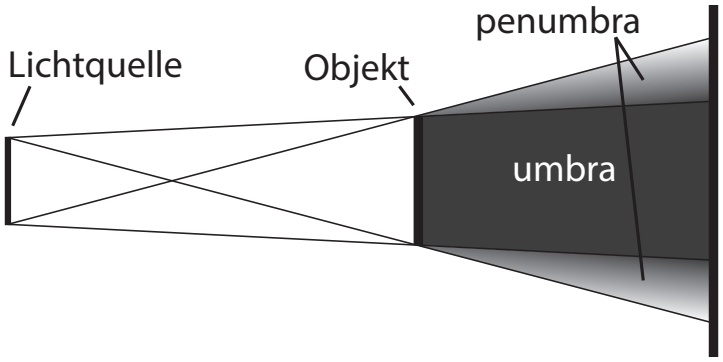


Abbildung 1: Besonderheit einer flächigen Lichtquelle gegenüber einer Punktlichtquelle. Die Penumbra ist der Bereich des Halbschattens, die Umbra ist der Kernschatten.

Bei der Darstellung von Halbschatten existieren verschiedene Ansätze. Im Gegensatz zu physikalisch korrekten Verfahren, versuchen andere Verfahren ein möglichst genaues Ergebnis zu approximieren, dieses jedoch in kürzerer Zeit zu berechnen. Ein Beispiel hierfür bilden Shadow-Map-basierte Verfahren, wie das von Fernando in [Fer05] vorgestellte „Percentage-closer soft shadows"-Verfahren. Hierbei werden weiche Schatten aus einer Shadow-Map berechnet, indem die Penumbra-Größe geschätzt wird. An-

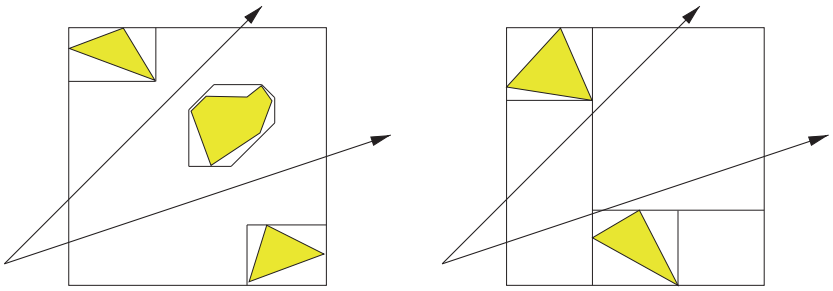
schließend wird durch das von Reeves *et al.* in [RSC87] vorgestellte „percentage-closer filtering“ der Schattenwert aus mehreren Werten der Shadow-Map berechnet.
Durch diese Annäherung bietet das Verfahren kein physikalisch korrektes Ergebnis. Dies ist jedoch ausreichend für echtzeitfähige Anwendungsfälle.

Um ein physikalisch korrektes Ergebnis zu erzielen, kann die Lichtquelle durch Sampling abgetastet werden. Anschließend wird die Anzahl der blockierten Strahlen durch die Anzahl aller gesendeten Strahlen dividiert. Dieses Verfahren benötigt jedoch viele Strahlen für ein plausibles Ergebnis, da ansonsten Rauschen auftritt (siehe Abbildung 11).

3.2 Datenstrukturen

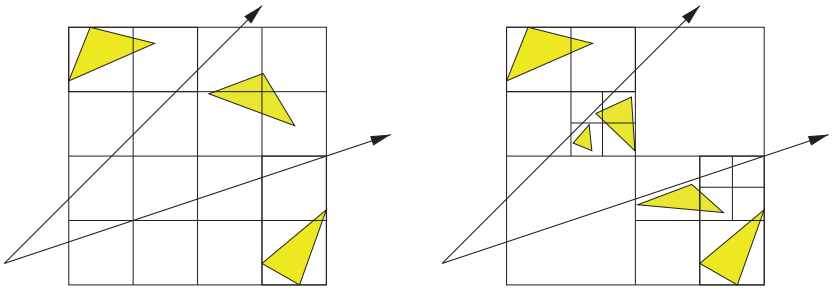
Die Strahlenverfolgung geschieht mittels Raytracing nach Appel, vorgestellt in [App68]. Dabei werden die Dreiecke der Szene mit dem Strahl geschnitten und nach dem ersten Schnittpunkt des Strahles entlang seiner Richtung gesucht. Jedoch stellen die Schnitttests einen rechenintensiven Teil des Algorithmus dar. Um die Anzahl der Schnitttests zu verringern existieren verschiedene Datenstrukturen zur effizienteren Raumaufteilung der Szene. Diese versuchen das Traversieren durch leeren Raum effizienter zu gestalten und möglichst relevante Dreiecke zu testen.

Zwei Beispiele für Datenstrukturen, die sich flexibel an die Objekte anpassen, sind Bounding-Volume-Hierarchien (kurz: BVH) und kd-Bäume (engl. kd-tree). Erstere erstellen um Objekte eine Box, die sogenannte Bounding-Box (siehe Abbildung 2(a)). Diese kann sich, wie von Kay und Kajiya in [KK86] gezeigt, flexibel an die Form der Geometrie anpassen. Strahlen werden zuerst mit den Bounding-Boxen geschnitten und bei einem erfolgreichen Schnitttest die Dreiecke innerhalb der Box. Dadurch müssen viele Dreiecke nicht überprüft werden, wodurch die Rechenzeit verringert wird. Bei einem hierarchischen kd-tree handelt es sich um einen Binärbaum nach [Ben75]. Hierbei besitzt ein Knoten maximal zwei Unterknoten. Ein Knoten ist ein Blatt, wenn dieser keine weiteren Unterknoten besitzt. Wie in Abbildung 2(b) zu sehen, kann dadurch der Raum ähnlich der BVH, schnell traversiert werden.



(a) BVH (b) kd-tree

Abbildung 2: Hierarchische Datenstrukturen
(a) Strahlen werden mit den Bounding-Boxes der Geometrie geschnitten und bei einem Treffer die Geometrie selbst, bzw. weitere Bounding-Boxes innerhalb.
(b) Strahlen werden mit beiden Unterknoten des Wurzelknotens geschnitten und bei einem Treffer die Traversierung in dem entsprechenden Teilbaum fortgesetzt.



(a) Uniform-Grid (b) Octree

Abbildung 3: Gitter
(a) Das Uniform-Grid besteht aus einem gleichmäßigen Gitter. Ein Strahl traversiert von Zelle zu Zelle durch das Gitter und testet die Geometrie innerhalb der Zelle.
(b) Ein Octree besteht aus einer Gitterauflösung von 2x2x2, unterteilt die Knoten wiederum in ein 2x2x2 Gitter, wenn diese Geometrie beinhalten. Ein Strahl traversiert die Knoten auf der obersten Ebene und führt die Traversierung bei einer weiteren Unterteilung des Knoten in dessen Octree durch.

Abbildung 3(a) zeigt den Aufbau eines Uniform-Grid. Dieses teilt den Raum gleichmäßig in ein dreidimensionales Raster auf und speichert Dreiecke in den jeweiligen Zellen. Der nicht-hierarchische Aufbau des Uniform-Grid ist dabei ein großer Nachteil. Anders als bei der BVH oder dem kd-tree ist das Uniform-Grid statisch und passt sich nicht an die Geometrie an. Es bietet den Vorteil einer hohen Auflösung und damit weniger Dreiecken pro Zelle.
Darauf aufbauend führt der Octree von Glassner aus [Gla88] eine hierarchische Struktur ein, in der ein Uniform-Grid der Ausmaße 2x2x2 weiter unterteilt wird (siehe Abbildung 3(b)). Wenn eine Zelle Geometrie beinhaltet, wird diese Zelle in acht kleinere Zellen unterteilt. Dies wird bis zu einer maximalen Tiefe fortgeführt. Leere Bereiche im Raum können somit schnell übersprungen werden, wohingegen Bereiche mit Geometrie stark unterteilt sind, sodass weniger Schnitttests benötigt werden.

3.3 Cascaded-Linespace Verfahren

Ein N-tree ist eine Erweiterung des Octree mit einstellbarer Auflösung. Eine Auflösung von 2 entspricht dabei einem Octree, jedoch kann er auch feiner aufgelöst werden.
Das Cascaded-Linespace-Verfahren nach Keul *et al.* aus [KLM] bietet einen Ansatz für eine effiziente Traversierung durch schnelles Überspringen leerer Bereiche der Szene (engl. empty-space-skipping). Es baut auf dem N-tree als Datenstruktur auf und berechnet zusätzlich für jeden Knoten innerhalb des N-tree einen Linespace. Dabei handelt es sich um vorberechnete Schäfte, die die Information auf Objekte innerhalb des Schafts speichern. Ein Schaft ist zwischen zwei Flächen auf der Außenseite des zugehörigen Knoten definiert (siehe Abbildung 4). Die Auflösung des Linespace entspricht der Auflösung des N-tree. Dadurch entspricht jedes Flächenelement auf der Außenseite des Knotens der Größe eines Unterknoten.

