UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Echtzeit Raytracing mittels GPU und der Linespace Datenstruktur

Bachelorarbeit

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

vorgelegt von
Alexander Seggebäing

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

Koblenz, im November 2017

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)

Danksagung

Ich möchte mich bei Prof. Dr. Stefan Müller für den regelmäßigen Gedankenaustausch und die hilfreiche Unterstützung während der Entstehungszeit dieser Bachelorarbeit bedanken.

Zusammenfassung

Diese Arbeit beschäftigt sich mit verschiedenen Ansätzen zur Beschleunigung von Raytracing-Berechnungen auf dem Grafikprozessor *(GPU)*. Dazu wird ein Voxelgrid verwendet, welches durch die Linespace-Datenstruktur erweitert wird. Der Linespace besteht aus richtungsbasierten Schäften *(Shafts)* und speichert die in ihm liegenden Objekte in einer Kandidatenliste. Es werden unterschiedliche Methoden zur Sortierung und Traversierung des Linespace vorgestellt und evaluiert. Die Methoden können keinen Anstieg der Bildfrequenz erreichen, ohne gleichzeitig in einer Verringerung der Bildqualität zu resultieren.

Abstract

This thesis explores different approaches for the acceleration of raytracing calculations on the graphics processing unit *(GPU)*. For that a voxel grid is used and extended by the linespace data structure. The linespace consists of direction based shafts and stores the objects located in those shafts in a candidate list. Different methods for the sorting and traversal of the linespace are presented and evaluated. The shown methods cannot provide a speed up of the frame rate without resulting in a loss of image quality.

Inhaltsverzeichnis

1 Einleitung **1**

2 Theoretische Grundlagen **2**
2.1 Raytracing 2
2.2 Datenstrukturen 3
2.2.1 Bounding-Volume-Hierarchie 3
2.2.2 Voxelgrid 4
2.2.3 N-Tree 4
2.3 Linespace 5
2.3.1 Aufbau 5
2.3.2 Initialisierung und Traversierung 6

3 Implementierung **8**
3.1 GPU Programmierung 8
3.1.1 Compute Shader 8
3.1.2 Shader Storage Buffer Objects 8
3.2 Linespace 9
3.2.1 Voxelisierung 9
3.2.2 Initialisierung 10
3.2.3 Sortierung der Kandidatenliste 12
3.3 Raytracer 15
3.3.1 Aufbau 15
3.3.2 Anpassung der Trace-Methode 16

4 Evaluation **18**
4.1 Testumgebung 18
4.1.1 Testsystem 18
4.1.2 Testaufbau 18
4.1.3 Bewertungskriterien 18
4.2 Initialisierung und Speicherbedarf 19
4.2.1 Sortierung nach Schnittpunkt 19
4.2.2 Sortierung nach orthogonaler Projektion 20
4.2.3 Sortierung nach gemittelter orthogonaler Projektion . 20
4.3 Performance 21
4.3.1 Erster Kandidat Methode 21
4.3.2 Erste n/k Kandidaten Methode 22
4.3.3 Erster Schnittpunkt Methode 23
4.4 Bildqualität 24
4.4.1 Erster Kandidat Methode 24
4.4.2 Erste n/k Kandidaten Methode 25
4.4.3 Erster Schnittpunkt Methode 27
4.5 Fazit 29

1 Einleitung

Seit Entstehung der Computergrafik stellt die Bildsynthese von virtuellen Szenen einen Kernaspekt des Bereichs dar. Durch die Einführung des Raytracing-Algorithmus durch Appel [App68] in den 60er Jahren wurde es erstmals möglich das physikalische Verhalten von Licht ansatzweise zu simulieren. Seitdem wurde das Verfahren stetig weiterentwickelt, um so immer realistischere Bilder zu erzeugen. Trotz der rasanten Entwicklung von neuer und immer leistungsfähigerer Hardware, ist das Rendern einer Szene durch Raytracing sehr zeitaufwändig, wodurch Raytracing in Echtzeit selbst auf modernen Systemen nur bedingt möglich ist. Aus diesem Grund wurde eine Vielzahl von verschiedenen Datenstrukturen zur Optimierung und Beschleunigung des Verfahrens entwickelt. Eine junge Datenstruktur ist der Linespace, welcher 2016 von Keul, Lemke und Müller [KLM16] vorgestellt wurde. Ziel des Linespace ist es einen Ansatz zu bieten, welcher echtzeitfähige Bildfrequenzen mit dem Raytracing-Algorithmus ermöglicht. Hauptmerkmal des Linespace sind richtungsbasierte Schäfte *(Shafts)*, welche Informationen über die in ihnen liegenden Objekte speichern.

