UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Erweiterung eines GPU-basierten Raytracers um lokale Linespaces

Bachelorarbeit

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

vorgelegt von

Christian Korbach

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

Koblenz, im September 2017

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Bachelorarbeit
Christian Korbach
(Mat. Nr. 213 100 843)

Thema: Erweiterung eines GPU-basierten Raytracers um lokale Linespaces

Im Bereich der Computergrafik hat sich bis heute das Raytracing als bewährte Methode bewiesen, um möglichst realistische Bilder oder Bildsequenzen zu rendern. Durch die Verwendung einer GPU kann die Renderzeit durch eine bessere Parallelisierung stark reduziert werden.. Zusätzlich werden für weitere Reduzierungen Datenstrukturen verwendet, die den Raytracer unterstützen, nur die wirklich benötigten Punkte zu verwenden. Dabei hat das Linespace-Verfahren gezeigt, dass es gegenüber herkömmlichen Datenstrukturen bessere Laufzeiten aufzeigen kann, da es durch vorberechnete Sichtbarkeiten die Auswahl dieser Punkte nochmals minimiert.

Ziel dieser Arbeit ist es, einen GPU-basierten Raytracer um lokale Linespaces zu erweitern. Bei lokalen Linespaces wird der Linespace für ein Objekt gespeichert und muss für Objektkopien nicht neu berechnet werden. Dadurch können mehrere Kopien eines Objektes, unabhängig diverser Manipulationen, ohne drastischen Anstieg der Renderzeit dargestellt werden.

Die inhaltlichen Schwerpunkte der Arbeit sind:

1. Recherche über Linespace und GPGPU
2. Einarbeitung in die nötigen Grundlagen
3. Konzeption eines geeigneten Verfahrens
4. Implementation des Verfahrens
5. Evaluation
6. Fazit

Koblenz, 16.03.2017

– Christian Korbach –

– Prof. Dr. Stefan Müller –

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.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Diese Arbeit zeigt die Verwendung einer lokalen Linespace Datenstruktur, welche auf Basis eines bestehenden GPU-basierten Raytracers mit globaler Linespace Datenstruktur konzipiert und implementiert wird. Für jedes Szenenobjekt wird ein N-Tree generiert, dessen Knoten jeweils einen Linespace besitzen. Dieser speichert in seinen Schäften Informationen über existierende Geometrie. Ein Schaft stellt ein Volumen zwischen zwei Flächen auf der Knotenaußenseite dar. Dies ermöglicht bei der Strahlverfolgung ein schnelleres Überspringen leerer Räume. Identische Objekte können auf bereits berechnete Linespaces zurückgreifen, wodurch der Speicherbedarf um bis zu 94,13% und die Initialisierungszeit der Datenstruktur um bis zu 97,15% vermindert werden kann. Aufgrund der lokalen Zugriffsmöglichkeiten können dynamische Szenen visualisiert werden. Dabei ist ebenso ein Anstieg der Qualität zu beobachten.

Abstract

This thesis presents the use of a local linespace data structure, which is designed and implemented on the basis of an existing GPU-based raytracer with a global linespace data structure. For each scene object, an N-tree is generated whose nodes each have a linespace. This saves informations about existing geometry in its shafts. A shaft represents a volume between two faces on the outside of the node. This allows a faster skipping of empty spaces during raytracing. Identical objects can access already calculated linespaces, which can reduce the memory requirement by up to 94.13% and the initialization time of the datastructure by up to 97.15%. Due to the local access possibilities dynamic scenes can be visualized. An increase in quality can also be observed.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Im Bereich der Computergrafik können durch neue und erweiterte Technologien, dreidimensionale Visualisierungen stetig verbessert werden. Dies betrifft u.a. eine möglichst fotorealistische Darstellung von Rohbildmaterial. Der Betrachter soll ein erzeugtes 3D-Bild nicht von der Realität unterscheiden können. Für dieses Ziel wird auf Verfahren zurückgegriffen, die das physikalisch korrekte Verhalten von Licht nachahmen. Durch die richtige Berechnung von Lichtquellen und deren Lichtstrahlen können Effekte, wie z.B. Lichtreflexion und Lichtbrechung, dargestellt werden. Zu diesen Verfahren gehört das Raytracing (dt. Strahlverfolgung). In der nicht erweiterten Form des Verfahrens werden Lichtstrahlen verfolgt, gegen alle Primitive der Szene getestet, bei Treffern der nächstgelegene Schnittpunkt bestimmt und sein Farbwert berechnet.
Eine realistische Darstellung hat jedoch meistens hohe Berechnungszeiten und Speicherauslastungen zur Folge. Um diesem Nachteil entgegen zu wirken, werden Hilfsmittel entwickelt, die die Prozesse vereinfachen und beschleunigen. Die meiste Berechnungszeit benötigt das Raytracing für die Schnittpunktberechnung der einzelnen Strahlen gegen alle Primitive. Datenstrukturen helfen die Auswahl der zu überprüfenden Primitive einzugrenzen, indem sie die Szene räumlich aufteilen.

Diese Arbeit erweitert die Datenstruktur eines bereits bestehenden Raytracing Verfahrens, welches die globale Linespace Datenstruktur verwendet. Sie basiert auf einem N-Tree und enthält für jeden Knoten einen Linespace. Der Aufbau des N-Trees erfolgt global auf der kompletten Szene. Innerhalb des Knotens generiert der Linespace Schäfte zwischen aufgeteilten Flächen der Außenseiten. Die Schäfte erhalten die Information über existierende Geometrie in diesen. Bei der Verfolgung eines Strahls können durch Überprüfungen der getroffenen Schäfte leere Räume schnell übersprungen werden.
Die Erweiterung ändert die globale Datenstruktur in eine lokale. Der Aufbau des N-Trees, inklusive der Linespaces, findet nicht mehr auf der kompletten Szenestruktur statt sondern für jedes Objekt der Szene. Durch Kombination des lokalen Linespaces mit Orientated-Bounding-Boxen, wird zuerst nach einem Schnittpunkt zwischen Strahl und Box gesucht, bevor der Linespace des getroffenen Objektes weiter traversiert wird.
Die Daten der Linespaces zweier identischer Objekte sind ebenfalls identisch und von Rotationen oder Translationen unabhängig. Für eine Verminderung der Speicherauslastung werden die lokalen Linespaces für identische Objekte nur einmal gespeichert.

Die globale Datenstruktur kann keine dynamischen Objekte darstellen, da sonst für jedes Bild der Linespace neu berechnet werden müsste. Aufgrund der einmaligen Berechnung eines Linespaces für ein Objekt und der Unabhänigkeit der Objektransformationen, muss sich die lokale Datenstruktur bei einer Objektbewegung nicht erneuern. So ist zudem eine Visualisierung einer dynamischen Szene mit lokalen Linespaces möglich.

1.2 Aufbau der Arbeit

Die Grundlage dieser Arbeit liefert das bereits bestehende Raytracing Framework, welches das Rendering auf der CPU(engl.) und GPU(engl.) beherrscht, mithilfe einer N-Tree oder Linespace Datenstruktur zur Beschleunigung. Das Framework wird in Kapitel 2 erläutert. Anschließend werden in Kapitel 3 die benötigten Grundlagen dargelegt. Dazu zählt das Raytracing in Unterkapitel 3.1 zum Berechnen von 3D-Bildern, Datenstrukturen in 3.2 zur Raytracing-Beschleunigung, sowie die Funktionsweise des Linespace-Verfahren in 3.3. Die Konzeption der neuen Linespace Datenstruktur wird in Kapitel 4 illustriert, die Implementation dieser in Kapitel 5. Dabei werden in 5.1 zuerst die Abläufe der Initialisierung und anschließend in 5.2 die Abläufe der Traversierung erklärt. Die Evaluation mit den Testergebnisse der neuen Datenstruktur werden in 6 dargestellt. Der Vergleich der Initialisierungszeiten und Speicherauslastungen findet in Unterkapitel 6.1 statt, während die Ergebnisse zur dynamischen Szenendarstellung in 6.2 gezeigt werden. Der Qualitätsvergleich der globalen und lokalen Datenstruktur ist in 6.3 zu sehen. Das Fazit in Kapitel 7 zieht Schlussfolgerungen zu dem neuen Verfahren und spricht mögliche Verbesserungen an.

2 Andere Arbeiten

Diese Arbeit erweitert ein bereits bestehendes Raytracing-Framework, welches durch K. Keul implementiert wurde. Das Framework besitzt Funktionen zum Rendern einer dreidimensionalen Szene durch Raytracing mithilfe einer N-Tree oder Linespace Datenstruktur. Die Klasse des GPU-basierten Raytracers `RayTracer` kann die Traversierung der Szene auf der GPU (*engl.: Grahpics Processing Unit*) ausführen, wie auch auf der CPU (*engl.: Central Processing Unit*). Er kann direkte Beleuchtung darstellen, wodurch das Anzeigen von harten Schatten ermöglicht wird.
Nach dem Erstellen eines Objektes der Klasse `RayTracer` wird in der *main*-Funktion die Methode *renderImage(...)* der Klasse aufgerufen. Diese benötigt ebenfalls den Rendertyp. Die Implementation der Erweiterung erfolgt auf der CPU und verwendet daher den Rendertyp *CPU_DIRECT*.
Der Typ wird den beiden Methoden *calcImage(...)* und *showImage(...)* übergeben, die nacheinander aufgerufen werden. Erfolgt die Traversierung auf der GPU durch den Rendertyp *GPU_DEFERRED_GPU*, werden in *calcImage(...)* die Farbwerte des Pixels berechnet und gespeichert. Die gespeicherten Daten werden anschließend in *showImage(...)* aufgerufen und das zusammengesetzte Bild dargestellt.
Die Verwendung der CPU Implementation verzichtet auf die Methode *showImage(...)* und stellt in *calcImage(...)* die berechneten Farbwerte der Pixel direkt dar. Pro Pixel in der Bildebene generiert *calcImage(...)* einen Strahl und überprüft diesen durch die Methode *trace* auf den zuerst getroffenen Schnittpunkt. *shade(...)* berechnet anschließend dessen Farbwert, falls ein Schnittpunkt gefunden wurde. Ansonsten wird der Wert auf eine definierte Hintergrundfarbe gesetzt. Zudem wird getestet ob ein anderes Objekt zwischen Lichtquelle und Schnittpunkt existiert, welches einen Schatten werfen könnte.
Die Klasse `NTreeRayTracer` und `LineSpaceRayTracer` erben von der Klasse `RayTracer` und überschreiben verschiedene Methoden, wie z.B. *trace(...)* und *shade(...)* zur Verwendung der jeweiligen Datenstruktur zur Traversierung. Die Datenstrukturen werden durch die Konstruktoren der Klassen `NTree` und `LineSpace` initialisiert. Die Verwendung der Linepace Datenstruktur zum Rendern von Rohbilddaten wurde durch die Klasse `LineSpace` und `LineSpaceRayTracer` nach Keul *et al.* aus [KKM16] implementiert.

3 Grundlagen

Zum Verständnis dieser Arbeit werden zu Beginn einige Grundlagen erläutert. Dazu gehört das Raytracing-Verfahren in Unterkapitel 3.1, auf welchem die Implementation der `RayTracer`-Klasse aus Kapitel 2 basiert. Des Weiteren werden in Unterkapitel 3.2 grundlegende Datenstrukturen erläutert, die für das Verständnis der Konzeption (Kapitel 4) und Implementation (Kapitel 5) von Nöten sind. In 3.3 wird der Aufbau des Linespace aus dem Raytracing-Framework aus Kapitel 2, dem Kernstück dieser Arbeit, erklärt.

3.1 Raytracing

Zur Visualisierung einer dreidimensionalen Szenenstruktur gibt es mehrere bewährte Verfahren zur Bildsynthese (*engl.: Rendering*) in der Computergrafik. Ein weitverbreitetes Verfahren ist das Raytracing nach Appel aus [App68].
Zur Berechnung der Farbwerte der darzustellenden Pixel auf einem Monitor wird jeweils ein Strahl in die Szene geschossen und verfolgt. Der Monitor wird in Form einer Bildebene dargestellt. Ein Strahl $\overrightarrow{S}$ ist eine Halbgerade der Form:

$$\overrightarrow{S} = \overrightarrow{A} + t(\overrightarrow{B} - \overrightarrow{A}), t \geqq 0 \tag{1}$$

Der Funktionsteil $(\overrightarrow{B} - \overrightarrow{A})$ ist unser Richtungsvektor, während $\overrightarrow{A}$ der Strahlenursprung ist. Der t-Wert gibt zusammen mit dem Richtungsvektor die Länge des Strahls an.