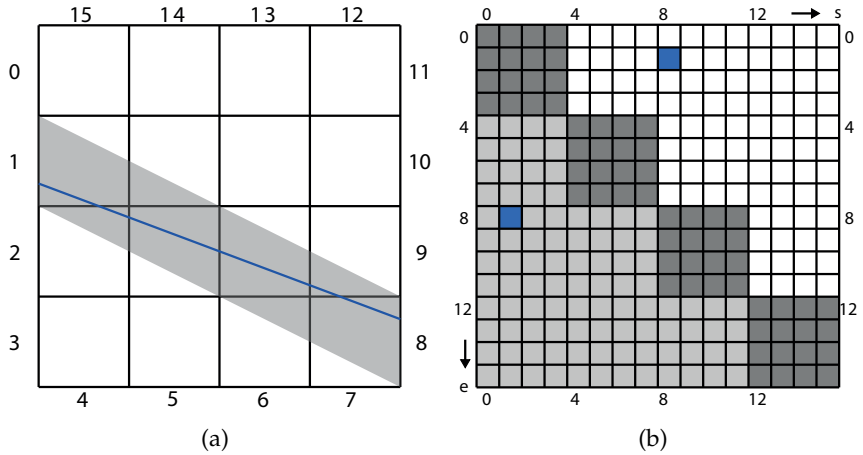
Der Linespace wird, wie in Abbildung 4(b) und Abbildung 5 visualisiert, als Bitmaske gespeichert. Ein Linespace-Eintrag wird durch einen boolean repräsentiert. Zur Berechnung dieses Wertes werden alle Knoten innerhalb des Schaftes untersucht. Enthält einer dieser Knoten Geometrie, wird der Eintrag im Linespace gesetzt. Dies geschieht mittels einer Bitmaske, die für jeden gefüllten Unterknoten erstellt wird (siehe Abbildung5(b)). Die Binärmasken der einzelnen Unterknoten werden durch die binäre OR-Operation zusammengefügt und erzeugen so einen Linespace.
Die in Abbildung 4(b) dunkelgrau markierten Einträge haben ihre Start- und Endfläche auf derselben Seite des Knoten und haben somit keinen Schaft. Die Bitmaske weist zudem eine Symmetrie der Einträge auf, da für

eine Startfläche s und eine Endfläche e gilt:

$$entry(s, e) = entry(e, s)$$

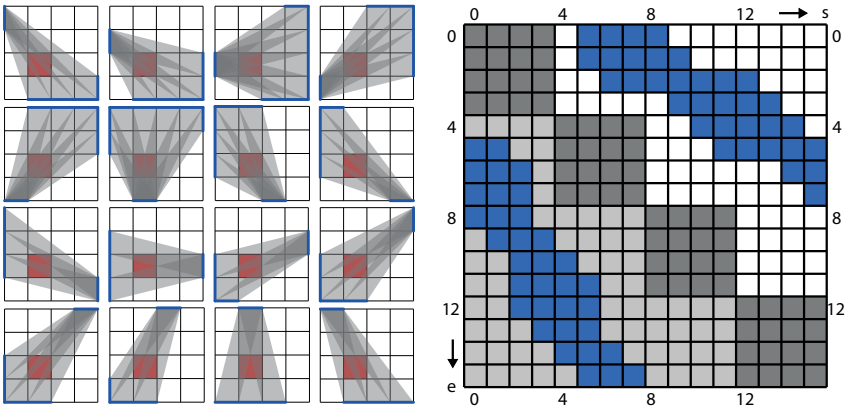
Dadurch ist es möglich den Speicherverbrauch zu verringern, indem diese Werte nicht gespeichert werden.

Die Initialisierung und Berechnung der Linespaces geschieht einmalig für alle Knoten innerhalb des N-tree, die Unterknoten besitzen. Der Vorteil des Linespace zeigt sich in einer schnelleren Traversierung des N-tree, indem leere Bereiche übersprungen werden können.
Wie in Abbildung 6(a) gezeigt, muss bei klassischem Raytracing über den N-tree, jeder Knoten entlang des Strahles verfolgt werden. Besitzt ein Knoten Kinder, müssen diese untersucht werden, bevor der nächste Knoten untersucht werden kann.
Durch Verwendung des Linespace in Abbildung 6(b), wird dieser Schritt verkürzt indem zunächst der Linespace-Eintrag für den Eintritts- und Austrittspunkt in den Knoten überprüft wird. Ist dieser Schaft leer, brauchen die Unterknoten nicht überprüft werden und es kann direkt der nächste Knoten überprüft werden. Sollte der Schaft Geometrie treffen, müssen die Unterknoten des Knotens traversiert werden.



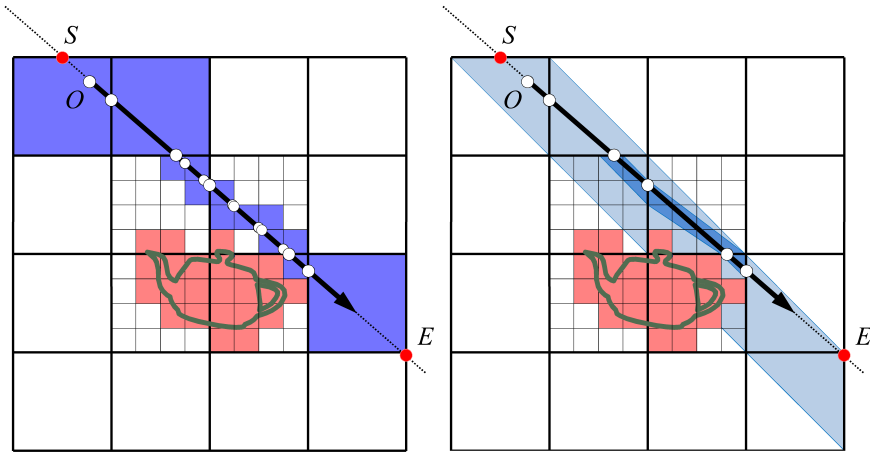
(a) (b)

Abbildung 4: Abbildung eines Strahles auf einen Schaft im Linespace der Auflösung 4x4 nach [KLM]. Die s-Achse ist die Achse der Startfläche, e die Achse der Endflächen.
(a) Ein Strahl kann durch seinen Start- und Endpunkt als Linespace-Schaft repräsentiert werden. (b) blau: Der zugehörige Eintrag in der Linespace-Bitmaske.



(a) (b)

Abbildung 5: Darstellung eines Unterknoten mit Geometrie innerhalb des Linespace nach [KLM]. Zu erkennen ist eine Symmetrie der Einträge. Dunkelgrau werden Flächen dargestellt, die auf derselben Seite des Knoten liegen und damit keinen Schaft haben. (a) rot: Ein Unterknoten mit Geometrie. grau: alle Schäfte, die den roten Unterknoten schneiden. (b) blau: Die zugehörigen Linespace-Einträge.



(a) (b)

Abbildung 6: Unterschied beim Raytracing aus [KLM]. weiß: Traversierungspunkte. rot: Zellen mit Geometrie. (a) Verfolgung eines Strahles mit dem DDA-Algorithmus. blau: zu untersuchende Knoten. (b) Beschleunigung durch den Linespace. hellblau: Schaft des Linespace. Die Traversierung wird durch Überprüfung der Linespace-Einträge verkürzt.

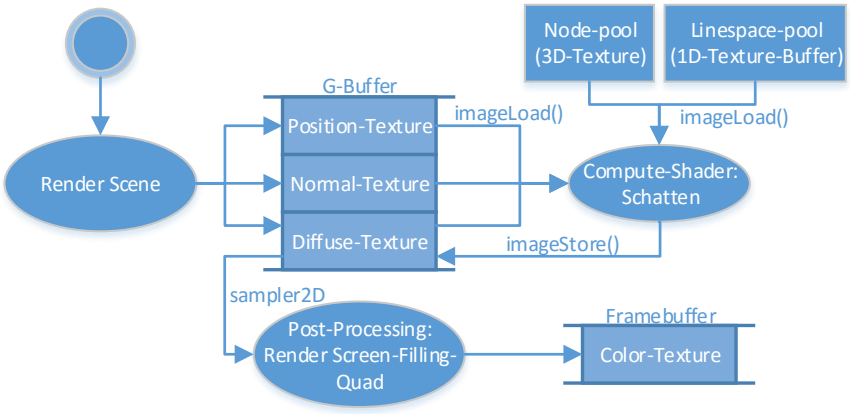


Abbildung 7: Rendering-Pipeline: Die Szene wird mit OpenGL gerendert und Position, Normale und diffuse Farbe in den G-Buffer geschrieben. Dieser dient als Eingabe in den Compute-Shader, der den Schattenwert für jeden Pixel berechnet und ihn mit dem diffusen Farbwert verrechnet. Dieser wird in die diffuse Textur zurückgeschrieben und anschließend ein Screen-Filling-Quad gerendert. Die diffuse Textur wird eingelesen und jeder Pixel in den Framebuffer geschrieben, der anschließend angezeigt wird.

4 Implementierung

Die Implementation besteht aus zwei Teilen. Der erste Teil beschäftigt sich mit der Render-pipeline auf der Grafikkarte, sowie dem Transferieren der Datenstrukturen in den Videospeicher. Im zweiten Teil geht es um das Traversieren dieser Datenstrukturen und Berechnung der Schattenwerte für jedes Pixel.

4.1 GPU-Rendering

Die Implementation folgt dem Beispiel von [Rob09]. Dieser stellt eine hybride Verwendung von GPU-Rendering und Raytracing vor. Unter „hybrid“ wird hierbei eine Kombination aus GPU-Rendering zur Berechnung der Primärstrahlen und Raytracing zur Berechnung sekundärer Effekte, wie Spiegelungen, bezeichnet.
Dieses Prinzip wird in dieser Implementation angewendet. Zur Vermeidung des Raytracing von Primärstrahlen, wird Hardware-Rasterisierung verwendet. Auf den daraus resultierenden Pixeln werden Sekundärstrahlen durch Raytracing verfolgt und das Ergebnis durch einen weiteren Rendering-Pass ausgegeben.

Die Rendering-Pipeline dieser Implementierung in Abbildung 7 besteht aus diesen drei Phasen. In der ersten Phase wird die Szene in den G-Buffer gerendert. Hierzu wird ein Framebuffer Object (kurz: FBO) gebunden und Position, Normale und diffuse Farbe jeweils in eine Textur geschrieben. Diese Texturen dienen in der zweiten Phase als Eingabe in den Compute-Shader. Dieser berechnet den Schattenwert der Szene, multipliziert diesen mit der diffusen Farbe und schreibt ihn anschließend in die Farbtextur zurück. Diese Farbtextur wird für die dritte Phase als $sampler2D$ gebunden. In der dritten Phase wird ein Screen-Filling-Quad gerendert, wodurch der Fragment-Shader für jeden Pixel aufgerufen wird und die Farbe der Farbtextur ausgibt.
Die Aufteilung in drei Phasen dient der Performance Optimierung. Eine alternative Herangehensweise würde die Szene rendern und den Schattenwert für jedes Fragment berechnen. Ein möglicher Fall wäre eine Render-Reihenfolge, in der Geometrie, die aus Sicht der Kamera hinter einem Objekt liegt, zuerst gerendert wird und dessen berechnete Pixel anschließend durch den Tiefentest nicht verwendet werden. Dadurch werden viele Schattenwerte berechnet, die im finalen Bild nicht abgebildet werden.
Zur Lösung dieses Problems wird die Szene in den G-Buffer gerendert, wodurch bei einem Post-Processing-Pass jedes Pixel einmalig, auf Basis der Daten des G-Buffer, aufgerufen wird. Der Schattenwert kann somit einmalig pro Pixel berechnet werden. Um die Vorteile moderner Hardware besser auszunutzen, wird die Berechnung der Schatten im Compute-Shader ausgeführt. Dieser bietet mehr Möglichkeiten der Parallelisierung und Optimierung von Algorithmen, im Gegensatz zum Fragment-Shader. Beispielsweise können Berechnungen eines Pixels in einer Gruppe (engl. „workgroup“) verteilt werden. Eine Gruppe besteht aus mehreren Threads, die auf den gleichen Daten operieren. Die Synchronisation der Threads geschieht mittels geteiltem Speicher (engl. „shared memory“). Der Fragment-Shader bietet diese Möglichkeit nicht und operiert immer auf einem Pixel.