In dieser Arbeit werden Verfahren erarbeitet, welche die richtungsbasierte Art des Linespace nutzen, um durch eine Sortierung der Informationen eine schnellere Traversierung zu ermöglichen. Dabei soll die Bildqualität der gerenderten Szenen weitestgehend erhalten bleiben. Es werden drei Methoden zur Sortierung und drei Methoden zur Traversierung des sortierten Linespace vorgestellt. Grundgerüst bietet eine Voxelgrid-Datenstruktur, welche durch den Linespace erweitert wird. Die Implementierung erfolgt durch Programmierung der Grafikkarte *GPU* mittels OpenGL.

In Abschnitt 2 werden zunächst die notwendigen theoretischen Grundlagen für diese Arbeit behandelt. Dabei handelt es sich um den Raytracing-Algorithmus, etablierte Datenstrukturen zur Beschleunigung und die Linespace-Datenstruktur. Abschnitt 3 befasst sich mit der Implementierung des Linespace und Raytracers auf der GPU und behandelt die verschiedenen Sortier- und Trace-Verfahren. Die Ergebnisse werden in Abschnitt 4 vorgestellt und anhand von fünf Testszenen ausgewertet. Ein Fazit schließt die Arbeit ab.

2 Theoretische Grundlagen

Im folgenden Kapitel werden die für diese Arbeit notwendigen Begrifflichkeiten und Grundlagen im Bereich des Raytracings und der Linespace-Datenstruktur erläutert.

2.1 Raytracing

Beim Raytracing handelt es sich um einen Rendering-Algorithmus, bei dem Strahlen im Raum verfolgt werden. Indem das physikalische Verhalten von Licht simuliert wird, lassen sich so realitätsnahe Bilder erzeugen. Unterschieden wird dabei zwischen zwei verschiedenen Ansätzen: Dem *Forward Ratracing*, bei dem die Strahlen ausgehend von der Lichtquelle verfolgt werden und dem *Backward Raytracing*, bei dem die Strahlen von der Kamera ausgehen. Obwohl das Forward-Raytracing-Prinzip intuitiver ist, da die Strahlen wie in der echten Welt von der Lichtquelle ausgehen, hat es gegenüber Backward Raytracing den Nachteil, dass es mit einem sehr hohen Zeitaufwand verbunden ist. Dies liegt daran, dass ein von der Lichtquelle ausgehender Strahl durch die Szenengeometrie in alle möglichen Richtungen reflektiert werden kann. Die Wahrscheinlichkeit, dass er die Kamera trifft, ist dabei sehr gering. Um ein vollständiges Bild zu erzeugen, müsste daher eine extrem hohe Anzahl an Lichtstrahlen verfolgt werden. In der Computergrafik wird aus diesem Grund in der Regel immer Backward Raytracing verwendet [Gla89].