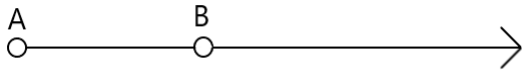


Abbildung 1: Ein Strahl (Halbgerade) ist eine Gerade, die auf einer Seite begrenzt ist und nur in eine Richtung ins Unendliche geht.

Als Ursprung der Strahlen dient ein Punkt außerhalb der Szene namens Augpunkt, welcher als Auge des Betrachters fungiert. Jeder Strahl bildet eine Halbgerade, die vom Augpunkt aus jeweils einen Punkt der Bildebene schneidet, wobei jeder Punkt einen Pixel repräsentiert.
Das Raytracing-Verfahren überprüft nun jeden Strahl auf einen Schnittpunkt, in dem der jeweilige Strahl gegen alle Primitive in der Szene getestet wird. Gewählt wird der Schnittpunkt der geringsten Distanz zum Strahlenursprung. Primitive sind geometrische Elemente, die durch Zusammenfügen ein Objekt, auch Mesh genannt, konstruieren. Dazu gehören z.B. Dreiecke oder andere Polygone.

Um die Farbe des Pixels korrekt zu bestimmen, wird, neben dem richtigen Farbwert des Materials des getroffenen Objektes, die Helligkeit benötigt. Die Berechnung der Helligkeit der Farbe findet mithilfe der Normalen der Primitive statt. Die Abbildung 2 illustriert das Verfahren anhand eines Strahls.

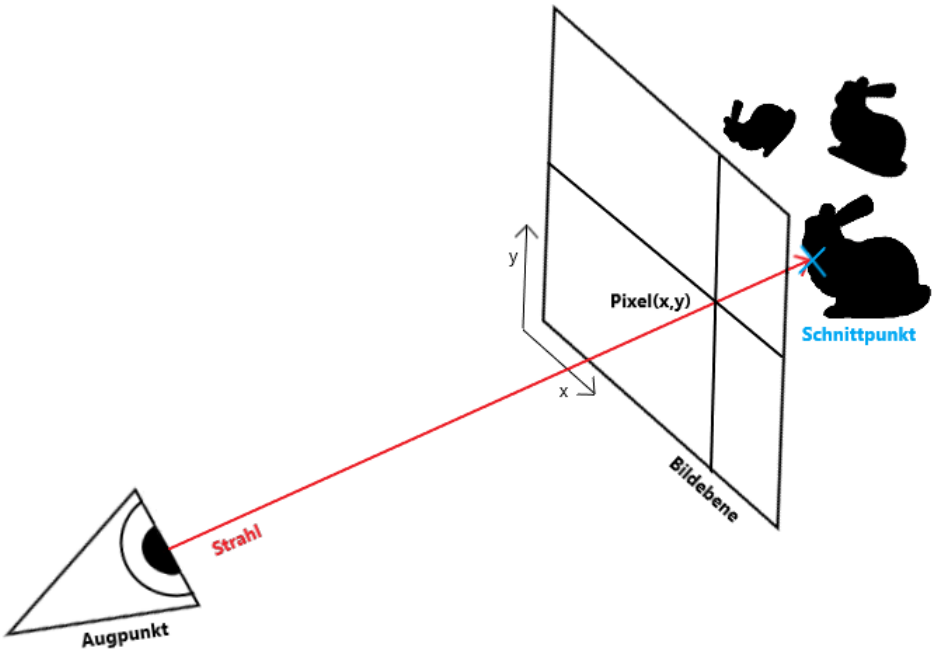


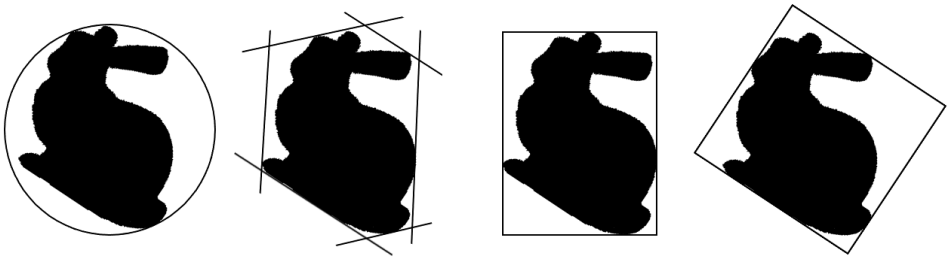
Abbildung 2: Raytracing Verfahren
Ein Strahl wird vom Augpunkt aus durch einen Pixel auf der Bildebene in die Szene geschossen und verfolgt, um den nächstgelegenen Schnittpunkt zu finden. Der Strahl wird mit jedem Primitiv der Szene getestet.

Eine wichtige Eigenschaft des Raytracings ist die annähernde physikalische Korrektheit, da die Strahlen Lichtstrahlen darstellen, die in den Augpunkt treffen. Eine einfache erweiterte Implementation ermöglicht die Berechnung von direkter Beleuchtung zur Darstellung von harten Schatten. Ein Algorithmus sucht Primitive, die einen Schatten werfen könnten, indem ein weiterer Strahl, auch Schattenstrahl genannt, vom Schnittpunkt aus zu allen Lichtquellen geschickt wird.
Weitere Modifikationen ermöglichen eine globale Beleuchtung, die neben harten und weichen Schatten auch Lichtreflexion, -brechung und -streuung darstellen können. Dazu gehören u.a. das Bidirektionale Path Tracing, wie E. Lafortune und Y. Willems in [LW93] zeigen, oder das Metropolis Light Transport Verfahren nach E. Veach und L. Guibas aus [VG97].

3.2 Datenstrukturen

Die benötigte Zeit zur Berechnung der Schnittpunkte für jeden Strahl mit allen Primitiven würde das Raytracing-Verfahren ineffizient machen. Daher werden zur Beschleunigung Datenstrukturen verwendet, die durch Vorberechnungen bereits eine Anzahl an Primitiven ausschließen können, sodass ein Strahl über eine geringere Menge solcher traversieren muss. Im Laufe der Zeit sind eine Vielzahl von Beschleunigungsverfahren entstanden. Zu diesen gehören u.a. Bounding-Volume-Hierarchien (kurz: BVH), das Uniform-Grid, der Octree und der N-Tree.

Eine BVH gehört zu den hierarchischen Datenstrukturen. Sie kreiert für jedes Objekt ein Bounding-Volumen (kurz: BV), die je nach Implementation wieder mehrere BVs als Kindknoten enthalten kann, wodurch eine Hierarchie erzeugt wird. Anstatt der Primitive werden nun zuerst die BVs untersucht. Gibt es einen Schnittpunkt mit einem Knoten, werden zunächst dessen Kinder untersucht. Bei keinem existieren Schnittpunkt kann der Knoten samt Kindern übersprungen werden. Ist ein BV ein Blattknoten, d.h. besitzt ein BV keine Kinder, wird die enthaltende Geometrie auf einen Schnittpunkt getestet. Die BVs können verschiedene Strukturen haben, wie in Abbildung 3 dargestellt.
Bounding Spheres (siehe Abb. 3a) umhüllen das Objekt durch eine Kugel, wodurch schnelle Schnittpunkttests durchgeführt werden können. Sie dienen besonders zur schnellen Kollisionserkennung, wie [JWCK08] zeigt.
k-DOPs (*engl.: k-Discretely Oriented Polytopes*) bestehen aus k-Ebenen, die das Objekt eingrenzen. Dabei stehen, wie in Abbildung 3b zu sehen, zwei Ebenen immer parallel zueinander, wodurch eine konvexe Hülle entsteht.
Abbildung 3c zeigt eine AABB (*engl.: Axis-Aligned Bounding Box*). Sie besteht aus vier Flächen, welche parallel zu den Achsen der Weltkoordinaten ausgerichtet sind. Sie sind häufig anzutreffen, da die Schnittpunkttests, wie bei der Bounding Sphere, ebenfalls sehr effizient sind und nur zwei Eckpunkte der Bounding Box benötigt werden.
Die OBB (*engl.: Orientated Bounding Box*) aus Abbildung 3d ist eine Erweiterung der AABB, bei der die Bounding Box mit der Rotation des Objektes rotiert. Dafür wird dessen Rotationsmatrix in der OBB gespeichert, wie [Sza17] in seiner Implementation zeigt.
In Abbildung 4a ist eine Verwendung von BVH's im Zusammenspiel mit k-DOPs zusehen, die eine effiziente Traversierung erzielen können, wie in [KK86] zu lesen ist.
Das Uniform Grid besitzt wie der Octree und N-Tree eine gitterförmige Struktur, wodurch ein 3D-Raster in der Szene aufgebaut wird. In 4b hat dieser eine Auflösung von $n = 5$, was zu einer Menge von 125 Würfel, bzw. Knoten führt. Jeder Knoten verfügt über die Information, ob Geometrie enthalten ist oder nicht.



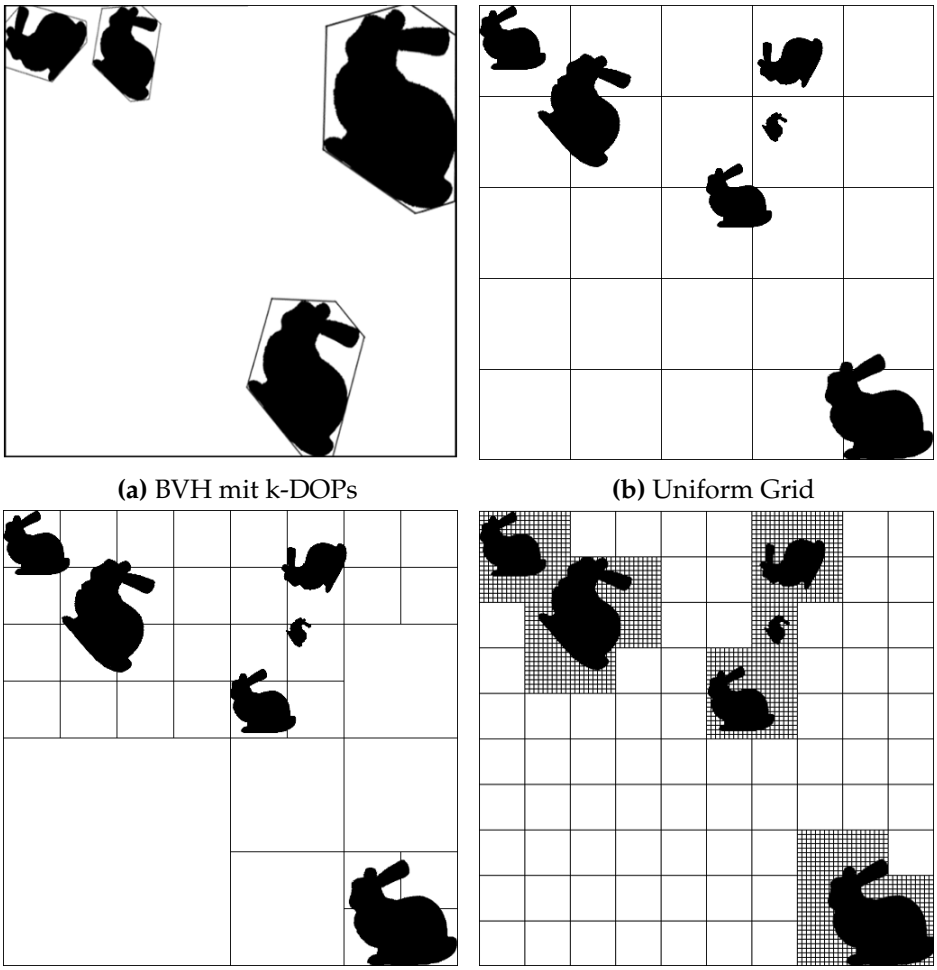
(a) Sphere **(b)** k-DOPs **(c)** AABB **(d)** OBB

Abbildung 3: 2D-Darstellung verschiedener Bounding Volumes
(3a): Eingrenzung durch eine Kugel.
(3b): k-DOPs grenzen das Objekt durch k-Ebenen ein.
(3c): Eine AABB hüllt das Objekt in einen Quader, dessen Seiten an den Koordinatenachsen ausgerichtet sind.
(3d): Eine OBB ist eine AABB, die sich mit dem umhüllten Objekt rotiert.