4.2 Datenstrukturen

Für die Traversierung des N-tree auf der GPU, muss dieser als Datenstruktur in den Grafikspeicher geladen werden. Die N-tree Knoten und Linespaces der Knoten werden jeweils in einem Datenpool gespeichert. Die Daten werden hierbei hintereinander in einen Speicherblock geschrieben, wodurch ein Buffer pro Pool benötigt wird. Zudem wird auf eine kompakte Speicherung geachtet, da der dedizierte Videospeicher der Grafikkarte geringer ist, als der Hauptspeichers (RAM – engl. „random access memory“).

Der Linespace eines Knoten wird in der C++ Implementation in einem eindimensionalen Array aus boolean-Werten gespeichert. Auf der GPU wer-

den boolean Werte durch 32 Bit gespeichert. Um diesen Speicherverbrauch zu verringern, speichert die GPU-Datenstruktur dieses Array in einer kompakteren Form. Hierzu werden 32 boolean-Werte in einen 32 Bit unsigned-Integer gespeichert. Jedes Bit entspricht somit einem eigenen Wert des Linespace.
Dieser unsigned-Integer-Wert kann im Shader aus der Textur ausgelesen werden. Der Wert des Linespace-Eintrages wird um die Bit-Position des gesuchten Wertes durch eine binäre links-Shift-Operation bitweise nach links verschoben und anschließend durch eine binäre-AND-Operation mit dem ersten Bit ausgelesen.

Zur Speicherung der Daten wird eine dreidimensionale Textur verwendet. Eine dreidimensionale Textur bietet gegenüber einer zweidimensionalen Textur den Vorteil einer größeren möglichen Auflösung. Ein Linespace mit einer Auflösung von 10x10x10 besitzt 360000 Einträge. Somit ergeben sich 11250 komprimierte 32-Bit Werte pro Linespace. Eine dreidimensionale Textur ist damit besser skalierbar mit einer steigenden Anzahl an Linespaces.
Diese Textur besteht aus 32 Bit rot-Werten. Die maximale Größe der Textur pro Achse sind 2048 Pixel. Die komprimierten Werte eines Linespace werden in die x-Achse der Textur geschrieben. Sollte die Anzahl der Werte größer als der maximale Index sein, wird der y-Wert um eins inkrementiert und dort fortgesetzt. Um die Linespaces aller Nodes zu speichern, werden die Linespace-Blöcke auf der y-Achse fortgesetzt und bei Erreichen des maximalen Index auf der z-Achse fortgesetzt.

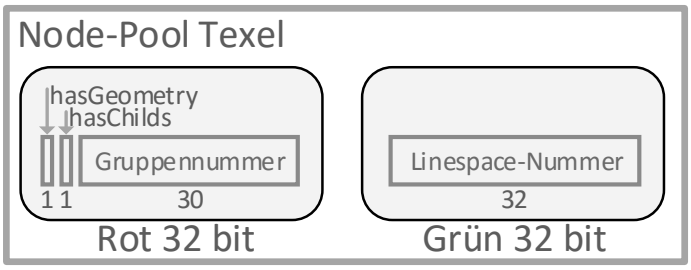


Abbildung 8: Aufbau eines Node-Texel. Der rot-Wert speichert in seinen ersten beiden Bits die Informationen auf Geometrie innerhalb des Knoten, sowie auf weitere Unterknoten. Die restlichen 30 Bit werden zur Speicherung der Gruppennummer seiner Unterknoten benutzt. Der grün-Wert speichert die Nummer des zugehörigen Linespace, sofern dieser existiert.

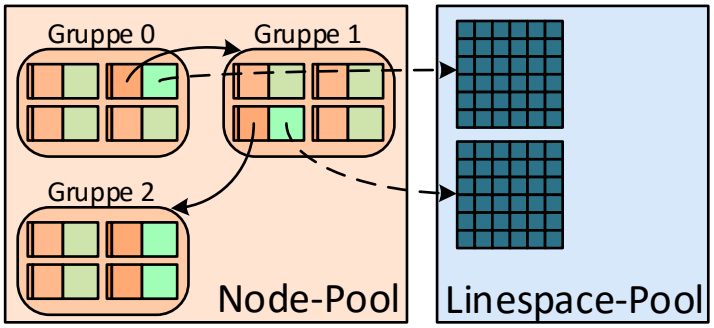


Abbildung 9: Aufbau der Datenpools: Der Node-Pool besteht aus Node-Gruppen. Diese speichern alle Unterknoten eines Knoten (engl. Node). Gruppe 0 speichert die Unterknoten des Wurzelknotens. Der rot-Wert eines Knotens speichert die Nummer der Gruppe seiner Unterknoten. In seinem grün-Wert speichert er die Nummer des Linespace im Linespace-Pool.

Die Speicherung des N-tree orientiert sich an Crassins Implementation eines Octree in [CNE09]. Ein Node wird in Abbildung 8 durch zwei unsigned-Integer-Werte repräsentiert. Der erste Wert verweist auf die Gruppe der Unterknoten des Knotens, indem er die Nummer dieser Gruppe speichert. Die gruppierte Speicherung ermöglicht eine einfache Referenzierung auf die Kinder eines Knotens (siehe Abbildung 9).
Der zweite Wert verweist auf den, dem Knoten zugehörigen, Linespace oder enthält null, wenn dieser nicht existiert. Die Zeiger auf Unterknotengruppen beziehungsweise Linespaces beziehen sich auf die lokale Position des Speicherblockes innerhalb des Pools und werden nicht als globale Zeiger gespeichert.
Der rot-Wert eines Node speichert in seinen ersten beiden Bits zusätzliche Informationen. Das erste Bit wird gesetzt, wenn der Knoten Geometrie beinhaltet. Das zweite Bit repräsentiert die Information auf Unterknoten. Die Traversierung unterscheidet zwischen den Fällen, dass ein Knoten Kinder besitzt oder dass er Dreiecke besitzt und somit keine Kinder. Wenn ein Node im N-tree Dreiecke speichert, wird das Bit für Geometrie gesetzt aber nicht das Bit für Unterknoten.

Der Node-Pool wird in einem 1D-Texture-Buffer gespeichert. Der Texture-Buffer speichert die Daten, wie in einer Textur, in den roten und grünen Farbkanälen, kann jedoch nicht als Textur im Shader verwendet werden. Die größere mögliche Texturauflösung bietet einen Vorteil gegenüber einer 1D-Textur. Es können jedoch keine Mip-Maps generiert werden und

ein Zugriff durch den OpenGL-Befehl *texture(...);* ist nicht möglich. Der Zugriff geschieht stattdessen mit dem Befehl *texelFetch(...);* oder ab OpenGL 4.2 mit *imageLoad(...);*. Diese Befehle geben den Wert des angesprochenen Pixels aus und interpolieren diesen nicht.

4.3 Traversierung

Die Traversierung des N-tree baut das Coherent-Grid-Traversal (kurz: CGT) Verfahren von Wald *et al.* in [WIK+06] aus. Das hier vorgestellte Verfahren orientiert sich dabei an der Implementation von Knoll *et al.*, vorgestellt in [KWH09], und erweitert dieses auf die Benutzung eines N-tree, sowie des Linespace. Knoll *et al.* benutzen als Datenstruktur einen Octree, dessen Unterschied zum N-tree in seiner Auflösung besteht.

Bei dem CGT-Verfahren handelt es sich um eine Methode zur Traversierung eines Uniform-Grid. Die Besonderheit des Verfahrens liegt in der Verfolgung eines Paketes von Strahlen, im Gegensatz zum Verfolgen eines einzelnen Strahles beim Raytracing. Wie in Abbildung 10(a) dargestellt, besteht ein Paket aus vielen Strahlen, die einen gemeinsamen Startpunkt und ähnliche Richtungen besitzen. Diese Strahlen schneiden in den meisten Fällen gleiche Zellen, wodurch die Datenstruktur einmalig traversiert werden muss. Dies vermeidet die Traversierung der Datenstruktur für jeden Strahl und erreicht zudem bessere Cache-Effizienz, da die benötigten Daten einer Zelle einmalig geladen werden und danach auf alle Strahlen innerhalb des Paketes angewendet werden. Im Gegenzug müssen mehr Zellen untersucht werden, desto größer die Differenz der unterschiedlichen Richtungen der Strahlen ist. Strahlen werden unter Umständen mit Zellen geschnitten, die Sie beim klassischen Raytracing nicht schneiden würden.

Bei klassischem Raytracing durch den von Amanatides und Woo in [AW87] vorgestellten 3DDA-Algorithmus, wird von Zelle zu Zelle traversiert. Die nächste zu traversierende Zelle wird dabei aus dem Austrittspunkt der aktuellen Zelle berechnet. Um, wie in Abbildung 10(a), ein Paket an Strahlen zu verfolgen, ist dieser Ansatz nicht mehr ausreichend. Statt einzelne Zellen zu untersuchen, wird eine Scheibe aus Zellen untersucht. Diese Scheibe wird in Abbildung 10(b) blau dargestellt und entspricht aller Zellen eines bestimmten Index der k-Achse.
Zur Verallgemeinerung wird die Traversierungsachse k-Achse genannt und die dazu orthogonalen Achsen u und v. Die Traversierungsachse wird aus der dominanten Achse des Lichtvektors bestimmt. Der Lichtvektor beschreibt die normalisierte Richtung des Startpunktes zum Mittelpunkt der Lichtquelle. Dadurch wird die Anzahl der zu untersuchenden Zellen minimiert.