Beim Backward Raytracing wird für jeden Pixel der Bildebene ein Strahl generiert. Diese von der Kamera ausgehenden Strahlen, auch Primärstrahlen genannt, werden mit der Szenengeometrie geschnitten. Im Falle mehrerer Schnittpunkte wird der erste Schnittpunkt, also der nächste Schnittpunkt zur Kamera bestimmt. Es wird ein weiterer Strahl von der Schnittstelle zur Lichtquelle erzeugt. Diese Strahlen werden Schattenstrahlen oder auch Schattenfühler genannt, da durch sie bestimmt wird, ob der Pixel durch eine Lichtquelle beleuchtet wird. Dazu wird der Schattenstrahl mit der Szenengeometrie geschnitten. Im Falle eines Schnittpunktes ist die Lichtquelle durch ein Objekt blockiert und der Pixel verschattet. Gibt es keinen Schnittpunkt, so wird der Pixel durch die Lichtquelle beleuchtet. Zusätzlich lässt sich der Raytracing-Algorithmus so erweitern, dass nicht nur diffuse Objekte sondern auch spiegelnde und lichtdurchlässige Materialien gerendert werden können. Dazu werden bei einem Schnittpunkt mit der Szenengeometrie weitere, sog. Sekundärstrahlen generiert und durch Rekursion bis zu einer festgelegten Tiefe weiterverfolgt.

Abbildung 1a verdeutlicht das Backward-Raytracing-Prinzip. Die von der Kamera ausgehenden Strahlen werden mit der Szenengeometrie geschnitten. Von den Schnittpunkten gehen Schattenstrahlen in Richtung der Lichtquelle aus. Der blau dargestellte Schattenstrahl wird durch kein Objekt blockiert, der entsprechende Pixel in der Bildebene wird daher beleuchtet. Der rote

Schattenstrahl wird durch die Kugel blockiert, der enstsprechende Pixel in der Bildebene ist also verschattet und wird nicht beleuchtet. Abbildung 1b zeigt das Forward-Raytracing-Prinzip. Die Strahlen gehen von der Lichtquelle aus und werden von den Objekten in der Szene reflektiert. Der blaue Strahl trifft die Kamera und der entsprechende Pixel wird beleuchtet. Der rote Strahl erreicht die Kamera nicht und wurde umsonst verfolgt.

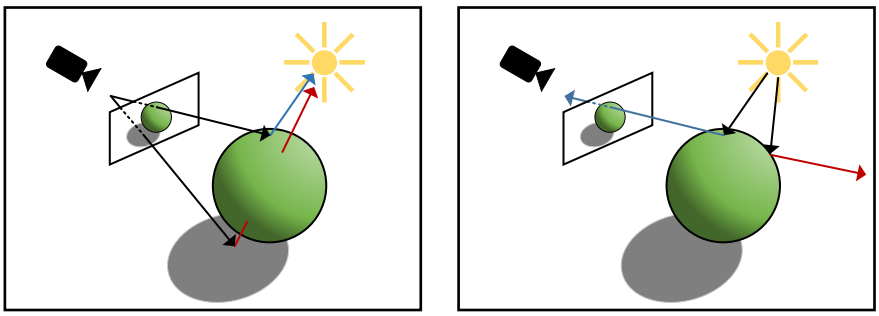

Abbildung 1: Das Raytracing-Prinzip. Die Strahlen werden als Pfeile symbolisiert.
a: Backward Raytracing
b: Forward Raytracing

2.2 Datenstrukturen

Ein großes Problem des Raytracings ist der hohe Zeitaufwand, welcher zum Rendern einer Szene erforderlich ist. Da für jeden Pixel in der Bildebene ein Strahl generiert und dessen vorderster Schnittpunkt mit der Szenengeometrie ermittelt werden muss, ist das Verfahren selbst auf modernen Systemen sehr zeitintensiv und nur bedingt zum Rendern in Echtzeit geeignet. Laut Whitted [Whi80] nimmt die Schnittpunktberechnung bei einfachen Szenen 75 Prozent der Rechenzeit in Anspruch. Bei komplexen Szenen steigt der Rechenaufwand auf über 95 Prozent. Um diesem Problem entgegenzuwirken, existieren verschiedene Datenstrukturen, welche die Szene aufteilen und beim Traversieren der Geometrie das Überspringen vieler Schnittpunktberechnungen erlauben.