Um nicht durch große leere Räume traversieren zu müssen, fasst der Octree aus Abbildung 4c diese zusammen. Die Szene teilt sich in 2x2x2 Würfel. Enthält ein Knoten Geometrie, teilt dieser sich wieder auf, bis eine gewählte Rekursionstiefe d erreicht wurde. Bei der Traversierung werden leere Räume nun schneller übersprungen wie beim Uniform Grid.
Der N-Tree ist eine erweiterte Variante des Octrees, bei dem die Auflösung variabel ist und selbst gewählt werden kann. Abbildung 4d zeigt einen N-Tree mit einer Auflösung von $n = 10$ und einer Tiefe von $d = 2$. Wie bereits beim Octree wird in größere freie Räume nicht tiefer traversiert.

3.3 Linespace

Die Generierung eines Linespaces aus [KKM16] erfolgt auf der Grundstruktur eines N-Tree. Für jeden N-Tree Knoten wird einmalig ein Linespace erstellt, dessen Auflösung n der des N-Trees entspricht. Bei einer Auflösung von n erhält jede Außenseite eines Knotens n^2 unterteilte gleichgroße Flächen, auch *face* genannt. Es entsteht eine Gesamtanzahl von $6 * n^2$ Flächen, die nun jeweils ein *Startface* und ein *Endface* repräsentieren.
Anschließend wird zwischen jedem Start- und Endface, die sich nicht auf derselben Außenseite befinden, ein Schaft generiert. Abbildung 5a illustriert einen Schaft zwischen Start- und Endface. Jeder Schaft erhält einen Linespace-Eintrag, welcher durch einen boolean-Wert angibt, ob in diesem Schaft Geometrie enthalten ist oder nicht. Für die Zuweisung traversiert ein Algorithmus durch alle Knoten, die der Schaft zwischen Start- und Endface schneidet. Zuvor werden Binärmasken für die gefüllten Unterknoten erstellt. In Abbildung 5b wird die passende Bitmaske aus 5a dargestellt.
Durch die Binärmasken werden die Knoten auf Geometrie überprüft und



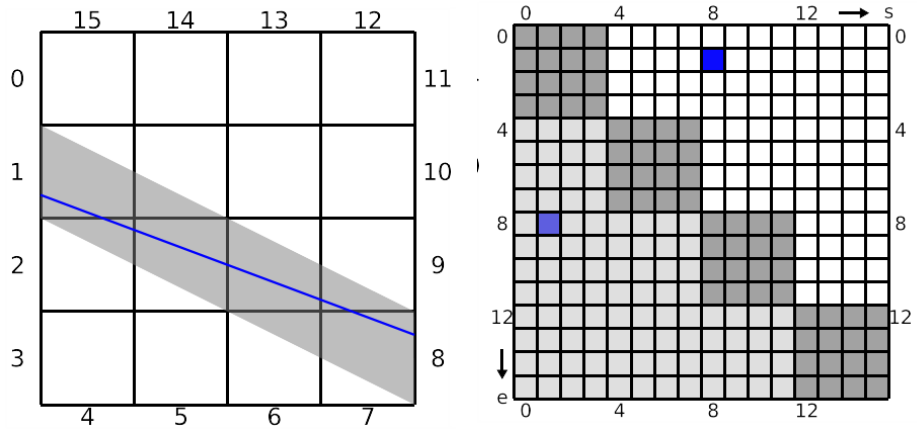
(a) BVH mit k-DOPs

(b) Uniform Grid

(c) Octree

(d) N-Tree

Abbildung 4: 2D-Darstellung verschiedener Datenstrukturen
(4a): Eine hierarchische Szenenaufteilung durch die BVH zusammen mit k-DOPs zur Aufteilung von Geometrie.
(4b): Das Uniform Grid unterteilt die Szene in ein gleichmäßig aufgebautes Raster (hier: 5x5x5).
(4c): Ein Octree hat eine Auflösung von 2x2x2 und generiert bei existierender Geometrie weitere Unterknoten.
(4d): Der N-Tree ist ein Octree mit variabler Größe (hier: 10x10x10).



(a) Strahl in Schaft **(b)** Eintrag in Bitmaske

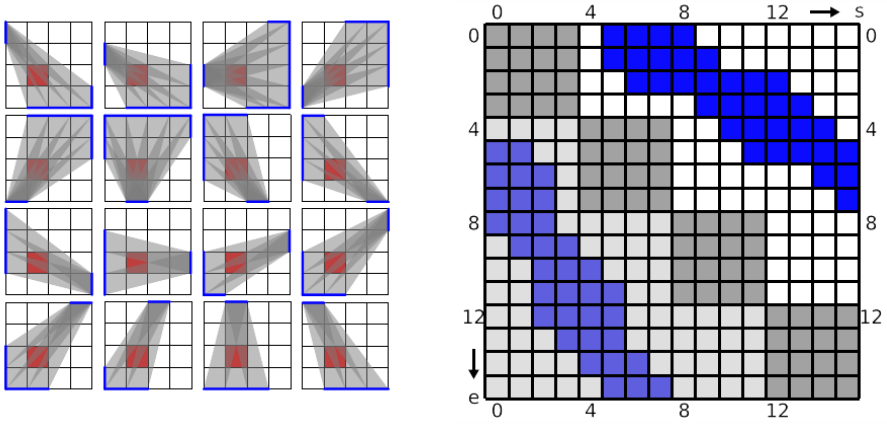
Abbildung 5: Schaft im Linespace und passender Bitmaske (Quelle: [KKM16])
5a zeigt einen Strahl (blau) mit zugehörigem Schaft(grau) innerhalb eines Knotens der Auflösung $n = 4$. Der Schaft ist durch sein Startface $s = 1$ und Endface $s = 8$ definiert. In 5b ist sein Eintrag in der Bitmaske des Linespaces zu sehen. Die dunkelgrau hinterlegten Flächen liegen jeweils auf der gleichen Außenseite des Knotens, zwischen denen kein Schaft existiert.

bei Existenz wird der Eintrag auf *true* gesetzt. Desweiteren zeigt Abbildung 6a einen Knoten der Auflösung $n = 4$, dessen Kind Geometrie (rot) besitzt. In 6b ist die zugehörige Bitmaske zu sehen, auf die die Startfaces (blau) und Endfaces (lila) aus 6a projiziert werden.

In Abbildung 5b und 6b ist eine Symmetrie zwischen den Einträgen in den Binärmasken zu erkennen. Damit diese identischen Schäfte, bei denen das Startface s dem Endface e und andersrum entspricht, nicht doppelt gespeichert werden, findet der Aufbau der Bitmaske eines Knotens symmetrisch statt. Für diese Speicherverringerung gilt für einen Eintrag *entry*:

$$entry(s, e) = entry(e, s) \tag{2}$$

Ursprünglich wird bei Traversierung eines Strahles über den N-Tree jeder getroffene Knoten der Entfernung nach untersucht. Besitzt ein Knoten Kinder, müssen diese zunächst überprüft werden. Dies kann zu erhöhten Traversierungszeiten führen, falls eine hohe Anzahl von leeren Knoten überprüft werden muss, bevor ein gefüllter Knoten gefunden wird.
Mithilfe des Linespace werden leere Bereiche schnell übersprungen, da anstatt aller Knoten und Knotenkinder, der Linespace-Eintrag des getroffenen Schafts eines Knotens überprüft wird. Nur wenn dieser Schaft gefüllt ist, müssen die Kinder des Knotens weiter traversiert werden. Bei einem leeren Eintrag 6a kann direkt mit dem nächsten Knoten fortgefahren werden.



(a) Knoten mit Geometrie **(b)** Einträge in Bitmaske

Abbildung 6: Knoten mit Geometrie und passender Bitmaske (Quelle: [KKM16])
In 6a ist ein Knoten mit einer Auflösung von $n = 4$ zu sehen, dessen Unterknoten (rot) Geometrie besitzt. Aufgelistet werden alle möglichen Kombinationen der Schäfte (grau) zwischen Startface s (blau) und Endface e (lila), die diesen Unterknoten schneiden. Die zusammengefassten Einträge in der Bitmaske sind in Abbildung 6b abgebildet. Zu erkennen ist eine Symmetrie der Einträge (blau). Die dunkelgrau hinterlegten Flächen liegen, wie in 5b, jeweils auf der gleichen Knotenseite.

4 Konzeption

In diesem Kapitel wird die grundlegende Programmstruktur dargestellt. Das Raytracing Framework aus Kapitel 2 beinhaltet bereits alle benötigten Werkzeuge, um eine beliebige Szene mithilfe des globalen Linespaces zu rendern. Dieser soll nun nicht mehr auf der kompletten Geometrie einer Szene generiert werden, sondern einmalig für jedes einzigartige Mesh. Für jedes Objekt in der Szene wird eine Bounding Box generiert. Bei der Verfolgung eines Strahls wird dieser nun zuerst gegen die Boxen getestet, bevor mögliche Schnittpunkte mit der Geometrie anhand des lokalen Linespaces überprüft werden. Die gegebene Struktur wird um Klassen und Methoden erweitert, um die Datenstruktur lokal auf jedes Objekt anwenden zu können.

Der MergedNode (siehe Abbildung 7) stellt die Basis zur Generierung eines lokalen Linespaces dar und leitet sich aus der Klasse `MergedScene` aus dem Raytracing Framework ab. Sie speichert Geometrie und Materialien der kompletten Szene. Eine `MergedNode` besitzt eine ähnliche Struktur, enthält jedoch nur die geometrischen Daten eines einzelnen Objektes ohne das Material, sowie den dazugehörigen Index *meshIndex* zur eindeutigen Identifikation der enthaltenden Geometrie. Zu den geometrischen Daten zählen die Attribute *vertexPositions* zum Speichern aller Punkte, *vertexNormals* für deren Normalen und *triangleIndices* zum Zusammenfügen der Punkte zu Dreiecken. Während der Initialisierung wird der minimalste und maximalste Punkt der *vertexPositions* berechnet und gespeichert. Die Geometrie und deren Extrempunkte werden zur Erstellung des Linespaces benötigt. Zu beachten ist, dass die Klasse die Geometrie ohne die urspünglichen Transformationen des Knoten speichert. Stattdessen wird diese mit einer neuen Matrix transformiert, die nur die Skalierung enthält. Abhängig ist die Matrix von dem größten Skalierungsvektor der Matrizen aller Knoten mit demselben Index. Dadurch werden grafische Fehler vermieden, die sonst durch die Abbildung des untransformierten Objektes auf ein größeres identischen Objekt zu einem Detailverlust führen würden. Zusätzlich zu der Geometrie werden die Punkte ein zweites Mal in *vertexPositionsMesh* ohne Transformationen gespeichert. Sie werden zum Darstellen dynamischer Szenen gebraucht. Die Initialisierung des Attributes *transformationsMatrix* findet zur Laufzeit statt und dient dem temporären Sichern einer Matrix.

Die OBB (siehe Abbildung 8) beinhaltet zusätzlich zur ihrem Mittelpunkt *centroid*, halbierten Größe *halfsize* und Orientierung *orientation*, den minimalsten und maximalsten Punkt der Bounding Box. *minPos* und *maxPos* werden während der Initialisierung berechnet und gespeichert. Die Wer-

```
struct MergedNode {
        unsigned int meshIndex;
        glm::vec3 minPos;
        glm::vec3 maxPos;
        glm::mat4 transformationsMatrix;

        std::vector<glm::vec4> vertexPositionsMesh;
        std::vector<glm::vec4> vertexPositions;
        std::vector<glm::vec4> vertexNormals;
        std::vector<MergedTriangle> triangleIndices;
};
```

Abbildung 7: Attribute der Klasse `MergedNode`

te werden zur Bestimmung eines Schnittpunktes zwischen Box und Strahl benötigt. Durch die Vorberechnung der Extrempunkte müssen diese somit nicht zur Laufzeit erneut berechnet werden. Die Belegung des Attributes *lengthToHit* (3) findet während der Laufzeit statt. Es speichert die Distanz des Mittelpunktes der Box zu dem Ursprung des zu traversierenden Strahles ab. Zur Identifizierung der zugehörigen Geometrie erhält die OBB, wie bereits die MergedNode, die ID des Objektes. Die Attribute *mnPos* und *mPos* fungieren als eine Art Zeiger, um die Geometrie und Material der OBB im jeweiligen Container wiederzufinden. Anhand des gespeicherten Knotennamen *nodeName* kann die OBB ebenfalls mit dem richtigen Szeneknoten verbunden werden.

$$obb.lengthToHit = \text{distance}(obb.centroid, hit.origin) \tag{3}$$

Ein Material enthält einen Integer *matIndex* zur Identifikation des Materialien Indizes, sowie einen Container des verwendeten Materials.