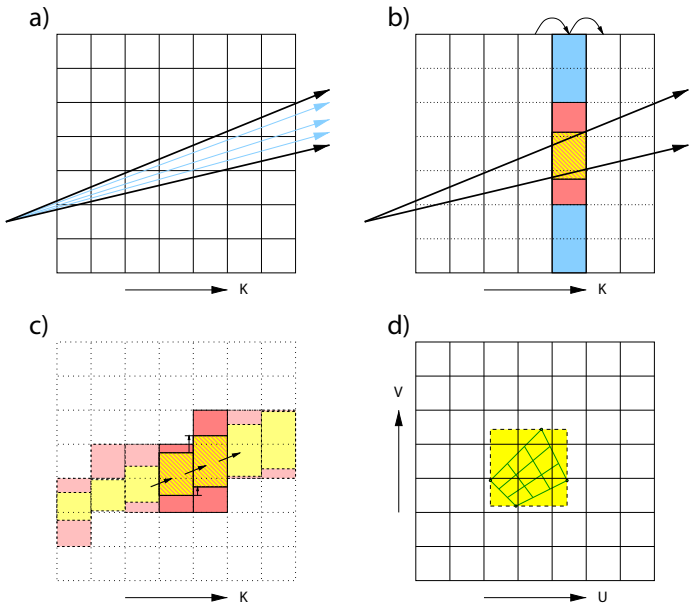


Abbildung 10: Traversierung eines Uniform-Grid nach [WIK+06]
a) Ein Paket aus Strahlen wird begrenzt durch die äußeren Strahlen (schwarz)
b) blau: Die aktuelle Schicht - rot: Die zu untersuchenden Zellen - gelb: durch u_min/u_max und v_min/v_max beschränkter Bereich
c) Die Veränderung von u_min/u_max und v_min/v_max ist konstant
d) Blick entlang der Traversierungsachse k. Grün: Paket aus Strahlen

Ein Paket tastet immer eine viereckige Lichtquelle ab und besitzt somit immer vier Eckstrahlen zu den jeweiligen Eckpunkten der Lichtquelle. Zur Optimierung werden nur Zellen der Scheibe untersucht, die innerhalb des Schaftes des Pakets liegen. Dieser Schaft wird durch die vier Eckstrahlen (schwarz) des Pakets definiert. Die dadurch betroffenen Zellen, in Abbildung 10 rot dargestellt, werden durch die Ein- und Austrittspunkte des Schaft definiert und werden gelb eingezeichnet. Dabei werden jeweils nur die minimalen und maximalen u und v-Koordinaten der Eckpositionen genommen. Daraus wird sichergestellt, dass das gesamte Paket innerhalb des zu untersuchenden Bereiches liegt. Das Paket wird in Abbildung 10(d) grün eingezeichnet.
Zur Traversierung über das Gitter, wird der k-Index um eins erhöht. Die Veränderung der minimalen und maximalen u-/v-Koordinaten ist konstant, wie in Abbildung 10(c) verdeutlicht. Dieser konstante Wert wird weiter als $delta\_uv$ bezeichnet und auf die Eckpositionen des Schaftes addiert.

Da die Lichtquelle mit vielen Strahlen abgetastet werden soll, werden diese Strahlen zur Laufzeit interpoliert. Dabei muss jeweils die Richtung des interpolierten Strahles zwischen den vier Richtungen der Eckstrahlen bilinear interpoliert werden. Der Ursprung der Strahlen bleibt bei allen Strahlen identisch in der Startposition des Schaftes.

4.3.1 Implementation

Die Implementation des Algorithmus erfolgt sowohl in C++, als auch in GLSL.
Ein Ansatz für die Erweiterung des CGT-Algorithmus auf hierarchische Datenstrukturen, ist die Verwendung von rekursiven Aufrufen. Diese würden Unterknoten eines zu traversierenden Knoten untersuchen. Da GLSL keine Rekursion unterstützt, muss eine Alternative verwendet werden. Diese besteht in der Verwendung eines „Stack“. Bei rekursiven Aufrufen auf der CPU, werden später benötigte Variablen und Rücksprungadressen in einem Stack gespeichert. Dieser Stack wird in dieser Implementation durch eine struct definiert.

Der Stack in Listing 1 ist über alle Tiefen des N-tree definiert. Tiefe null ist dabei die Tiefe der Kinder des Wurzelknotens des N-tree. Der Elternknoten der Tiefe null ist demnach der Wurzelknoten. Die Variable $parent\_node$ speichert somit immer den Elternknoten der aktuell untersuchten Knoten ab.

```
struct Stack
{
  uvec2 parent_node;
  uvec2 nodelist[100];
  ivec2 offset[100];
  int nodelist_index;
  bool traverse_childs;
  int current_slice;
  int last_slice;
  float u_min, u_max, v_min, v_max;
} stack[ NUM_MAX_DEPTH ];
```

Listing 1: Structure-Definition von „Stack“ in GLSL

Die $nodelist[100]$ speichert Knoten mit Geometrie und Kindern, die weiter traversiert werden müssen. Die Länge des Arrays ist über die maximale Anzahl der Zellen einer Schicht für die Auflösung des N-tree definiert. In dieser Implementation wird ein N-tree mit einer Auflösung von 10 auf jeder Achse verwendet, wodurch die maximale Anzahl an Zellen einer Schicht durch das Quadrat der Auflösung berechnet wird. Das Array $offset[100]$ ist von der Länge wie das Array $nodelist$ definiert. Als Offset werden die u- und v-Koordinate des Knotens gespeichert, da diese für eine Transformation in das Koordinatensystem des Knoten erforderlich sind. Der $nodelist\_index$ gibt die Anzahl der zu untersuchenden Knoten an. Dadurch muss die $nodelist$ nicht zurückgesetzt werden, wenn Sie abgearbeitet wurde. Die Einträge der Liste werden stattdessen überschrieben.
Der boolean-Wert $traverse\_childs$ wird zum Wechsel des Methodenzustands benutzt. Die Methode unterscheidet zwischen zwei Zuständen. Im Zustand „Schicht untersuchen“ werden Knoten mit Kindern in die $nodelist$ geschrieben. Der zweite Zustand dient der Abarbeitung der $nodelist$. Somit ist eine weitere Abarbeitung der $node\_list$, nach einem Rücksprung aus einer tieferen Ebene, gewährleistet. Die Variablen $current\_slice$ und $last\_slice$ speichern die aktuell untersuchte und letzte zu untersuchende Schicht der Tiefe.
Die float-Werte u_min, u_max, v_min und v_max dienen dem Speichern der vier Eckpunkte des Schafts, für den Eintritt in die untersuchte Schicht. Dabei werden, wie in Abbildung 10(c) gelb dargestellt, die Minima und Maxima der Eintritts- und Austrittspunkte gespeichert.

Der Schaft wird, wie der Stack, durch eine struct repräsentiert. Diese struct dient der Zusammenstellung wichtiger Variablen der Traversierung. Vektoren werden hierbei im Koordinatensystem des N-tree gespeichert.

```
struct Shaft {
  vec3 origin;
  vec3 destination;
  vec3 corner_dir[4];
  bool active_rays[ NUM_RAYS ];
  vec3 ray_start[ NUM_MAX_DEPTH ];
  vec3 ray_direction[ NUM_RAYS ];
} shaft;
```

Listing 2: Structure-Definition von „Shaft“ in GLSL

Die Vektoren $origin$ und $destination$ speichern den Start- und Endpunkt des Schafts in N-tree-Koordinaten. Der Endpunkt ist dabei der Mittelpunkt der Lichtquelle und dient der Berechnung des End-Index.
Das Vektorarray $corner\_dir[4]$ speichert die Richtungen der Eckstrahlen, in der Reihenfolge:

$$\{(u\_min, v\_min), (u\_min, v\_max), (u\_max, v\_min), (u\_max, v\_max)\} \tag{1}$$

Diese Reihenfolge wird zur Interpolation der Strahlen benötigt. Die Richtungen sind zudem auf den k-Wert von eins normiert. Aus diesen Richtungen wird $delta\_uv$ und die Richtungen der interpolierten Strahlen berechnet. Der Vermerk auf einen Treffer eines Strahles wird in dem boolean-Array $active\_rays[\,]$ gesetzt. Diese Einträge werden weiter dazu benutzt, inaktive Strahlen in der Traversierung zu überspringen.
Um die Strahlen mit den Knoten zu schneiden und die Schnittpunkte der Strahlen mit dem Knoten berechnen zu können, muss der Startpunkt und die Richtung jedes Strahles im Koordinatensystem des Knotens vorliegen. Dazu wird der Startpunkt in das Koordinatensystem des untersuchten Unterknotens transferiert und in das Array $ray\_start[\,]$ für die nächste Tiefe geschrieben:

$$ray\_start[depth + 1] = n \times (ray\_start[depth] - uvk\_offset);$$

Es wird der dreidimensionale Index des Unterknoten innerhalb des Knoten als uvk_offset bezeichnet. Dieser ist ganzzahlig und befindet sich zwischen 0 und der Auflösung des N-tree n. Der resultierende Vektor wird mit n multipliziert, um den Bereich von m bis $m + 1$ einer der drei Achsen, in den Bereich 0 bis n zu transformieren. Die Transformation des Startpunkts geschieht einmalig für einen Knoten, da dieser für jeden Strahl identisch ist. Die Richtungen der Strahlen werden hingegen nicht transformiert, da diese für jeden Unterknoten identisch sind. Stattdessen werden Sie einmalig aus den Eckrichtungen des Schaftes interpoliert und in dem Array $ray\_direction[\,]$ gespeichert.

Algorithm 1 Pseudocode: Traversierung – Vorbereitung
Es werden für die Traversierung benötigte Werte berechnet und die structs $shaft$ und $stack$ initialisiert.