2.2.1 Bounding-Volume-Hierarchie

Bei der Bounding-Volume-Hierarchie werden sog. Begrenzungsvolumen *(Bounding Volumes)* um die Objekte der Szene gelegt. Bei Bounding Volumes handelt es sich um einfache geometrische Körper bzw. Hüllen, welche ein Objekt möglichst eng umschließen. Mehrere Bounding Volumes können wiederum von einem Bounding Volume umschlossen werden, wodurch sich eine hierarchische Baumstruktur bildet [RW80]. Beim Raytracing wird der Strahl

zunächst mit den Bounding Volumes geschnitten. Liegt ein Schnittpunkt vor, werden die Bounding Volumes bzw. die Dreiecke innerhalb des Bounding Volumes geschnitten. Die Reihenfolge der Traversierung wird dabei von einer Priority-Queue bestimmt [KK86]. Abbildung 2 zeigt eine beispielhafte Bounding-Volume-Hierarchie.

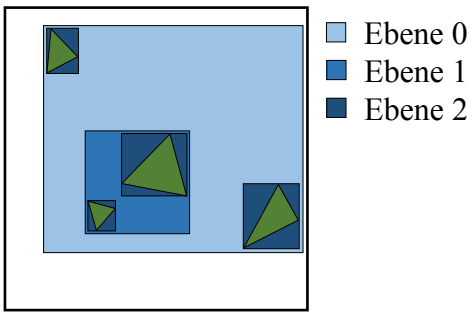

Abbildung 2: Bounding-Volume-Hierarchie

2.2.2 Voxelgrid

Die Voxelgrid-Datenstruktur teilt die gesamte Szene in Volumenelemente *(Voxel)* gleicher Größe auf. Jedem Voxel werden die in ihm liegenden Objekte zugeordnet. Beim Raytracing müssen nur die Objekte betrachtet werden, welche in Voxeln liegen die vom Strahl traversiert werden. Abbildung 3a zeigt eine beispielhafte Voxelgrid-Datenstruktur.

2.2.3 N-Tree

Beim N-Tree handelt es sich um eine hierarchische Baumstruktur, welche auf dem Voxelgrid aufbaut. Die Szene wird in N^3 Voxel aufgeteilt. Enthält einer dieser Voxel Szenengeometrie, so wird dieser wiederum aufgeteilt. Dies wird bis zu einer festgelegten maximalen Tiefe fortgeführt. Die Kandidaten werden in den Blattknoten gespeichert. Beim Raytracing traversiert der Strahl zunächst die Voxel auf der obersten Ebene. Hat ein Voxel Kinderknoten, so werden diese wiederum traversiert. Ist der Strahl bei einem Blattknoten angekommen, werden dessen Kandidaten auf Schnittpunkte getestet. Der N-Tree hat im Vergleich zum Voxelgrid den Vorteil, dass die Voxelauflösung dynamisch ist. Eine häufig verwendete Variante des N-Trees ist der Octree [Gla84] bei dem $N = 2$ gilt. Abbildung 3b zeigt eine beispielhafte N-Tree-Datenstruktur.

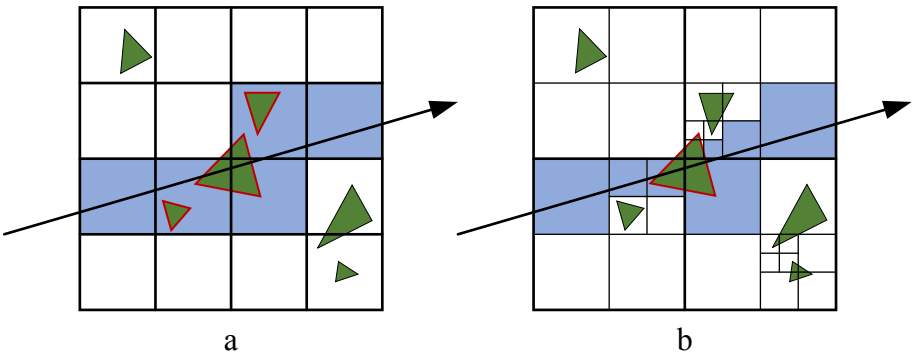

Abbildung 3: a: Voxelgrid-Datenstruktur
b: N-Tree-Datenstruktur
Der Strahl traversiert die blau gefärbten Voxel. Die rot umrandete Geometrie wird auf Schnittpunkte getestet.