Der LineSpaceContainer initialisiert und verwaltet alle Szenedaten. Dazu gehört ein Material und eine OBB pro Objekt. Die MergedNodes und Linespaces werden einmal pro einzigartiger Geometrie erstellt. Zur Überprüfung der Existenz wird der Container durchlaufen und der aktuelle Index der Geometrie des Objektes mit den Indizes der bereits Gespeicherten verglichen. Wurde eine Übereinstimmung gefunden, kann das Objekt übersprungen werden. Die OBB's erhalten einen Paramter, der auf den Linespace des Objektes verweist. Abbildung 9 illustriert die Vorgehensweise der Klasse.

Der LocalLineSpaceRayTracer erbt von der Klasse `RayTracer` aus Kapitel 2 und benutzt dieselbe Struktur zum Rendern der Szene, indem das

```
struct OBB {
        glm::vec3 centroid;
        glm::vec3 halfsize;
        glm::vec3 orientation;
        glm::vec3 minPos;
        glm::vec3 maxPos;

        unsigned int meshIndex;
        unsigned int mPos;
        unsigned int mnPos;
        std::string nodeName;
        float lengthToHit;
};
```

Abbildung 8: Attribute der Klasse `OBB`

Bild zunächst berechnet und anschließend dargestellt wird. Zur Traversierung der lokalen Datenstruktur wird die Methode *trace(...)* überschrieben. Abbildung 10 stellt den neuen Ablauf dar.
Im ersten Schritt wird der Container *obbScene* überprüft und gegebenenfalls sortiert, falls dieser nicht aufsteigend nach der Länge zwischen OBB-Mittelpunkt und Strahlenursprung sortiert ist.
Im zweiten Schritt untersucht die Methode die OBB's auf einen Schnittpunkt. Bei einem Treffer, wird dessen t-Parameter gespeichert, um anschließend den neuen Schnittpunkt zu berechnen. Der zugehörige MergedNode und Linespace der getroffenen OBB wird ausgelesen und der Strahl in die lokalen Koordinaten des Linespaces transformiert. Die Traversierung des Linespaces gibt Aufschluss darüber, ob der Strahl auch Geometrie schneidet. Dann wird der t-Parameter erneut überschrieben, um die Länge des Strahls im nächsten Durchlauf zu verkleinern, damit keine weitere Geometrie hinter der getroffenen getestet wird. Für die Schattenberechnungen werden Informationen über die Positionen der MergedNode, des Materials und der Transformationsmatrix der OBB benötigt, die dem Strahl mitübergeben werden. Die boolean Variable *isIntersection*, welche zu Beginn auf *false* gesetzt ist, wird bei erfolgreichem Schnittpunkttest auf den Wert *true* gesetzt.
Im dritten und letzten Schritt der Traversierung werden nach jedem Durchlauf Ursprung und Richtung des Strahls zurückgesetzt. Der Rückgabewert *isIntersection* wird nach der Iteration zurückgegeben.
Nach dem Überprüfen eines Schnittpunktes berechnet die Methode *shade(...)* bei Erfolg den Farbwert des getroffenen Punktes oder weist bei Nichterfolg dem Farbwert die Hintergrundfarbe zu.

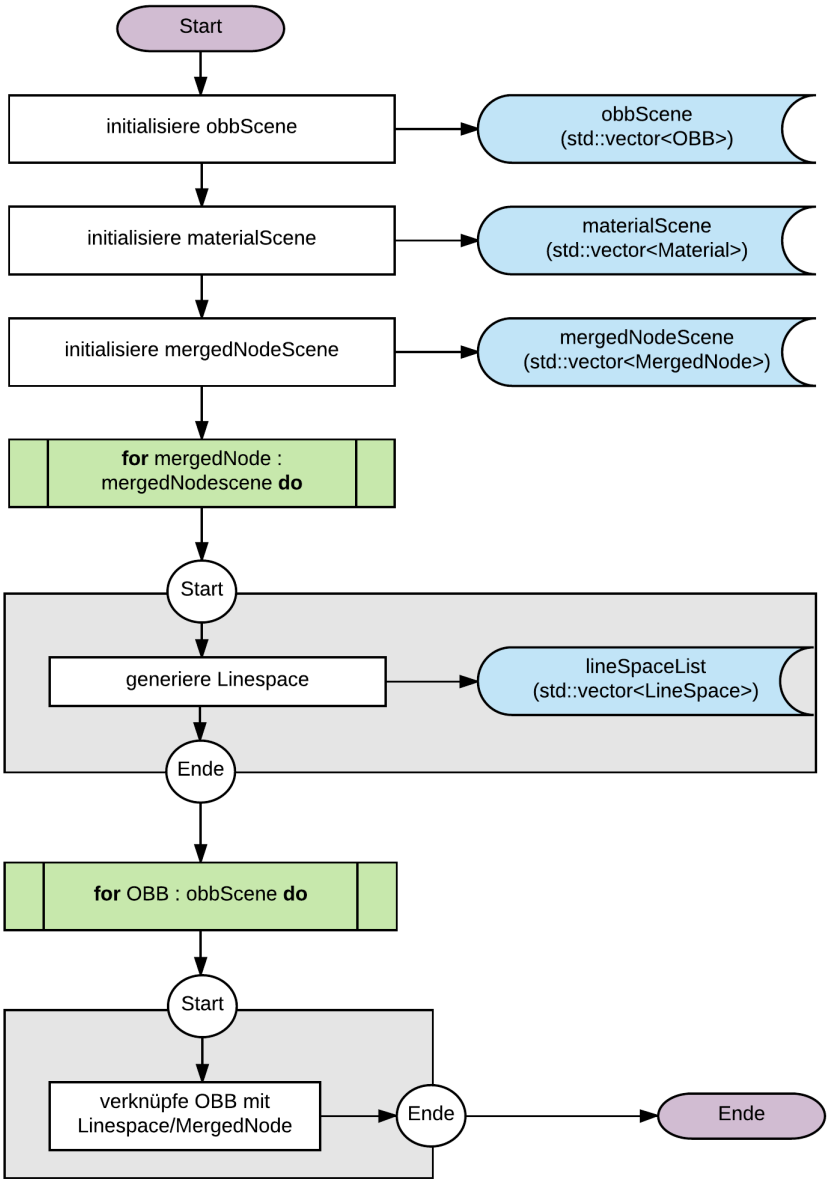


Abbildung 9: Das Flussdiagramm zeigt den Ablauf des Konstruktors der Klasse `LineSpaceContainer` zum Speichern aller benötigten Daten zur späteren Traversierung der lokalen Datenstruktur.

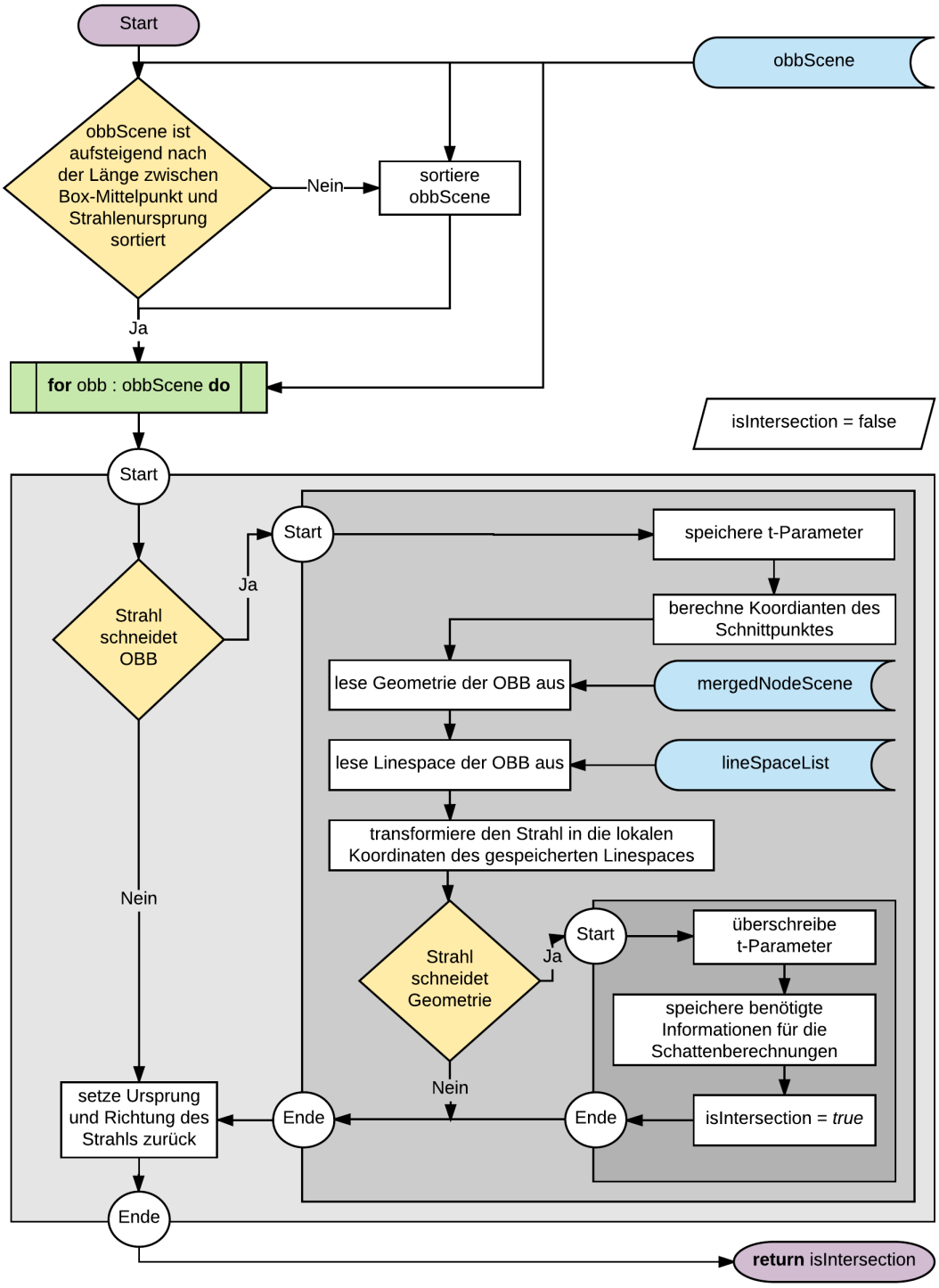


Abbildung 10: Flussdiagramm der Methode *trace(hit)* aus der Klasse `ObjectLineSpaceRayTracer` zur Traversierung eines Strahls.

5 Implementierung

Das Kapitel beinhaltet die Implementierung der in Kapitel 4 dargestellten Konzeption. Der erste Abschnitt behandelt die Intialisierung der Datenstruktur. Im zweiten Abschnitt wird die Traversierung der Szene durch diese erklärt.

5.1 Initialisierung

Zu Beginn eines Programms wird die Szene geladen und als Szeneknoten gespeichert. Die benötigten Daten zur Traversierung werden durch die Klasse `LineSpaceContainer` initialisiert. Dazu gehören die Materialen, Geometrien, Bounding Boxen und Linespaces der Objekte. Gespeichert werden sie in sequentiellen Containern "*std::vector<T>*", die Arrays dynamischer Größe kapseln, wie in [CPP17] beschrieben wird. Der Vorteil liegt darin, dass die Elemente zusammenhängend gespeichert werden, wodurch ein Zeiger auf ein Element eines Vektors an eine beliebige Funktion übergeben werden kann. Zudem wird der Speicherplatz des Vektors automatisch angepasst, wodurch ein effizienteres Speichermanagement erreicht wird.