Input:
pos — Die Weltkoordinate der Startposition
light_dir — Die normalisierte Richtung von pos zum Mittelpunkt der Lichtquelle
Output:
shadow — Der Schattenwert zwischen $(0, 1)$: 0 Schatten – 1 Licht

1: **procedure** SHADOW
2: bestimme Traversierungsrichtung {...}
3:
4: //Initialisiere *shaft*: Transformiere in das N-tree-Koordinatensystem
5: $shaft.origin \leftarrow (pos -$ ntree.minBound$) \times$ ntree.invNodeSize
6: $shaft.dest \leftarrow (light\_dir -$ ntree.minBound$) \times$ ntree.invNodeSize
7: $active\_rays \leftarrow$ NUM_RAYS
8: $shaft.corner\_dir[\,] \leftarrow$ vier Eckpunkte der Lichtquelle im N-tree-KS
9: $shaft.ray\_start[0] \leftarrow shaft.origin$
10: **for** NUM_RAYS **do**
11: $shaft.ray\_end \leftarrow$ interpolierte Richtung zwischen Eckpositionen
12: **end for**
13: $delta\_uv \leftarrow$ minimale/maximale Inkrementierung der u-/v-Koordinate
14: $shaft.corner\_pos[\,] \leftarrow$ Position unter Verrechnung von $delta\_uv$
15:
16: initialisiere $stack[0]$ unter Verwendung von $shaft$ {...}
17: $depth \leftarrow 0$
18:

Zur Vorbereitung der Traversierung in Algorithmus 1 wird zunächst die dominante Achse k bestimmt. Diese wird festgelegt durch die größte Komponente des normierten Lichtvektors. Die Achsen u und v werden für die anderen beiden Achsen gesetzt. Zusätzlich wird die Traversierungsrichtung als $+1$ oder -1 in dem Integer dir gespeichert. Dieser Wert ist negativ, wenn die Traversierungsrichtung in Richtung einer negativen Achse ist. Benötigt wird der Wert in der Schleife über alle Scheiben zur Gewährleistung einer Front-to-back-Traversierung, für frühe Terminierung durch getroffene Geometrie.
Die Initialisierung der struct Shaft, dient der Initialisierung der Schattenstrahlen. Hierzu werden Start- und Endpunkt des Strahles zum Mittelpunkt der Lichtquelle in das Koordinatensystem des Wurzelknotens transformiert. Diese Transformation erfolgt nach [KLM] durch folgende Gleichung:

$$shaft.origin = (position - ntree.minBound) \times ntree.invNodeSize;$$

Der minimale Index des N-tree-Knotens in Weltkoordinaten wird als $minBound$ verrechnet. Durch die Subtraktion von der Position wird der Punkt in den Ursprung des N-tree verschoben. Die Multiplikation mit der inversen Größe eines Unterknoten des Wurzelknotens transformiert die Koordinaten des Punktes in das Einheitensystem des Knotens. Wenn der Punkt innerhalb des Knotens liegt, liegt er nach der Multiplikation im Einheitenbereich 0 bis $maxIndex$ oder ist größer, beziehungsweise kleiner, wenn der Punkt außerhalb liegt. Diese Transformation wird nur für die initiale Transformation der Punkte in das N-tree-Koordinatensystem verwendet.

Die Richtungen der Strahlen zum Abtasten der Lichtquelle werden bilinear aus den vier Eckstrahlen interpoliert. Da die Eckstrahlen bei der Speicherung bereits in der in Gleichung 1 beschriebenen Reihenfolge vorliegen, werden zunächst die v-Koordinaten zwischen den Strahlen 0 und 1, sowie 2 und 3 linear interpoliert. Anschließend wird die u-Koordinate zwischen den interpolierten v-Koordinaten interpoliert. Die Anzahl der interpolierten Strahlen auf den Achsen u und v muss zuvor manuell in die Konstanten $SAMPLES\_U$ und $SAMPLES\_V$ eingetragen werden. Hier würde sich eine Winkel-abhängige Implementierung anbieten.

In der Traversierung in Algorithmus 1(Fortsetzung) wird zunächst die aktuelle Schicht der aktuellen Tiefe untersucht und Knoten mit Geometrie und Kindern in die $nodelist$ geschrieben. Im Anschluss wird für jeden Eintrag der $nodelist$ der Linespace des Knotens überprüft. Trifft einer der Strahlen Geometrie im Linespace, wird über die Unterknoten traversiert, indem die Variablen der aktuellen Tiefe im Stack gespeichert werden und der Stack der nächsten Tiefe vorbereitet wird. Die Überprüfung des Linespace wird in Abschnitt 4.3.2 erläutert.

Algorithm 1 Pseudocode: Traversierung (Fortsetzung)
Die Traversierung vermeidet rekursive Aufrufe. Die äußere while-Schleife ist für die Traversierung aller Tiefen zuständig. Darin wird eine for-Schleife ausgeführt, die alle Schichten eines Knoten traversiert. Innerhalb der for-Schleife werden entweder alle getroffenen Knoten der Schicht, die Kinder besitzen, in eine Liste geschrieben oder diese Liste bearbeitet.

```
19:     //Traversierung
20:     while depth >= 0 do
21:         for all slice von current- bis last_slice do
22:             if stack[depth].traverse_childs ist false then
23:                 for all Nodes innerhalb u/v_min/max do
24:                     if node besitzt Geometrie und hat Unterknoten then
25:                         stack[depth].nodelist ← push_back node
26:                     end if
27:                 end for
28:             end if
29:             for all node in nodelist do
30:                 shaft.ray_start[depth + 1] ← transf. in Subnode-KS
31:                 //Linespace Überprüfung
32:                 if ISINTERSECTION(node, active_rays) then
33:                     continue                          ▷ Empty-Space-Skipping
34:                 end if
35:                 if active_rays gleich 0 then
36:                     return 0.f
37:                 end if
38:                 //Nächste Traversierungstiefe vorbereiten
39:                 stack[depth].traverse_childs ← true
40:                 stack[depth].current_slice ← slice
41:                 setze Variablen für stack[depth + 1]
42:                 //+2 da am Ende der while-Schleife dekrementiert wird
43:                 depth ← depth + 2
44:                 next_depth ← true
45:                 break
46:             end for
47:             if next_depth ist true then
48:                 break
49:             end if
50:             stack[depth].traverse_childs ← false
51:             stack[depth].u/v_min/max ← addiere delta_uv
52:         end for
53:         depth ← depth − 1
54:     end while
55:     return active_rays ÷ NUM_RAYS
56: end procedure
```

Die Methode endet, wenn alle Strahlen Geometrie getroffen haben, oder die Traversierung der ersten Tiefe endet. Die Rückgabe der Methode ist ein Wert zwischen null und eins der das Verhältnis der aktiven Strahlen zu allen gesendeten Strahlen enthält. Dieser wird berechnet durch die Division der aktiven Strahlen $active\_rays$ mit der Anzahl aller Strahlen NUM_RAYS.

4.3.2 Linespace-Überprüfung

Die Überprüfung des Linespace wird durch die Funktion *isIntersection(node, active_rays)* in Algorithmus 2 ausgeführt. Dabei werden inaktive Strahlen übersprungen und durch die Methode *isNodeIntersection(...)* überprüft, ob der Strahl den Knoten schneidet. Diese Methode berechnet zudem die Start- und Endseite, sowie den Start- und Endvektor.
Der Start- und Endvektor geben den Ein- und Austrittspunkt des Strahles mit dem Knoten an. In diesen Punkten ist ein Wert des Vektors entweder null oder N, wobei N für die Auflösung des N-tree steht. Die Start- bzw. Endseite gibt die Ein- bzw. Austrittsseite des Strahles mit dem Knoten an und liegt im Wertebereich $\{-3, -2, -1, 1, 2, 3\}$. Die x-Achse wird durch die 1 repräsentiert, die y-Achse durch die 2 und die z-Achse durch die 3. Diese Zahl ist positiv für einen Wert gleich null und negativ für einen Wert gleich N.
Diese Werte dienen als Eingabe in die Methode *getEntry(...)*, die den Linespace-Eintrag der Eingabewerte ausliest und $true$ für getroffene Geometrie ausgibt. Weiter wird überprüft, ob die untersuchte Tiefe der letzten Tiefe mit Linespaces entspricht, um in diesem Fall einen Strahl, bei getroffener Geometrie, auf inaktiv zu setzen. Durch dieses Vorgehen wird die tiefste Ebene des N-tree nicht untersucht und keine Schnitttests mit Dreiecken durchgeführt. Stattdessen dient der Linespace als Indikator auf getroffene Geometrie.
Für den Fall einer positiven Linespace-Überprüfung in einer anderen Tiefe, gibt die Funktion $true$ zurück ohne die weiteren Strahlen zu testen. Es wird in diesem Fall von der Funktion lediglich die Information auf einen positiven Schnitt, eines Strahles mit der Geometrie, benötigt. Somit können unnötige Überprüfungen übersprungen werden.

Die Funktion *isNodeIntersection(...)* aus Algorithmus 3 setzt den Algorithmus von Kay und Kajiya in [KK86] um. Dieser Algorithmus dient dem Schnitttest eines Strahles mit einer Bounding-Box und wird in dieser Implementation als Schnitttest eines Strahles mit einem Knoten benutzt. Der Startpunkt des Strahles wurde bereits in Algorithmus 1 in das Koordinatensystem des Knotens transformiert. Die Richtung kann direkt übernommen werden, da ein Koordinatensystem eines Unterknoten abhängig vom Koordinatensystem seines Elternknoten und nicht rotiert ist.

Algorithm 2 Pseudocode: Linespace Überprüfung
Es wird er Linespace-Eintrag für jeden Strahl untersucht. Dazu wird der Strahl mit dem Knoten geschnitten und aus den Schnittpunkten die Start-und Endfläche des Linespace-Schaft berechnet.

Input:
node Zu untersuchender Node
active_rays Anzahl der aktiven Strahlen
Output:
found true wenn Geometrie von einem Strahl getroffen wurde – sonst false

```
1:  function ISINTERSECTION(node, active_rays)
2:      found_intersection ← false
3:      for all Strahlen in shaft do
4:          if Strahl inaktiv then
5:              continue
6:          end if
7:          if ! ISNODEINTERSECTION(start/endFace, start/endVec) then
8:              continue                    ▷ Der Strahl trifft den Knoten nicht
9:          end if
10:         start−/endDim ← berechne aus start/endVec abhängig von start/endFace
11:         if GETENTRY(node, start/endFace/Dim) then
12:             found_intersection ← true
13:             if depth ist die letzte Tiefe mit Linespaces then
14:                 shaft.active_rays[] ← setze Strahl inaktiv
15:                 active_rays ← active_rays − 1
16:             else
17:                 return true          ▷ Es muss ein Strahl treffen, damit der
    Knoten weiter untersucht wird.
18:             end if
19:         end if
20:     end for
21:     return found_intersection
22: end function
```

Algorithm 3 Pseudocode: Überprüfe Node auf Schnitt
Der Strahl wird mit dem Knoten geschnitten und der Ein- und Austrittspunkt des Strahles mit dem Knoten bestimmt. Zusätzlich werden die Start- und Endfläche für die Linespace-Überprüfung berechnet.