2.3 Linespace

Der Linespace bietet die Möglichkeit eine vorhandene Datenstruktur zu erweitern, um so eine noch effizientere Traversierung der Szene zu ermöglichen. Dazu liefert er zusätzliche Entscheidungsvariablen, aufgrund welcher entschieden werden kann, ob Teile der Szenengeometrie bei der Strahlenverfolgung übersprungen werden können [KLM16].

2.3.1 Aufbau

Der Linespace teilt jedes Voxel bzw. jeden Knoten einer Datenstruktur in richtungsbasierte Schäfte *(Shafts)* auf. Dazu wird jede Seite des Bounding Volumes des Knotens in $N \times N$ Flächen *(Patches)* der selben Größe unterteilt. Insgesamt ergeben sich so $6 \times N^2$ Patches, welche je einen Index zugeordnet bekommen. Jedes Patch wird mit jedem anderem Patch zu einem Shaft verbunden, was eine Anzahl von $36 \times N^4$ Shafts pro Voxel ergibt.

Abbildung 4a zeigt einen Strahl im zweidimensionalen Raum und den dazu gehörigen Shaft. 4b visualisiert den zugehörigen Linespace. Abbildung 5a zeigt ein Objekt im Raum und alle dadurch gefüllten Shafts, 5b zeigt den zugehörigen Linespace. Gut erkennbar ist eine Symmetrie, welche dadurch gegeben ist, dass jeder Shaft für zwei Richtungen existiert; Es gilt: $LS(s; e) = LS(e; s)$. Der Speicherbedarf kann daher verringert werden, indem nur die Hälfte des Linespace gespeichert wird. Zusätzlich sind an der Diagonalen $N \times N$ große Blöcke sichtbar. Diese sind keine gültigen Shafts, da sich ihre Start- und Endflächen auf derselben Seite des Knotens befinden. Auch diese können bei der Speicherung vernachlässigt werden. Insgesamt reduziert sich dadurch die Anzahl an Shafts auf $15 \times N^4$.

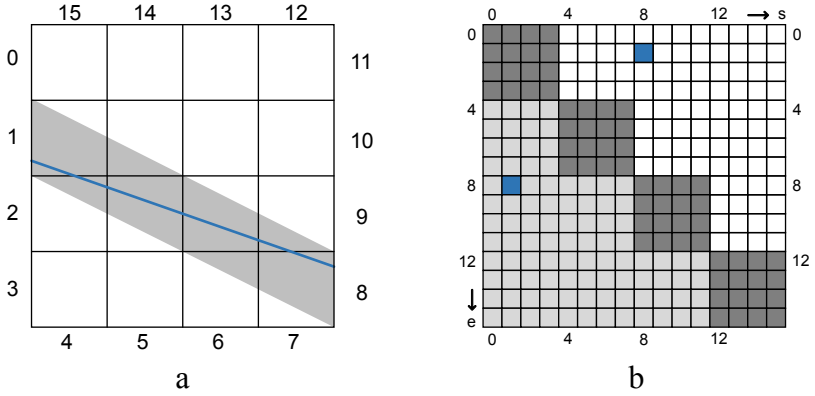

Abbildung 4: Linespace im 2D-Fall [KLM16]
a: Strahl (blau) und zugehöriger Shaft (grau) im 2D-Fall
b: Zugehöriger Linespace mit markiertem Shaft (blau)