Zuvor müssen die Parameter n und d definiert und dem anschließend dem `LineSpaceContainer`, inklusive dem Szeneknoten, übergeben werden. Transformationsmatrizen der Objekte können bis zu diesem Schritt geändert werden. Der Pseudocode der Initialisierung des LineSpaceContainers wird in Algorithmus 1 dargestellt, während die Methoden zum Erstellen der der einzelnen Container nun genauer erläutert werden.

createMergedNodeScene(...) iteriert durch die Szene und speichert pro Mesh eine *mergedNode*. Knoten mit identischer Geometrie werden übersprungen. Die Methode *getMaxScaleMat(...)* sucht anhand des *meshIndex* und der *transformationsMatrix* der Szeneknoten nach dem Mesh mit dem größten Skalierungsvektor. Anschließend wird durch die Methode *isSquare(...)* festgestellt, ob *minPos* und *maxPos* einen Würfel bilden. Wenn die x-, y- und z-Werte des Vektors *diff*, der die Differenz der beiden Punkte wiederspiegelt, nicht identisch sind, ruft sich die Methode *makeSquare(...)* auf. Sie verschiebt die Extrempunkte so, dass der neue Differenzvektor identische Werte besitzt und alle Seiten gleich lang sind. Den Mittelpunkt der Skalierung bildet der aktuelle Mittelpunkt zwischen *minPos* und *maxPos*, sodass sich dessen Position während der Transformation nicht verändert, wie in Abbildung 11 zu sehen ist.

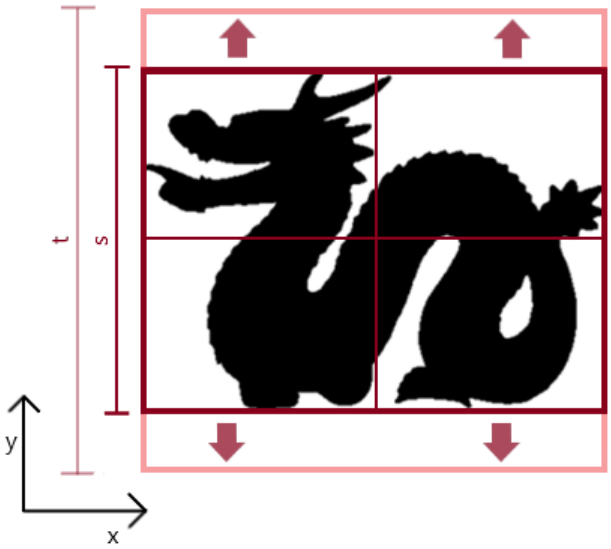


Abbildung 11: Erweiterung einer Bounding Box zu einem Würfel.
Es wird die längste Seite *t* der Box gesucht und die kleineren Seiten *s* auf deren Länge angepasst. Dabei dient der Mittelpunkt der Box ebenfalls als Mittelpunkt der Skalierung. Durch eine gleichmäßige Extension in die zu vergrößernden Richtungen findet keine Verschiebung des Mittelpunktes statt.

createOBBScene(...) kreiert für jede Node eine Orientiere Bounding Box. Genau wie in *createMergedNodeScene(...)* findet eine Erweiterung zu einem Würfel statt, falls dieser nicht schon bereits existiert. Das Attribut *centroid* wird dazu als Mittelpunkt verwendet.

createMaterialScene(...) iteriert ebenfalls durch die Szene. Pro Knoten wird ihr Material und dessen Index gespeichert. Es wird in *shade(..)* zum Darstellen der Farbe eines Pixels benötigt.

createLocalLineSpace(..) generiert einen Linespace eines einzelnen Objektes. Die Methode gleicht der Initialisierung des globalen Linespaces aus dem Framework aus Kapitel 2. Der Unterschied besteht darin, dass der lokale Linespace auf einem einzelnen Objekt, anstatt auf einer kompletten Szene, aufbaut. So wird eine `mergedNode` und keine `Scene` übergeben. Der generierte Linespace wird dem Container *lineSpaceList* hinzugefügt.

Anschließend erfolgt die Initialisierung des `LocalLineSpaceRayTracer`, welcher die Daten aus dem `LineSpaceContainer` zur Traversierung erhält. Sie werden den Attributen der Klasse übergeben.

Algorithm 1 Pseudocode: Initialisierung des LineSpaceContainer

Im ersten Schritt werden die Container *obbScene*, *mergedNodeScene* und *materialScene* initialisiert. Im zweiten Schritt iteriert eine for-Schleife durch *mergedNodeScene* und generiert pro Iterationsschritt den Linespace einer `mergedNode`, bzw. eines Objektes. Zuletzt werden die Bounding Boxen der *obbScene* mit dem passenden Linespace verlinkt. Zur Identifikation dient die ID der Geometrie.

Input:

n	Die Auflösung
d	Die Tiefe
sceneNode	Der Szenenknoten

```
1:  function LINESPACECONTAINER(n, d, sceneNode)
2:      scene ← createScene(sceneNode)
3:      mergedNodeScene ← createMergedNodeScene(scene)
4:      obbScene ← createOBBScene(scene)
5:      materialScene ← createMaterialScene(scene)
6:
7:      for all mergedNodes in mergedNodeScene do
8:          createObjectLineSpace(n, d, scene)
9:      end for
10:
11:     for all obb in obbScene do
12:         for all mergedNode in mergedNodeScene do
13:             if obb.meshIndex == mergedNode.meshIndex then
14:                 obb.mnPos ← mergedNode.pos
15:             end if
16:         end for
17:     end for
18: end function
```

5.2 Traversierung

Die Anzahl der Strahlen ist abhängig von der gewählten Auflösung des Fensters. Pro Frame müssen die Farbwerte aller Pixel bestimmt werden, wobei pro Pixel ein Strahl generiert und in Richtung der Szene geschossen wird. Vom Startpunkt ausgehend, wird dieser verfolgt und auf mögliche Schnittpunkte überprüft. Wenn ein Schnittpunkt gefunden wurde, müssen der Farbwert oder mögliche Schatten, die durch Lichtquellen entstehen, berechnet werden. Ansonsten wird die definierte Hintergrundfarbe zurückgegeben.

trace(...) wird per Pseudocode in Algorithmus 2 dargestellt. Die Methode gibt einen boolean Wert zurück, welcher angibt, ob der Strahl Geometrie trifft oder nicht. Die Variable *isIntersection* des Typ *bool* ist anfangs auf *false* gesetzt. Bei einem Treffer mit Geometrie ändert sich dieser zu *true*. Dafür werden alle enthaltenden Bounding Boxen aus *obbScene* auf Schnittpunkte getestet. Realisiert wird dies in einer for-Schleife, die durch den kompletten Container läuft und für jede Box die Methode *traceBox(...)* aufruft (Zeile 12-13). Zuvor werden in *sortBox(...)* die Bounding Boxen Distanzen zwischen dem Strahlenursprung und den Mittelpunkten der Boxen berechnet und in aufsteigender Reihenfolge neu sortiert (Zeile 8-10). Damit dies nur bei einem unsortierten Container geschieht, wird dieser vorher durch *isSorted(...)* überprüft. Bei der Traversierung des Containers werden nun zuerst die naheliegendsten Boxen überprüft, wodurch sich die Schnittpunkte je nach Szene und Anzahl der Objekte schneller feststellen lassen und falsche Berechnung eines Schattens vermieden werden.
Weist eine Bounding Box einen Schnittpunkt auf, so berechnet sich dieser auf Grundlage des überschriebenen *maxT* Parameters des Strahls. Der neue Punkt stellt nun den Ursprung des Strahls dar (Zeile 16). Der Zugriff auf den dazugehörigen MergedNode findet per Referenz statt. Um die Rotation des Objektes richtig mit einzubeziehen, müssen Ursprung und Richtung mit der inversen Transformationsmatrix des Objektes, und der Transformationsmatrix der Geometrie multipliziert werden (Zeile 17-18). Dadurch werden diese genau auf den gespeicherten Linespace abgebildet, der lediglich von einer Skalierung abhängig ist. Dieser wird ebenfalls per Referenz aufgerufen.
Der transformierte Strahl wird nun im Linespace traversiert (Zeile 19). Existiert innerhalb ein Schnittpunkt, wird der boolsche Wert *isIntersection* auf *true* gesetzt und die temporäre Variable *maxTtmp* mit dem aktuellsten maxT-Wert überschrieben. Neben der Position der Geometrie in *mergedNodeScene*, wird zudem die Position des Materials des Objektes im Container *materialScene* mit übergeben, damit die Methode *shade(...)* auf die richtigen Daten zur Farbbestimmung zugreifen kann. *MergedNodeScene* wird in diesem Fall für die Normalen des Objektes benötigt.

Nach der Traversierung eines Strahls mit einer Bounding Box werden Ursprung und Richtung zurückgesetzt, welche zu Beginn gespeichert wurden. Der *maxT* Parameter erhält für die Traversierung mit der nächsten Bounding Box den Wert von *maxTtmp*. Dadurch wird gewährleistet, dass keine Objekte getestet werden, die aus der Sicht des Strahlenursprungs bereits hinter getroffener Geometrie liegen. Dies vermeidet unnötige Durchläufe und steigert die Effizienz des Algorithmus. Nachdem kompletten Durchlauf von *obbScene*, wird *isIntersection* zurückgegeben und die Farbe kann bestimmt werden.

traceBox(...) bekommt einen Strahl, eine OBB und die inverse Richtung des Strahls übergeben. Letzteres kann auch innerhalb der Methode berechnet werden, für eine bessere Performanz berechnen wir diese jedoch schon im Vorraus. Die OBB ist zu diesem Zeitpunkt noch nicht rotiert, sodass der Algorithmus aus 3 auf den Achsen der Weltkoordinaten durchgeführt wird, da die Achsen der Box noch parallel zu diesen stehen. Die Rotationsmatrix *orientation* der OBB wird erst nach einem gefundenen Schnittpunkt benötigt.
Die Funktion bekommt *minPos* und *maxPos* der OBB übergeben, wodurch alle Eckpunkte der Box bestimmt werden können. Die t-Werte des Strahls zu den Eckpunkten einer Achse werden gesucht und in $t0$, bzw. $t1$ gespeichert (Zeile 5-10). Durch Überprüfung des inversen Richtungsvektor *invdir* gibt $t0$ den Abstand zum näheren, $t1$ den Abstand zum entfernteren Schnittpunkt an. Die `IF`-Bedingungen in Zeile 13-15 und 16-18 suchen nun den größten $t0$-Wert, sowie den kleinsten $t1$-Wert. Diese werden *tmin* und *tmax* zugewiesen. Sobald *tmin* auf dem Strahl hinter *tmax* liegt, kann die Schleife abgebrochen werden, da der Strahl die Box nicht schneidet (Zeile 20-21). Bei einer erfolgreichen vollständigen Iteration der Schleife existiert ein Schnittpunkt. Gesucht wird der nächstgelegene Schnittpunkt, sodass dessen Berechnung durch *tmin* stattfindet. Für die weitere Traversierung übernimmt der *maxT*-Wert des Strahls den errechneten *tmin*-Wert (Zeile 24).

traceLinespace(...) traversiert nun den Linespace, wie in Kapitel 3.3 beschrieben. Um auf die richtige Geometrie zuzugreifen, wird die passende MergedNode der Methode mitübergeben, welche für die enthaltende Methode *intersectCandidate(...)* benötigt wird.

shade(...) berechnet den Farbwert eines Pixels, indem auf das Material der getroffenen Geometrie zugegriffen wird. Dafür wurde in *trace(...)* die richtige Position des Materials im *materialScene* Container in *hit.mnPos* gespeichert, und die Position der MergedNode in *mergedNodeScene* in *hit.lsPos*.

Letzteres hat die Normalen des Objektes gespeichert, welche für die korrekte Lichtberechnung benötigt werden. Dafür müssen sich die Normalen mit der Transformation des Objektes mitbewegen. Mathematisch gelöst wird dies durch Multiplikation der transponierten Inversen der Transformationsmatrix des Objektes und der Normalen.

update(...) muss im Falle dynamischer Objekte pro Frame ausgeführt werden. Algorithmus 4 zeigt, dass die Methode über den übergebenen Szeneknoten auf alle Kindknoten zugreift und die *vertexPositions* der zugehörigen MergedNode aktualisiert, in dem sie die untransformierten Punkte *vertexPositionsMesh* mit der neuen Transformationsmatrix multipliziert (Zeile 2-9). Auf den neuen Werten lässt sich anschließend die neue OBB des Objektes berechnen und notfalls erweitern (Zeile 10-14). Um die Schatten, die ein Objekt bei einer Bewegung wirft, zu aktualisieren, werden in *shade(...)* die Normalen ebenfalls mit der Transformationsmatrix multipliziert.
Der Paramter *modification* des Knotens stellt eine Nebenbedingung dar. Nur wenn dieser auf *true* gesetzt ist, wird das Objekt aktualisiert. Statischen Objekten wird keine Beachtung geschenkt, sie werden übersprungen.