Input:
node Zu untersuchender Node
shaft Struktur zum Speichern der Strahlinformationen
Output:
found true wenn der Strahl den Node schneidet – sonst false

1: **function** ISNODEINTERSECTION(node, shaft, start/endFace/Vec)
2: $startPoint \leftarrow shaft.ray\_start[depth]$ ▷ *origin* im aktuellen KS
3: $direction \leftarrow$ Richtung des Strahl aus $shaft.ray\_direction[]$
4: $min \leftarrow$ Index des Eintrittspunkts in den Node
5: $max \leftarrow$ Index des Austrittspunkts aus dem Node
6: **if** *min* größer *max* – Node wird nicht geschnitten **then**
7: **return** *false*
8: **end if**
9: $startVec \leftarrow startPoint + min \times direction$
10: $endVec \leftarrow endPoint + max \times direction$
11: $startFace \leftarrow$ berechne aus $startVec$
12: $endFace \leftarrow$ berechne aus $endVec$
13: **return** $true$
14: **end function**

Algorithm 4 Pseudocode: Auslesen eines Linespace-Eintrages
Der Linespace-Eintrag wird aus der Textur ausgelesen. Dazu wird die die Position innerhalb der 3D-Textur bestimmt und das gesuchte Bit bestimmt.

Input:

node	Zu untersuchender Node
s-/eFace	Index des Ein- und Austrittspunktes
s-/eDim	Index der Ein- und Austrittsfläche des Linespace-Schaftes

Output:

found	true wenn der Linespace-Eintrag positiv ist – sonst false

1: **function** GETENTRY(node, start-/endFace, start-/endDim)
2: $entry\_num \leftarrow$ Index des Eintrages innerhalb des Linespace-Array
3:
4: //Bestimmung der Position innerhalb der 3D-Textur
5: $entry\_num\_comp \leftarrow$ Komprimierter Index (*entry* dividiert durch 32)
6: $bit\_position \leftarrow$ Position des gesuchten Bits innerhalb des 32-Bit Eintrag
7: $x/y/z\_coord \leftarrow$ Position innerhalb der 3D-Textur abhängig von $entry\_num\_comp$ und der Nummer des Linespace von $node$
8:
9: //Auslesen des Wertes
10: $entry \leftarrow$ unsigned-Integer-Eintrag aus der 3D-Textur in $x/y/z\_coord$
11: $entry\_value \leftarrow$ Wert des Bits von $entry$ an der Stelle $bit\_position$
12: **if** $entry\_value$ ist größer null – Geometrie wird getroffen **then**
13: **return** true
14: **else**
15: **return** false
16: **end if**
17: **end function**

```
1  /* x-position in the linespace */
2  int entry_num_compressed = int(entry_num / 32.f);
3  int bit_position = int(mod(entry_num, 32.f));
4
5  /* y-position in the linespace */
6  int y_position = (int(ls_index) * ls_y_size)
7    + int(entry_num_compressed / tex_x_res);
8
9  /* coordinates in the linespace_buffer */
10 int y_coord = int(mod(y_position, tex_y_res));
11 int x_coord = int(mod(entry_num_compressed, tex_x_res));
12 int z_coord = int(y_position / tex_y_res);
```

Listing 3: Codeausschnitt zur Berechnung der Linespace-Position innerhalb der 3D-Textur in GLSL

Das Auslesen eines Linespace-Eintrages geschieht mit Hilfe der Parameter aus Unterabschnitt 4.2 und wird in Algorithmus 4 und Listing 3 dargestellt. Die Berechnung der dreidimensionalen Position erfordert hierbei die Auflösung der 3D-Textur auf der x-und y-Achse, sowie die Größe eines Linespace auf der y-Achse. Diese Konstanten werden durch die Variablen tex_x_res, tex_y_res und ls_y_size repräsentiert. Die x-Koordinate ergibt sich aus der komprimierten Position $entry\_num\_comp$ modulo der Auflösung der Textur auf der x-Achse.
Die y-Position berechnet sich aus der Nummer des Linespace multipliziert mit der Größe eines Linespace auf der y-Achse. Zudem muss die lokale y-Position des Eintrages innerhalb des Linespace addiert werden. Diese ergibt sich durch das abgerundete Ergebnis der Division von $entry\_num\_comp$ mit der Auflösung der Textur auf der x-Achse. Daraus berechnet sich die y-Koordinate des Eintrages aus der y-Position modulo der Auflösung der Textur auf der y-Achse. Abschließend wird die z-Koordinate durch das Ergebnis der abgerundeten Division der y-Position mit der y-Auflösung berechnet.
Der Eintrag wird aus der Textur durch die *imageLoad(...)*-Funktion ausgelesen. Um den Wert des Eintrages an der berechneten Bit-Position zu erhalten, wird der Wert um die Bit-Position binär nach links verschoben. Dies wird durch den binären Links-Shift-Operator durchgeführt. Das gesuchte Bit befindet sich nach der Operation an der ersten Stelle des Wertes und kann somit mit einer binären AND-Operation, mit der ersten Stelle des Wertes, ausgelesen werden. Der daraus resultierende Wert ist größer null, wenn das Bit gesetzt ist und ansonsten null.

5 Evaluation

Um das in dieser Arbeit vorgestellte Verfahren zu testen, wird der Stanford-Bunny mit dem Stanford-Dragon verglichen. Der N-tree und Linespace besitzen eine Auflösung von 10x10x10 mit einer maximalen Tiefe von 3. Die Berechnungszeiten der Schattenberechnung auf CPU und GPU werden ohne die Berechnung der Primärstrahlen gemessen. Unter den Primärstrahlen fällt in diesem Zusammenhang die Berechnung des Bildes ohne Schatten. Die Schattenberechnung wird auf dieses Bild angewendet.
Zur Reduzierung statistischer Ungenauigkeiten durch unterschiedliche Berechnungszeiten, wird eine Messung über ein Intervall von zehn Bildern durchgeführt und der Mittelwert aus diesen Daten berechnet.

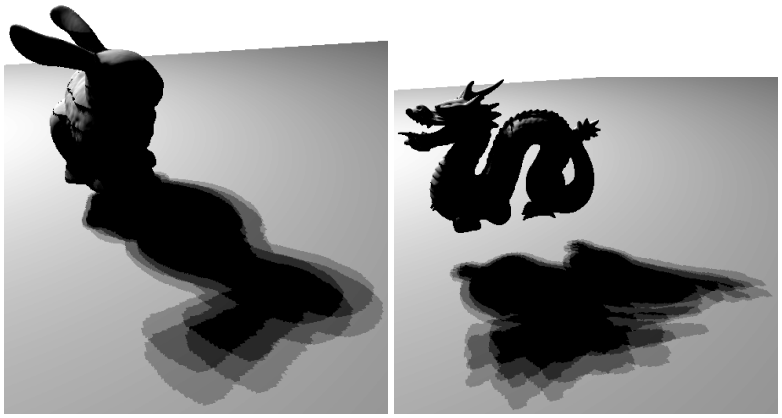
Primärstrahlen	
Variante	Zeit in Sekunden
GPU Bunny	0,000993448
CPU Bunny	0,178985
GPU Dragon	0,00189566
CPU Dragon	0,197976

Tabelle 1: Das initiale Bild für die Berechnung der Schatten (Primärstrahlen), wird auf der CPU durch Raytracing berechnet und auf der GPU durch GPU-Rendering. Die unterschiedlichen Berechnungszeiten werden in dieser Tabelle dargestellt.

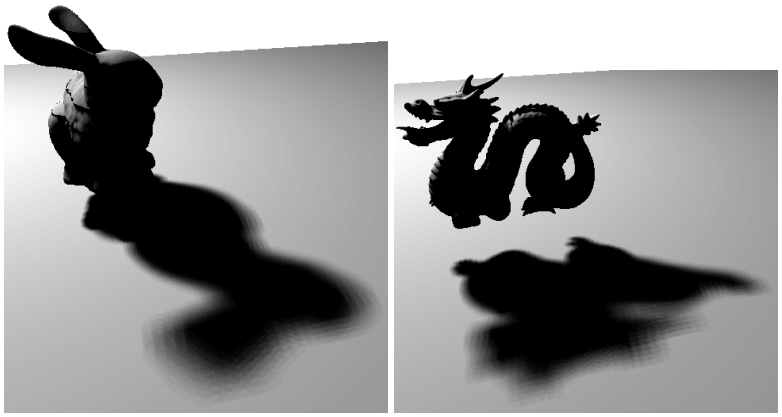
Das Testsystem besteht aus einem Intel Xeon E3-1230 v3 Prozessor mit 4 x 3,30 GHz, 8 Gigabyte RAM und einer NVIDIA GTX 760 Grafikkarte mit 2 Gigabyte dediziertem GDDR5 Speicher. Die Testbilder wurden in einer Auflösung von 512 x 512 erzeugt.

In den Abbildungen 11(a) und 11(b) zeigen sich kantige Ränder der Schatten. Diese Artefakte entstehen durch die Verwendung des Linespace und der damit verbundenen geringeren Genauigkeit. Diese Ungenauigkeit wird durch die Verwendung vieler Strahlen eingedämmt und tritt in den Abbildungen 11(c) und 11(d) nicht mehr auf. Stattdessen werden hier Artefakte aufgrund einer Unterabtastung der Lichtquelle deutlich. Die Lichtquelle wird in diesen Bildern mit 25 Strahlen abgetastet, wodurch maximal 25 unterschiedliche Grauwerte für den Schatten entstehen können. Durch die Größe der Penumbra werden diese Abstufungen deutlich. Durch die Verwendung von 100 Schattenstrahlen in den Abbildungen 11(e) und 11(f) werden diese Abstufungen geglättet und ein weicher Schatten erzeugt.