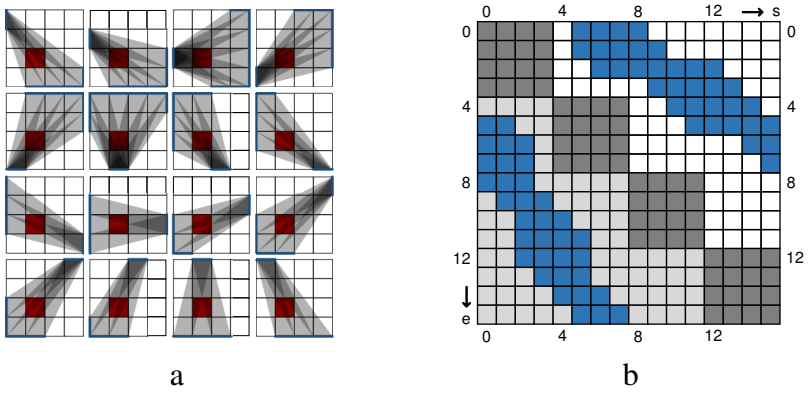

Abbildung 5: Linespace im 2D-Fall [KLM16]
a: Objekt (rot) und alle Shafts (grau) welche durch dieses gefüllt sind
b: Zugehöriger Linespace mit markierten Shafts (blau)

2.3.2 Initialisierung und Traversierung

Die Linespace-Datenstruktur muss einmalig für jeden Knoten initialisiert werden. Dabei können beliebige relevante Daten in den Shafts gespeichert werden. In der hier vorgestellten Implementation wird für jeden Shaft eine Kandidatenliste angelegt, welche die in dem Shaft liegenden Objekte speichert. Bei der Traversierung der Datenstruktur durch einen Strahl werden die durchdrungenen Start- und End-Patches bestimmt. Mithilfe der Patch-Indizes lassen sich der zugehörige Shaft und Linespace-Eintrag lokalisieren. Die in dem Shaft gespeicherten Daten, in diesem Fall eine Kandidatenliste, können dann ausgelesen werden.

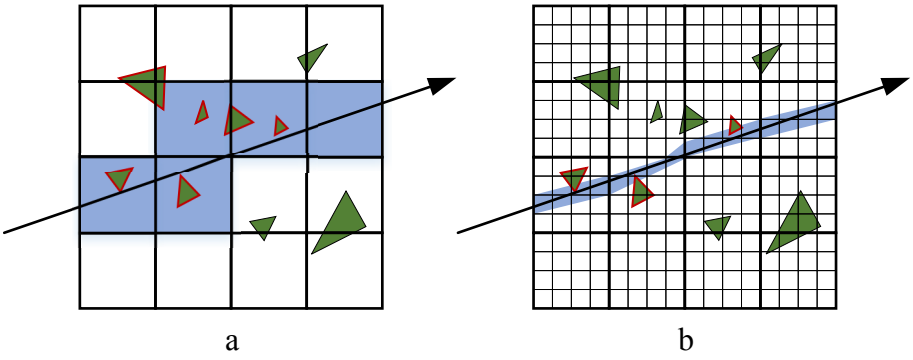

Abbildung 6: a: Voxelgrid
b: Erweiterung durch den Linespace
Der Strahl traversiert die blau gefärbten Voxel bzw. Shafts. Die rot umrandete Geometrie wird auf Schnittpunkte getestet.

Abbildung 6a zeigt die Traversierung einer Szene mithilfe der Voxelgrid-Datenstruktur. 6b zeigt die Traversierung derselben Szene, wobei die Voxelgrid-Datenstruktur durch den Linespace erweitert wurde. Erkennbar ist, dass der Strahl durch einen Shaft deutlich besser als durch ein Voxel eingegrenzt wird. Dadurch liegen auch weniger Kandidaten in ihm, welche auf Schnittpunkte getestet werden müssen. Eine bessere Eingrenzung durch das Voxelgrid ist zwar auch durch eine erhöhte Auflösung möglich, allerdings müsste dann auch eine weitaus größere Anzahl von Voxeln traversiert werden.

3 Implementierung

Der folgende Abschnitt befasst sich mit der Implementierung der Linespace-Datenstruktur und beschäftigt sich insbesondere mit der Sortierung der Kandidatenliste. Die Implementierung der Algorithmen wurde in C++ und GLSL umgesetzt.