Algorithm 2 Pseudocode: Traversierung (allgemein)

Dem Traversierungsalgorithmus wird ein Strahl übergeben. Dieser wird auf einen möglichen Schnittpunkt mit Geometrie getestet. Die for-Schleife durchläuft alle Bounding Boxen der Szene. Bei einer erfolgreichen Suche wird auf den dazugehörigen Linespace verwiesen und dieser auf Schnittpunkte untersucht.

Input:

hit	Der generierte Strahl

Output:

isIntersection	Der boolean Wert mit der Information über einen existierenden Schnittpunkt

```
1:  function TRACE(hit)
2:      isIntersection ← false
3:      origintmp ← hit.origin
4:      directiontmp ← hit.direction
5:      maxTtmp ← hit.maxT
6:      invdir ← 1/hit.direction
7:
8:      if not isSorted(hit) then
9:          sortBox(hit)
10:     end if
11:
12:     for all obb in obbScene do
13:         if traceBox(hit, obb, invdir) then
14:             linespace ← Linespace der obb
15:             mergedNode ← Geometrie der obb
16:             iPoint ← Schnittpunkt von hit und obb in Weltkoordinaten
17:             hit.origin ← iPoint in lokalen Koordinaten
18:             hit.direction ← hit.direction in lokalen Koordinaten
19:             if traceLinespace(hit, mergedNode, linespace) then
20:                 hit.mnPos ← Position der Geometrie in mergedNodeScene
21:                 hit.mPos ← Position des Materials in materialScene
22:                 maxTtmp ← hit.maxT
23:                 isIntersection ← true
24:             end if
25:         end if
26:         hit.max ← maxTtmp
27:         hit.origin ← origintmp
28:         hit.direction ← directiontmp
29:     end for
30:     return isIntersection
31: end function
```

Algorithm 3 Pseudocode: Traversierung der Bounding Boxen

Es werden die Koordinatenachsen aller Seiten der Bounding Box auf Schnittpunkte mit einem Strahl getestet, indem die jeweiligen *t*-Parameter bestimmt werden. Ein Schnittpunkt existiert, wenn nach dem kompletten Durchlauf der FOR-Schleife der größte *tmin* Wert geringer als der kleinste *tmax* Wert ist.

Input:

hit	Der generierte Strahl
obb	Die zu traversierende Bounding Box
invdir	Der inverse Richtungsvektor des Strahls

Output:

isIntersection	Der boolean Wert mit der Information über einen existierenden Schnittpunkt

```
1:  function TRACEBOX(hit, obb, invdir)
2:      tmin ← 0.f
3:      tmax ← hit.maxT
4:      for int i = 0; a < 3; a++ do
5:          if invdir[i] >= 0.0f then
6:              t0 ← (obb.minPos[i] − hit.origin[i]) ∗ invdir[i]
7:              t1 ← (obb.maxPos[i] − hit.origin[i]) ∗ invdir[i]
8:          else
9:              t1 ← (obb.minPos[i] − hit.origin[i]) ∗ invdir[i]
10:             t0 ← (obb.maxPos[i] − hit.origin[i]) ∗ invdir[i]
11:         end if
12:
13:         if t0 > tmin then
14:             tmin ← t0
15:         end if
16:         if t1 < tmax then
17:             tmax ← t1
18:         end if
19:
20:         if tmax < tmin then
21:             return false;
22:         end if
23:     end for
24:     hit.maxT ← tmin
25:     return true
26: end function
```

Algorithm 4 Pseudocode: Update des Szenenknoten

Für die Darstellung dynamischer Objekte, werden die gespeichert Daten in den Containern aktualisiert. Der Zugriff geschieht über die Kinder des Szeneknotens, aus denen die Transformationsmatrix ausgelesen wird.

Input:
sceneNode Der Elternknoten der Szene

```
1:  function UPDATE(sceneNode)
2:      children ← Kinder des Szenenknotens
3:      for all nodes in children do
4:          if node.modification=true then
5:              obb ← Bounding Box des Knotens
6:              mn ← MergedNode des Knotens
7:              matrix ← Transformationsmatrix des Knotens
8:
9:              Berechne neue mn.vertexPositions
10:             Berechne neue obb durch mn.vertexPositions und matrix
11:
12:             if obb ist kein Würfel then
13:                 Dehne obb zu einem Würfel aus
14:             end if
15:         end if
16:     end for
17: end function
```

6 Evaluation

Zum Vergleich der globalen und lokalen Linespace Datenstruktur werden Szenen mit einer unterschiedlichen Anzahl an identischen Objekten verwendet (Bsp. in Abbildung 12). Als Testobjekte dienen der Standford-Bunny und Standford-Dragon. Diese besitzen eine Menge von 69.451 und 871.414 Dreiecken.
Zum Ermitteln der Initialisierungszeiten und der Speicherauslastung werden die Testszenen quadratisch aufgebaut. Die Anzahl der Objekte pro Achse entspricht dem Paramter m, die Gesamtanzahl der Objekte o in der Szene beträgt n^3. Die Testobjekte besitzen zueinander einen Abstand von 0.5, identische Materialien und sind unskaliert.
N-Tree und Linespace haben eine Auflösung von $n = 6$ und eine Tiefe von $d = 3$. Die Einstellungen wurden so gewählt, das ein möglichst großes Kosten-Nutzen-Verhältnis entsteht. Keul *et al.* sprechen dies in [KKM16] an. Sie verwenden eine Auflösung von $n = 10$, welche für die meisten Szenen die passende Wahl darstellt und die Speicherauslastung eingrenzt. Durch Wahl eines $n > 10$ oder $d > 5$ resultiert eine zu hohe Speicherauslastung. Bei einem $n < 5$ wird die gewünschte Qualität nicht erzielt. Um die Speicherauslastung in den Testszenen mit großen Mengen an Primitiven weiter einzuschränken, wird eine niedrigere Auflösung verwendet.
Für die Darstellung der Qualitätsunterschiede sowie der dynamischen Szenen, wurden individuelle Szenen gewählt.

(a) 2^3 Objekte **(b)** 4^3 Objekte **(c)** 6^3 Objekte

Abbildung 12: Beispiele der Testszenen mit dem Standford-Bunny.

Das Testsystem besteht aus einem 64-Bit Windows 10 Betriebssystem, Intel(R) Core(TM) i7-6700HQ Prozessor mit 4 Kernen, bzw. 8 Threads, bei einer Grundtaktfrequenz von 2,6 GHz, einer NVidia Geforce GTX 970m mit 3 Gigabyte dediziertem GDDR5-Grafikspeicher und 16 Gigabyte Random Access Memory (RAM).

6.1 Initialisierungszeiten und Speicherauslastungen

Tabelle 1 und 2 zeigen einen Überblick der ausgewerteten Daten der Testszenen. Für eine detailliertere Auflistung werden die Dragon-Testszenen verwendet. Dragon-Szenen mit einer Anzahl von $o = 7^3$ oder mehr Objekten konnten durch die globale Datenstruktur auf dem genannten Testsystem nicht erzeugt werden.
Während die ermittelnden Zeiten bei einer niedrigen Anzahl von Objekten ähnlich sind, steigt die Initialisierungszeit t_{global} der globalen Datenstruktur deutlich über die der Lokalen t_{lokal}. Die Bunny-Testszene mit $m = 10$ zeigt eine Zeitersparnis von 88,82%, während die Dragon-Testszene mit $m = 6$ eine Beschleunigung von 94,13% aufweist. Die Speicherauslastung s_{global} erhöht sich ebenfalls mit der Anzahl der Objekte auf bis zu mehrere Gigabyte, s_{lokal} hingegen bleibt unverändert. Für die Bunny-Testszene mit $m = 10$ beträgt die Verringerung des Speicherbedarfs 97,06% und 97,15% für die Dragon-Testszene mit $m = 6$.

Bunny		Initialisierungszeit in s		Speicherauslastung in MB	
m	o	t_{global}	t_{lokal}	s_{global}	s_{lokal}
1	1	1,78	1,95	396	465
2	8	2,05	1,97	443	465
3	27	2,58	2,15	763	465
4	64	4,65	2,49	1.200	465
5	125	13,55	2,92	1.700	465
6	216	20,36	3,46	3.400	465
7	343	28,14	4,35	4.900	465
8	512	39,09	5,46	7.200	465
9	729	51,13	6,98	10.700	465
10	1.000	77,02	8,61	15.800	465

Tabelle 1: Überblick über die Resultate der Bunny-Testszenen

Dragon		Initialisierungszeit in s		Speicherauslastung in MB	
m	o	t_{global}	t_{lokal}	s_{global}	s_{lokal}
1	1	9,87	13,97	947	1.017
2	8	11,43	14,61	2.000	1.017
3	27	24,94	16,51	5.300	1.017
4	64	52,78	19,86	11.000	1.017
5	125	210,77	25,60	23.900	1.017
6	343	583,49	34,25	35.700	1.017

Tabelle 2: Überblick über die Resultate der Dragon-Testszenen

Die globale Datenstruktur generiert zur Initialisierung einen N-Tree und Linespace auf der kompletten Geometrie der Szene. Die Geometrie muss zudem für die Traversierung und Schattenberechnung komplett gespeichert werden, wodurch bei komplexen Szenen ein großer Speicherbedarf besteht. Für die Darstellung der Dragon-Testszene (Tabelle 2) mit $m = 6$ müssen 343 Dragons mit 871.414 Dreiecken gespeichert werden. Dies entspricht einer Menge von 298.895.002 Dreiecke in der Szene. Ein hohes Aufkommen von Geometrie hat zur Folge, dass die Menge der N-Tree Knoten ebenso steigt. Daraus resultiert eine längere Initialisierungszeit (Tabelle (3)), sowie ein größerer Speicherbedarf (Tabelle 4). Die Größe des Linespaces hingegen richtet sich nach der Menge der getroffenen Geometrie in den jeweiligen Linespace-Shafts. Je mehr Daten die Shafts enthalten, desto höher fällt die Initialisierungszeit des Linespaces aus. Im Vergleich zum N-Tree ist diese jedoch gering.

Dragon		Initialisierungszeiten (global) in s		
m	o	t_{ntree}	t_{ls}	t_{gesamt}
1	1	6,27	0,82	9,87
2	8	7,10	0,37	11,43
3	27	14,51	0,39	24,94
4	64	29,19	0,41	52,78
5	125	59,45	0,79	210,77
6	216	143,12	0,95	583,49

Tabelle 3: Detaillierte Initialisierungszeiten der globalen Linespace Datenstruktur anhand der Dragon-Testszenen

Dragon		Speicherausl. (global) in MB			LS-Eigenschaften	
m	o	s_{ntree}	s_{ls}	s_{gesamt}	n_{shafts_ratio}	n_{shafts}
1	1	16	44	947	3320,32	154.912.654
2	8	32	14	2.000	1107,93	51.691.569
3	27	109	15	5.300	1007,54	51.673.458
4	64	244	17	11.000	1212,81	56.585.019
5	125	462	20	23.900	1337,66	62.409.726
6	216	782	21	35.700	1370,23	63,929,322

Tabelle 4: Detaillierte Speicherauslastung und Anzahl der gefüllten Schäfte der globalen Linespace Datenstruktur anhand der Dragon-Testszenen

Bei der lokalen Linespace Datenstruktur wird zunächst die *mergedNodeScene* initialisiert (Tabelle 5). Da diese in den aufgelisteten Testszenen jeweils nur ein einzigartiges Objekt enthält, ist dessen Initialisierungszeit $t_{mnScene}$ konstant. Die Dauer $t_{obbScene}$ zur Erstellung der *obbScene* und t_{mScene} zur Erstellung der *materialScene* hängt von der Anzahl der Objekte o ab, denn für jedes muss je eine OBB und ein Material abgespeichert werden. Da bei der lokalen Datenstruktur, N-Tree und Linespace einmalig auf einem Objekt generiert und identische Objekte übersprungen werden, bleiben dessen Initialisierungszeiten und Speichergrößen konstant (Tabelle 6).