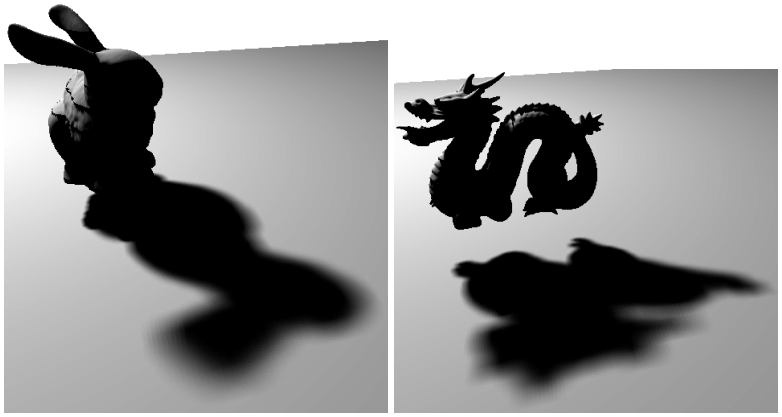
Ein Aspekt des hier vorgestellten Verfahrens ist die Verwendung von GPU-Rendering zur Erzeugung des initialen Bildes. Tabelle 1 zeigt die unterschiedlichen Berechnungszeiten auf der CPU und der GPU. Die CPU-Implementation verwendet Raytracing und die GPU das in Unterabschnitt 4.1 beschriebene GPU-Rendering. Als Testszenen werden der Bunny mit dem Dragon verglichen, um die Skalierung mit zunehmender Dreieckszahl zu untersuchen. Die Raytracing-Implementation benötigt hierfür in Tabelle 1 0,179 Sekunden für den Bunny und 0,198 Sekunden für den Dragon. Durch Verwendung des GPU-Rendering ergibt sich in ein Performance-Gewinn von 0,17 Sekunden beim Bunny, bzw. 0,19 Sekunden beim Dragon, gegenüber dem CPU-basierten Raytracing. Die GPU benötigt 0,001 Sekunden für den Bunny und 0,0019 Sekunden für den Dragon.



(a) 4 Strahlen (b) 4 Strahlen

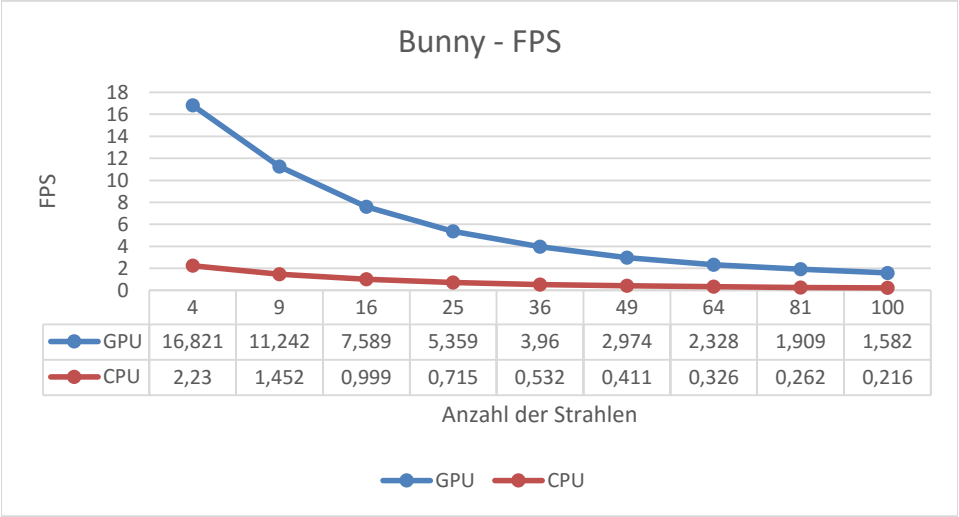


(c) 25 Strahlen (d) 25 Strahlen

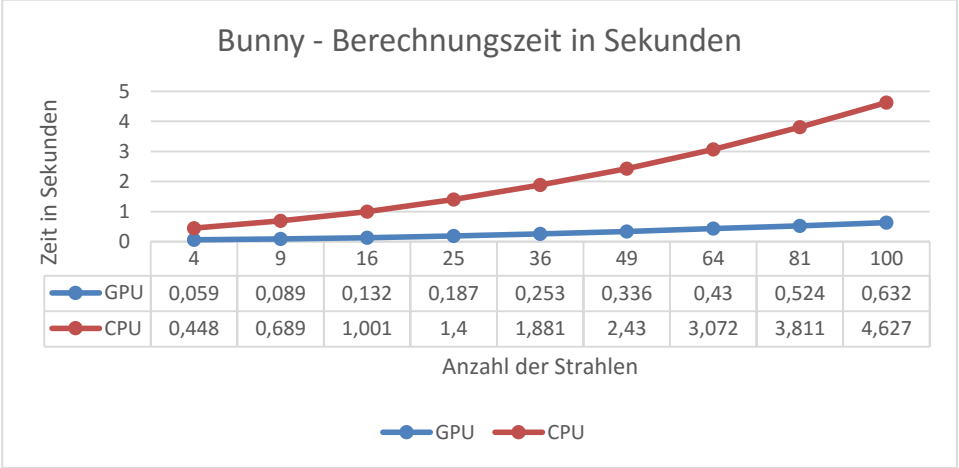


(e) 100 Strahlen (f) 100 Strahlen

Abbildung 11: Resultat der Schattenberechnung für den Stanford-Bunny (69k Dreiecke) und den Stanford-Dragon (871k Dreiecke). Es wurde ein N-tree der Tiefe 3 mit einer Auflösung von 10x10x10 traversiert. (a) und (b) zeigen Artefakte an den Rändern der Schatten. Diese treten in (c) und (d) durch die Verwendung von 25 Strahlen nicht auf. (e) und (f) zeichnen den Schatten weicher, durch eine höhere Abtastung.



(a)



(b)

Abbildung 12: Die Lichtquelle wird mit 2x2 bis 10x10 Strahlen uniform abgetastet. Die Berechnungszeiten der Schatten werden in (a) in Frames-pro-Sekunde (FPS) und in (b) in Sekunden angegeben. Die GPU zeigt dabei durchgängig bessere Ergebnisse als die CPU. (a) Die FPS sinken bei der GPU-Implementation nicht unter 1 FPS. (b) Die Berechnungszeiten der GPU und CPU zeigen große Differenzen. Die CPU benötigt mehrere Sekunden für mehr als 36 Strahlen, wohingegen die GPU unter einer Sekunde für die Berechnungen benötigt.

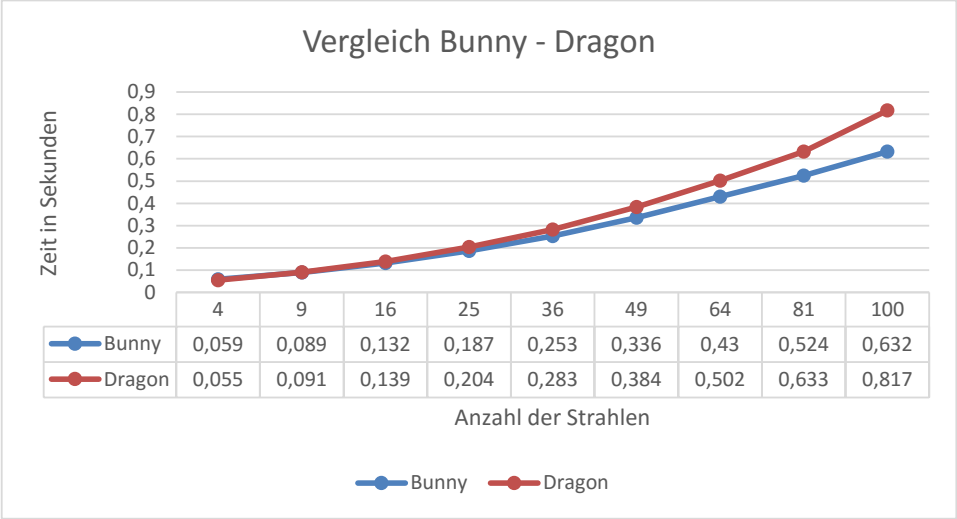


Abbildung 13: Vergleich der Rechenzeiten des Bunny und des Dragon. Der Bunny besteht aus 69 tausend Dreiecken und der Dragon aus 871 tausend Dreiecken. Für 4 bis 16 Strahlen bleiben die Berechnungszeiten ähnlich, unterscheiden sich jedoch für 100 Strahlen stärker. Der N-tree enthält beim Dragon mehr gefüllte Knoten und somit mehr Knoten und Unterknoten die traversiert werden, wodurch ein Performance-Unterschied entsteht.

Die Berechnung der Schatten wurde für einen direkten Vergleich auf der CPU und der GPU programmiert (siehe Unterunterabschnitt 4.3.1). Zum Messen der Performance-Optimierung durch Verwendung der GPU, wird die Lichtquelle uniform abgetastet. Die Messungen starten mit 2x2 Strahlen und enden mit 10x10 Strahlen. Als Testszene dient hierfür der Bunny. Die Berechnung der Schattenstrahlen wird dabei in Abbildung 12 um 86% schneller auf der GPU ausgeführt. Dieser Performance-Gewinn bleibt für alle Messungen konstant.
Um die Skalierung des Verfahrens mit komplexeren Szenen zu ermitteln, wird der Bunny mit dem Dragon verglichen. Diese Tests werden in der GPU-Implementation durchgeführt. Abbildung 13 zeigt ähnliche Berechnungszeiten für vier bis 16 Strahlen. Der Unterschied der Schattenberechnung des Dragon liegt bei 3% für vier Strahlen und wächst auf 22% für 100 Strahlen an.

Vergleich Schnitttests-Linespace: Bunny - 4 Strahlen	
Variante	Zeit in Sekunden
CPU Schnitttests mit Dreiecken	6,064665
CPU Linespace-Abfrage	0,448443

Tabelle 2: Vergleich der Berechnungszeiten von 4 Schattenstrahlen. Variante 1 zeigt die Berechnungszeit für Dreiecksschnitttests, Variante 2 die Verwendung des Linespace zur Bestimmung eines Treffers mit der Geometrie. Der Unterschied entsteht durch die Traversierung einer weiteren Ebene, sowie der Verwendung von Schnitttests. Die Benutzung des Linespace ermöglicht eine frühe Terminierung eines Strahles, ohne Untersuchung der Dreiecke. Zusätzlich muss die tiefste Ebene des N-tree nicht traversiert werden, da diese keine Linespaces besitzt (siehe Unterabschnitt 3.3).

Algorithmen-Vergleich auf der CPU: Bunny				
Variante	4 Strahlen Zeit in Sekunden	9 Strahlen Zeit in Sekunden	16 Strahlen Zeit in Sekunden	100 Strahlen Zeit in Sekunden
CGT	0,628609	0,867256	1,18496	4,82775
DDA	0,356825	0,550786	0,829777	3,88766

Tabelle 3: Vergleich des CGT-Algorithmus mit dem DDA-Algorithmus. Beide Algorithmen werden auf der CPU ausgeführt und berechnen die Primärstrahlen durch Raytracing. Die Messungen werden mit 4, 9, 16 und 100 Schattenstrahlen ausgeführt und die jeweiligen Berechnungszeiten angegeben. Beide Verfahren traversieren über einen N-tree der Auflösung 10x10x10 mit einer maximalen Tiefe von 3. Die Ergebnisse zeigen Optimierungspotential des CGT-Algorithmus.