Dragon		Initialisierungszeiten (lokal) in s					
m	o	$t_{mnScene}$	$t_{obbScene}$	t_{mScene}	t_{ntree}	t_{ls}	t_{gesamt}
1	1	0,40	0,10	0,0001	7,71	0,96	13,97
2	8	0,41	0,76	0,0001	7,78	0,94	14,61
3	27	0,41	2,56	0,0001	7,87	0,95	16,51
4	64	0,40	6,06	0,0003	7,77	0,92	19,87
5	125	0,41	11,74	0,0004	7,70	0,99	25,59
6	216	0,41	20,37	0,0005	7,71	1,02	34,25
7	343	0,41	31,32	0,0008	7,47	0,96	44,84
8	512	0,40	45,13	0,0011	7,44	0,95	59,61
9	729	0,40	65,86	0,0014	7,43	0,94	79,29
10	1.000	0,40	91,78	0,0021	7,52	0,95	105,39

Tabelle 5: Detaillierte Initialisierungszeiten der lokalen Linespace Datenstruktur anhand der Dragon-Testszenen

Dragon		Speicherausl. (lokal) in MB			LS-Eigenschaften	
m	o	s_{ntree}	s_{ls}	s_{gesamt}	n_{shafts_ratio}	n_{shafts}
1	1	19	55	1.017	4178,86	194.968.698
2	8	19	55	1.017	4178,86	194.968.698
...	...	...	...	...	...	...
10	216	19	55	1.017	4178,86	194.968.698

Tabelle 6: Detaillierte Speicherauslastung und Anzahl der gefüllten Schäfte der lokalen Linespace Datenstruktur anhand der Dragon-Testszenen

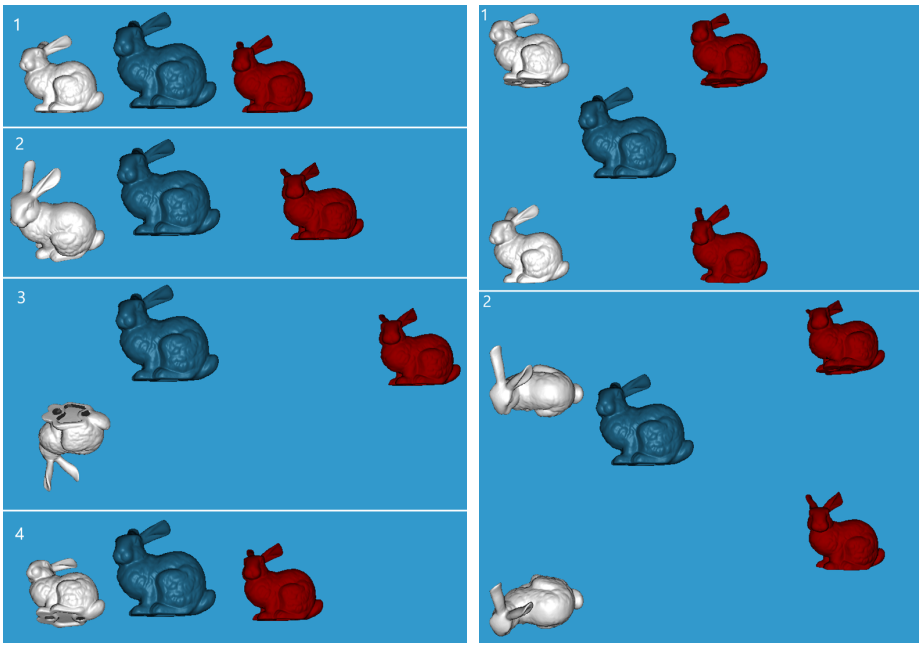
6.2 Dynamische Szenen

Bisweilen konnten dynamische Szenen mit der CPU-basierten Variante des globalen Linespace nicht dargestellt werden. Durch dessen Initialisierung zu Beginn des auszuführenden Programmes, haben Änderungen der Szenenstruktur, wie Transformationen von Objekten, zur Traversierungszeit keinen Einfluss.
Durch eine lokale Implementierung des Linespace ist dieser unabhängig von den Transformationen des Objektes, da die Schnittpunkttests in dem lokalen Koordinatensystem stattfinden. So kann eine geringe Anzahl an sich bewegenden Objekte z.T. in Echtzeit dargestellt werden.
In der Implementierung des lokalen Linespaces ist durch das Aufrufen der Methode *update(...)* aus Kapitel 5.2 eine Aktualisierung der Szenenknoten möglich, um dynamische Szenen zu rendern.
Zur Evaluation der Echtzeit-Fähigkeit der CPU-Implementation findet ein Vergleich zwischen dem globalen und lokalen Linespace ohne Szenenupdate, sowie dem lokalen Linespace mit Szenenupdate (dynamisch) statt. Tabelle 7 zeigt die Berechnungszeiten in Frames-Pro-Sekunde (FPS) des Renderings mit der jeweiligen genannten Datenstruktur an. Für die FPS aus Tabelle 7 wurde eine Mittelwert aus den ersten 20 Frames ermittelt.
In den Testszenen aus Abbildung 13 findet eine Rotation der weißen Objekte um ihre x-Achse statt und eine Translation des roten Objekte entlang der x-Achse. Das blaue Objekt transformiert sich nicht und dient als Kontrast zu den bewegten Objekten. Die jeweiligen ersten Bilder (Bild 1) beider Szenen zeigen die Objekte ohne Bewegung, wie sie auch durch den globalen Linespace oder dem lokalen Linespace ohne Szenenupdate zu sehen sind.

	FPS		
Testszene	Global	Lokal	Dynamisch
1	8,05	6,42	6,18
2	4,96	3,82	3,64

Tabelle 7: Berechnungszeiten der dynamischen Testszenen in FPS.

In Tabelle 7 ist zu sehen, dass der globale Linespace eine schnellere Berechnung aufweist als die Lokalen, da diese zunächst die OBBs traversieren müssen. Im Vergleich zwischen den statischen Varianten beträgt dies in Testszene 1 eine um 20,3% und in Testszene 2 eine um 23,0% schnellere Berechnungszeit. Der globalen Variante ist es jedoch nicht ermöglicht dynamische Szenen darzustellen. Die Verwendung des Szenenupdates in der lokalen Datenstruktur verringert die Berechnungszeit lediglich um 3,9%, bzw. 4,9%.

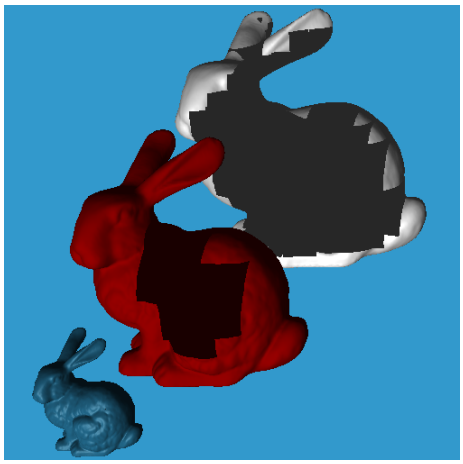


(a) Dynamische Testszene Nr.1 **(b)** Dynamische Testszene Nr.2

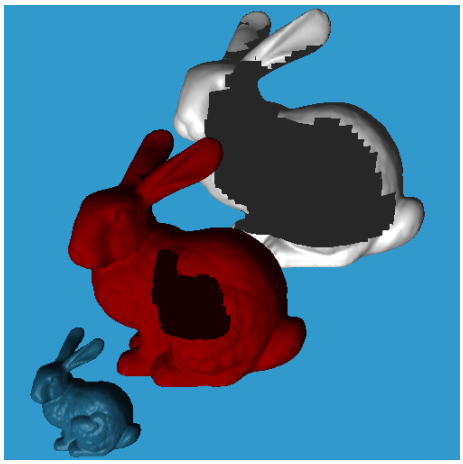
Abbildung 13: Einzelne Bilder einer Bildfolge zweier dynamischer Testszenen. Bild 1 beider Szenen zeigt die Objekte ohne Bewegung. Die weißen Objekte rotieren um ihre x-Achse, während die roten Objekte eine Translation entlang der x-Achse ausführen. Das blaue Objekt transformiert sich nicht.

6.3 Qualität

Die Abbildungen 14, 15, 16 und 17 zeigen die Resultate der Traversierung durch den globalen und lokalen Linespace mit unterschiedlich gewählten Auflösungen und Tiefen. Abbildung 14 und 15 betrachten Qualitätsunterschiede bei Schattenumrissen, Abbildung 16 und 17 bei auftretenden Artefakten.

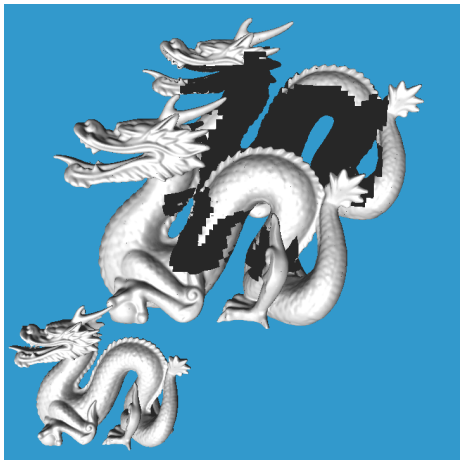


(a) Global (n=5, d=2)



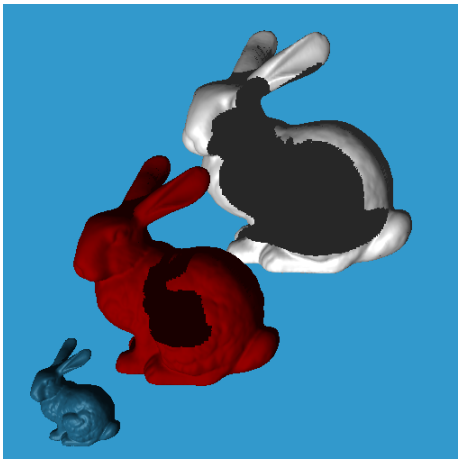
(b) Lokal (n=5, d=2)

(c) Global (n=5, d=2)

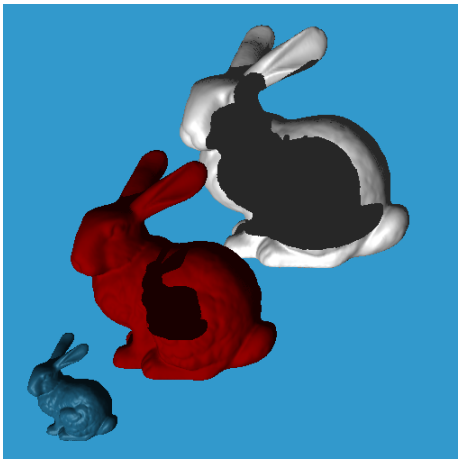


(d) Lokal (n=5, d=2)

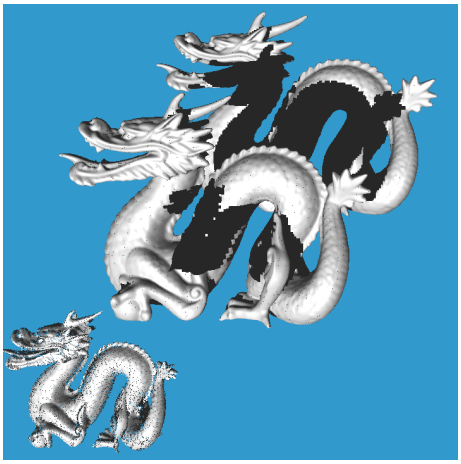
Abbildung 14: Die Schatten der globalen Variante weisen in refg:52global und 14c starke Kanten auf, die durch einen niedrigeren Detailgrad (n=5, d=2) stark zum Vorschein treten. Die Objekte lassen sich anhand der Silhouette nicht eindeutig zu identifizieren. Die Verwendung des lokalen Linespaces in 14b und 15d führt zu einer sichtbaren Verbesserung der Umrisse.



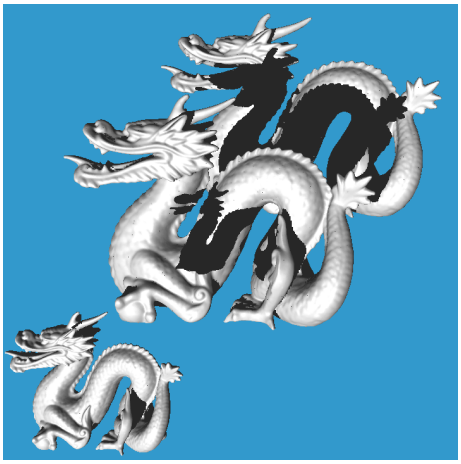
(a) Global (n=6, d=3)



(b) Lokal (n=6, d=3)



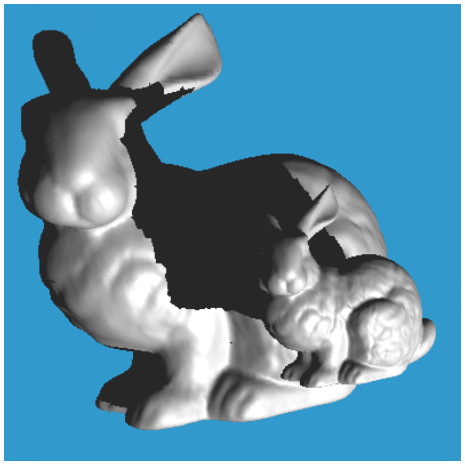
(c) Global (n=6, d=3)



(d) Lokal (n=6, d=3)

Abbildung 15: Wie in Abbildung 14 weisen die Schatten der globalen Variante in 15a und 15c Kanten auf, die jedoch durch eine höher gewählte Auflösung geringer ausfallen. Die Konturen des Schattens werden durch die Verwendung des lokalen Linespaces in 15b und 15d schärfer. In 15c sind, wie in 14c schwarze Punkte auf der Objektoberfläche zu sehen. Sie werden in Abbildung 17 genauer betrachtet.

(a) Global (n=5, d=2)

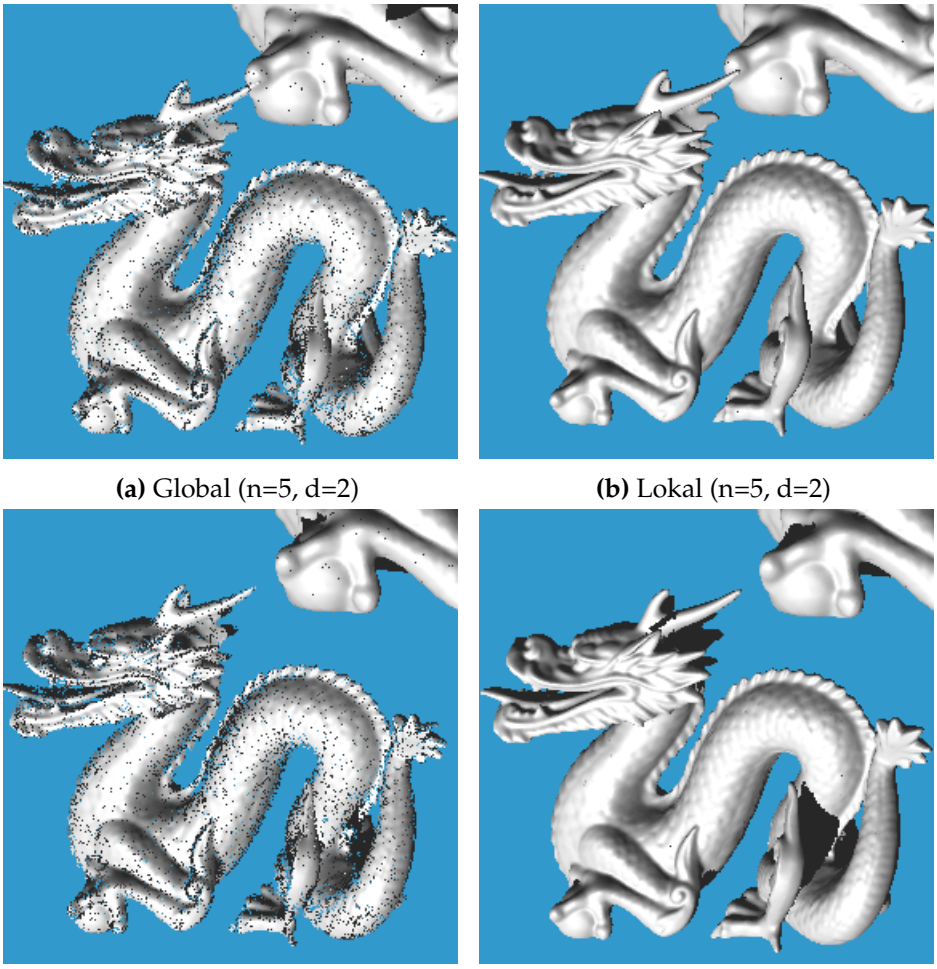


(b) Lokal (n=5, d=2)

(c) Global (n=6, d=3)

(d) Lokal (n=6, d=3)

Abbildung 16: Bei Traversierung durch die globale Linespace Datenstruktur entstehen bei der Abbildung von Schatten vereinzelte Artefakte. In 16a treten schwerwiegende Grafikfehler auf, die keine Schattenumrisse erkennen lassen. Zudem sind fehlerhafte Schattenberechnungen auch innerhalb des Objektes zu sehen. Dort fehlen Schatten, die durch die eigene Objekt-Geometrie entstehen müssten. Diese und weitere Artefakte, wie in Abbildung 16c, sind bei der Traversierung durch die lokale Datenstruktur nicht, bzw. weniger zu sehen. In 16b sind keine fehlerhaften Schattenberechnungen erkennbar. Die Artefakte aus 16c sind in 16b großteils verschwunden.



(a) Global (n=5, d=2)

(b) Lokal (n=5, d=2)

(c) Global (n=6, d=3)

(d) Lokal (n=6, d=3)

Abbildung 17: Die Ausschnitte der Dragon-Testszene aus Abbildung 14 und 15 verdeutlichen in 17a und 17c fehlerhaft dargestellte Pixel des globalen Linespace Verfahren. Diese entstehen bei kleinen Objekten, wenn in den Szenen größere Objekte existieren. Durch die niedrigere Auflösung in 17a und 17b werden vereinzelte Schatten nicht dargestellt, welche jedoch in 17c bei einer höheren Auflösung auch mit dem globalen Linespace fehlerhaft sind, jedoch nicht in 17d.

Die gesammelten Daten aus Kapitel 6.1 zeigen in Tabelle 4 und 6, dass die Speichergröße eines Linespaces von der Menge der getroffenen Dreiecke pro Shaft abhängt. Je größer n_{shafts_ratio} ist, desto mehr Informationen speichert der Linespace und desto mehr Details können dargestellt werden. Der globale Linespace verliert bei größer werdenden Szenen dadurch viele Informationen über getroffene Geometrie, da dieser alle existierenden Objekte enthält. Ein lokaler Linespace enthält nur die eigentliche Objektgeometrie, wodurch eine höhere Qualität gewährleistet wird.

7 Fazit

In dieser Arbeit wurde eine Linespace Datenstruktur in Kombination mit Orientated-Bounding-Boxen verwendet, um einen Linespace pro Objekt zu generieren und diesen lokal abrufen zu können. Dazu wird für jedes Objekt ein N-Tree in der Bounding Box und ein Linespace für jeden N-Tree Knoten erstellt. Die Schäfte geben nun Auskunft über existierende Geometrie, wodurch leere Bereiche schneller übersprungen werden können und die Knoten nicht tiefer traversiert werden müssen.

Die Evaluation der Datenstrukturen zeigt, dass die lokale Initialisierung des Linespaces innerhalb der Bounding Box bei gleicher Auflösung mehr Informationen über die Objektgeometrie enthält als der globale Linespace. Dies führt zu einem höheren Detailgrad, sodass weniger Artefakte und Schattenkanten zu sehen sind.
Neben der besseren Qualität, können nun auch dynamische Szenen auf der CPU teilweise in Echtzeit dargestellt werden.
Die lokale Implementation erreicht zudem eine Beschleunigung der Initialisierungszeiten von bis zu 94,13% gegenüber der globalen Variante. Bei steigender Zahl identischer Szenenobjekte wächst die Beschleunigungsrate mit. Die Speicherauslastung verringert sich in den Testszenen um bis zu 97,15% und steigt ebenfalls mit der Zahl identischer Objekte.

Zum Weiteren Testen der Datenstruktur könnte man Szenen verwenden, die mehrere verschiedene Objekte beinhalten und verschiedene Transformationen ausüben. Skalierungen sind momentan nur zum Verkleinern möglich. Falls es in einer Szene identische Objekte gibt, die jedoch unterschiedlich skaliert sind, wird der Linespace auf dem größten Objekt initialisiert. Damit wird dem Verschwinden von sichtbarer Geometrie bei einem vergrößerten Objekt entgegengewirkt. Wird ein Objekt nun dynamisch skaliert, und erreicht eine Größe die größer ist als die des Objektes auf dem der Linespace generiert wurde, greift der Raytracer auf falsche Daten zu und es kommen Artefakte zum Vorschein.

Um weitere Beschleunigungen der Berechnungszeiten zu erlangen, kann das Verfahren auf der GPU implementiert werden. Dort können durch parallelisierte Prozesse die Geschwindigkeiten, besonders die der Traversierungszeit, stark erhöht werden. Das echtzeitfähige Rendern von großen statischen Szenen wäre wahrscheinlich möglich. Bei Erfolg sollten auch größere dynamische Szenen gerendert werden können, da der Unterschied in den verwendeten Testszenen nur bis zu 4,9% beträgt und somit eher gering ausfällt. Es lässt sich vermuten, dass bei steigender Anzahl dynamischer Objekte diese Differenz mitwächst.

Zudem könnte eine Kombination aus Bounding-Volume-Hierarchie und Linespace stattfinden, um die Schnittpunkttests zwischen einem Strahl und Orientated-Bounding Boxen zu beschleunigen. Die Szene würde effizienter traversiert, leere Räume schneller übersprungen und die Berechnungszeiten verringert werden. Die Verwendung der Orientated-Bounding-Boxen als Bounding-Volume könnte beibehalten oder z.B. durch k-DOPs ersetzt werden.

Des Weiteren kann der Fotorealismus des Raytracers gesteigert werden, indem die Darstellung von direkter Beleuchtung auf globale Beleuchtung ausgeweitet wird. Keul *et al.* zeigen in [KKM17] bereits die Visualisierung von weichen Schatten mithilfe der globalen Linespace Datenstruktur. Ebenso kann der lokale Linespace zusammen mit einem Pathtracer [KMA+15] oder bidirektionalen Pathtracer [LW93] verwendet werden, um fotorealistische Effekte darzustellen. Die Informationen, die ein Schaft speichert, könnten erweitert werden. Informationen über einen durchschnittlichen Farbwert oder ein akkumulierter Helligkeitswert von Lichtquellen wäre eine Option.

Literatur

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

[CPP17] *Standard C++ Library Reference*, September 2017. http://de.cppreference.com/w/cpp/container/vector.

[JWCK08] Wenping Wang Jung-Woo Chang and Myung-Soo Kim. Efficient collision detection using a dual bounding volume hierarchy. In *Proceedings of the 5th International Conference on Advances in Geometric Modeling and Processing*, pages 143–154. GMP, 2008.

[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*, pages 269–278. SIGGRAPH, 1986.

[KKM16] Paul Lemke Kevin Keul and Stefan Müller. Accelerating spatial data structures in ray tracing through precomputed linespace visibility. In *Proceedings of the 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 17–26. WSCG, 2016.

[KKM17] Nicolas Klee Kevin Keul and Stefan Müller. Soft shadow computation using precomputed line space visibility information. In *Proceedings of the 25th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 97–106. WSCG, 2017.

[KMA+15] Markus Kettunen, Marco Manzi, Miika Aittala, Jaakko Lehtinen, Frédo Durand, and Matthias Zwicker. Gradient-domain path tracing. *ACM Trans. Graph.*, 34(4), 2015.

[LW93] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Proceedings of Compugraphics*, pages 145–153, 1993.

[Sza17] Gabor Szauer. *Game Physics Cookbook*. Packt Publishing Ltd., Birmingham, United Kingdom, 2017.

[VG97] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pages 65–76. SIGGRAPH, 1997.