Die Verwendung des Linespace in Unterunterabschnitt 4.3.2 dient der Bestimmung eines Treffers eines Strahles mit der Geometrie. Um den Unterschied zwischen Linespace-Abfragen und Schnitttests mit Dreiecken zu messen, werden vier Strahlen mit dem hier vorgestellten CGT-Algorithmus verfolgt. Vier Strahlen sind hierbei die minimale Anzahl an Strahlen die verfolgt werden können, da Sie einen Schaft aufspannen (siehe Unterabschnitt 4.3). Die Verwendung von Schnitttests läuft in Tabelle 2 mit 6 Sekunden Berechnungszeit deutlich langsamer als die Verwendung des Linespace mit 0,45 Sekunden. Der CGT-Algorithmus profitiert von dem Überspringen der Schnitttests, sowie der tiefsten Ebene des N-tree. Zur Berechnung von Schnitttests muss diese traversiert werden, um Dreiecke aus den getroffenen Knoten auszulesen. Sie muss bei der Verwendung des Linespace nicht traversiert werden, da ein Linespace maximal auf der letzten Ebene mit Unterknoten berechnet wird (siehe Unterabschnitt 3.3).

Diese Implementation des CGT-Algorithmus präsentiert eine alternative Berechnungsweise für Halbschatten. Anders als beim DDA-Algorithmus werden die Strahlen hier als Paket verfolgt und nicht einzeln (siehe Unter-

abschnitt 4.3). Um diese beiden Algorithmen miteinander zu vergleichen werden vier Messungen mit 4, 9, 16 und 100 Strahlen durchgeführt. Beide Algorithmen nutzen dabei den Linespace als Abbruchkriterium, wie in Unterunterabschnitt 4.3.2 beschrieben.
Im Vergleich zum DDA-Algorithmus weist der CGT-Algorithmus in Tabelle 3 für alle Messungen längere Berechnungszeiten auf. Für vier Strahlen unterscheidet sich die Zeit um 0,272 Sekunden. Der CGT Algorithmus benötigt hierbei 0,629 Sekunden und der DDA 0,357 Sekunden. Der Unterschied wächst auf 0,94 Sekunden für 100 Strahlen. Die Zeiten betragen hierfür 4,828 Sekunden für den CGT- und 3,889 Sekunden für den DDA-Algorithmus.
Ein Problem entsteht durch die Schicht-basierte Traversierung des N-tree. Dabei werden Strahlen mit Knoten geschnitten, die beim DDA-Algorithmus nicht traversiert werden. Die gemessenen Zeiten zeigen Optimierungspotential für den CGT-Algorithmus. So wurde beispielsweise bisher die in [WIK+06] beschriebene SIMD-Befehlssatz-Erweiterung (engl. Single-Instruction-Multiple-Data) nicht verwendet.

6 Fazit

In dieser Arbeit wurde eine Möglichkeit vorgestellt, Halbschatten durch Verwendung vorberechneter Schäfte zu berechnen. Dabei wird ein N-tree als Datenstruktur traversiert, der in seinen Knoten zusätzlich einen Linespace speichert. Dieser Linespace besteht aus einer Bitmaske mit der Information auf Geometrie innerhalb vorberechneter Schäfte. Diese Information ersetzt die Durchführung von Schnitttests mit Dreiecken und dient als Indikator für einen Treffer eines Strahles mit der Geometrie.
Die Traversierung basiert auf Paketen von Strahlen. Alle Schattenstrahlen eines Pixels bilden ein Paket, welches durch den N-tree traversiert wird. Der N-tree wird in Schichten traversiert und anschließend alle Knoten innerhalb der Schicht untersucht, die vom Paket getroffen werden. Um die Traversierung auf der GPU auszuführen, wurde Sie iterativ geschrieben und ermöglicht die Traversierung mehrerer Tiefen des N-tree durch Verwendung eines Stack.
Zur Beschleunigung der Schattenberechnung werden diese im Compute-Shader ausgeführt. Die Szene wird dazu in einen FBO gerendert und in den Compute-Shader eingelesen. Zusätzlich werden der N-tree und der Linespace kompakt als Texturen in den GPU-Speicher geladen.

Die Evaluation des Verfahrens zeigt für die Berechnung der Startpositionen der Schattenberechnung einen deutlichen Vorteil des hier verwendeten GPU-Rendering anstelle von Raytracing. Die Verwendung des Li-

nespace anstelle von Schnitttests mit Dreiecken bringt eine Zeitersparnis von 92,6%. Dieser Wert resultiert vorrangig aus einer fehlenden Optimierung des Verfahrens für Schnitttests. Der Compute-Shader bringt einen Performance-Gewinn von 86% gegenüber der CPU-Implementation und zeigt zudem eine gute Skalierung mit steigender Komplexität der Szene. Im Vergleich zum DDA-Algorithmus weist der hier verwendete CGT-Algorithmus jedoch Optimierungspotential auf.

Die GPU-Implementation bietet Optimierungspotential in der effizienteren Nutzung des Compute-Shader. Eine bessere Aufteilung der Aufgaben innerhalb einer work-group ist eine Möglichkeit. Vier work-groups könnten einen Schaft verfolgen, indem jeder Thread die Berechnung eines Eckstrahles übernimmt und die Berechnungen der Schnitte eines Strahles mit einem Knoten, sowie der Linespace-Überprüfungen auf die vier Threads aufgeteilt werden. Die Ergebnisse würden dann durch den shared-memory der work-group synchronisiert.
Weiter muss die Registernutzung des Compute-Shader optimiert werden, um das Verfahren für größere N-tree Auflösungen zu optimieren. Hierbei stellt vor allem der Stack ein Problem dar, da die Speicherung aller getroffenen Knoten einer Schicht viel Speicherplatz verbraucht.
Die GPU-Datenstruktur des N-tree kann für die Speicherung von Dreiecken erweitert werden. Der grün-Wert eines Node-Texel speichert die Nummer eines Linespace. Ein Linespace ist jedoch nur für Knoten mit Unterknoten definiert und ist demnach für Blätter undefiniert. Blätter speichern Dreiecke, sodass der grün-Wert für Blätter als Zeiger auf eine Kandidatenliste genutzt werden kann. Eine Kandidatenliste ist hierbei eine Liste aus Dreiecken.

Weiter muss das Verfahren mit größeren Szenen getestet werden. Beispiele wären hierbei architektonische Szenen. Zudem wurde die Skalierung des Verfahrens mit zunehmender Lichtquellenzahl bisher nicht getestet. Weitere Optimierungen zur Verringerung der benötigten Schattenstrahlen werden hierfür benötigt.
Der Linespace als Datenstruktur besitzt weiteres Potential. Größere Auflösungen sind ein Beispiel für weitere Untersuchungen. Es kann auch untersucht werden, weitere Informationen innerhalb eines Schaft-Eintrages zu speichern, zum Beispiel getroffene Dreiecke. Das „Cascaded-Linespace“-Verfahren kann zudem für weitere Effekte der Computergrafik eingesetzt werden, wie Ambient-Occlusion und Spiegelungen. Spiegelungen würden eine Farbinformation erfordern, die als Farbe oder Zeiger auf ein Dreieck im Linespace-Schaft gespeichert werden müsste.

Literatur

[ADM+08] Thomas Annen, Zhao Dong, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. Real-time, all-frequency shadows in dynamic scenes. *ACM Trans. Graph.*, 27(3), 2008. 4

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM. 6

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987. 15

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975. 6

[CNE09] Cyril Crassin, Fabrice Neyret, and Elmar Eisemann. Building with bricks: Cuda-based out-of-core gigavoxel rendering. In *Intel Visual Computing Research Conference*, 2009. 4, 14

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM.

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. *Comput. Graph. Forum*, 30(7):1921–1930, 2011. 4

[Fer05] Randima Fernando. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. 5

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware 2005, Los Angeles, California, USA, July 30-31, 2005*, pages 15–22, 2005. 4

[GIK+07] Christiaan P. Gribble, Thiago Ize, Andrew E. Kensler, Ingo Wald, and Steven G. Parker. A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Trans. Vis. Comput. Graph.*, 13(4):758–768, 2007. 5

[Gla88] A. S. Glassner. Tutorial: Computer graphics; image synthesis. chapter Space Subdivision for Fast Ray Tracing, pages 160–167. Computer Science Press, Inc., New York, NY, USA, 1988. 8

[GMG08] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008. 4

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François X. Sillion. A survey of real-time soft shadows algorithms. *Comput. Graph. Forum*, 22(4):753–774, 2003. 4, 5

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics, SI3D 2007, April 30 - May 2, 2007, Seattle, Washington, USA*, pages 167–174, 2007. 4

[KK86] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 269–278, New York, NY, USA, 1986. ACM. 6, 23

[KLM] Kevin Keul, Paul Lemke, and Stefan Müller. Accelerating spatial data structures in ray tracing through precomputed line space visibility. Submitted. 8, 9, 10, 21

[KWH09] Aaron Knoll, Ingo Wald, and Charles D. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer*, 25(3):209–225, 2009. 5, 15

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 55–63, New York, NY, USA, 2010. ACM. 4

[ORM07] Ryan S. Overbeck, Ravi Ramamoorthi, and William R. Mark. A real-time beam tracer with application to exact soft shadows. In *Proceedings of the Eurographics Symposium on Rendering Techniques, Grenoble, France, 2007*, pages 85–98, 2007. 4

[RAMN12] Ravi Ramamoorthi, John Anderson, Mark Meyer, and Derek Nowrouzezahrai. A theory of monte carlo visibility sampling. *ACM Trans. Graph.*, 31(5):121, 2012. 3

[Rob09] Austin Robison. Interactive ray tracing on the gpu, 2009. I3D 2009 – http://graphics.cs.williams.edu/i3d09/NVIRT-Overview.pdf. 11

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 283–291, New York, NY, USA, 1987. ACM. 6

[Sny96] John M. Snyder. Area light sources for real-time graphics. Technical Report MSR-TR-96-11, Microsoft Research, March 1996. 3

[WIK+06] Ingo Wald, Thiago Ize, Andrew E. Kensler, Aaron Knoll, and Steven G. Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3):485–493, 2006. 5, 15, 16, 33