3.1 GPU Programmierung

3.1.1 Compute Shader

Bei der Implementierung wurden *Compute Shader* zur Programmierung der GPU mit GLSL verwendet. Compute Shader sind kein Teil der OpenGL-Rendering-Pipeline, sondern Allzweck-Shader, welche für beliebige Berechnungen auf der GPU verwendet werden können. Dadurch dass keine spezifischen Eingabedaten definiert sind, kann genau festgelegt werden wie die zu berechnenden Daten aufgeteilt werden sollen. Diese Aufteilung kann ein-, zwei- oder dreidimensional sein. Die Daten werden dazu zunächst in globale Arbeitsgruppen *(global work groups)* aufgeteilt, welche wiederum in lokale Arbeitsgruppen *(local work groups)* aufgeteilt werden. Die globale Gruppengröße wird bei der Ausführung des Compute Shaders bestimmt:

```
glDispatchCompute(num_groups_x, num_groups_y, num_groups_z);
```

Die lokale Gruppengröße ist innerhalb des Shaders durch eine spezielle Layout-input-Angabe festgelegt:

```
layout(local_size_x = X, local_size_y = Y, local_size_z = Z) in;
```

Durch die Built-in-Variablen `gl_GlobalInvocationID` und `gl_LocalInvocationID` des Compute Shaders können die globalen und lokalen Indizes der aktuellen Shader-Einheit abgerufen werden. Durch die Parallelisierung der Linespace-Berechnungen auf der GPU, können diese deutlich schneller ausgeführt werden als bei einer Berechnung auf der CPU.

3.1.2 Shader Storage Buffer Objects

Shader Storage Buffer Objects (SSBOs) sind Buffer-Objekte, welche zur Speicherung und Abfrage von Daten innerhalb von Shader-Programmen benutzt werden können. Sie funktionieren ähnlich wie *Uniform Buffer Objects (UBOs)*, haben aber einige bedeutende Vorteile. Während sich die Größe von UBOs je nach GPU auf 16kB bis 64kB beschränkt, ist die Größe von SSBOs in der Praxis nur durch den verfügbaren Grafikspeicher begrenzt. Weiterhin sind SSBOs von Shader-Programmen beschreibbar und verfügen über einen

Satz von atomaren Operationen, welche sicheren Zugriff auf Variablen erlauben ohne Gefahr zu laufen, dass diese bereits von anderen Shader-Einheiten verwendet werden. SSBOs werden innerhalb des Shaders mithilfe eines speziellen Layout-Ausdrucks gebunden:

```
layout(std430, binding = x) buffer layoutName{...};
```

Durch ihre Größe sind SSBOs gut geeignet, um große Datenstrukturen wie den Linespace auf dem Speicher der Grafikkarte abzulegen.

3.2 Linespace

3.2.1 Voxelisierung

Als Grundstruktur des Linespace dient in der hier vorgestellten Implementierung ein Voxelgrid. Die Voxelisierung wird mithilfe einer orthographischen Projektion und konservativen Rasterisierung der Szene realisiert. Die Viewport-Größe entspricht dabei der Größe einer Seite des Voxelgrids. Die Primitive können dadurch anhand ihrer Positions- und Tiefenkoordinaten den Voxeln zugeordnet werden. Um Löcher im Voxelgrid zu vermeiden ist es notwendig, dass jedes Dreieck entlang derjenigen Achse projiziert wird, welche die größte Projektionsfläche und damit auch die meisten Fragmente bei der Rasterisierung erzeugt [CG12]. Die Projektionen werden kombiniert indem die X, Y, Z Koordinaten so vertauscht werden, dass die Projektionsrichtung für alle Primitive einheitlich ist. Da bei einer normalen Rasterisierung nur der Mittelpunkt jedes Pixels auf Abdeckung mit den Dreiecken getestet wird, können auch hier wieder Löcher im Voxelgrid entstehen. Aus diesem Grund wird eine konservative Rasterisierung [PF05] durchgeführt. Dazu werden die Dreieckspunkte so verschoben, dass ein etwas größeres Dreieck entsteht, welches für jeden angeschnittenen Pixel ein Fragment erzeugt. Die konservative Rasterisierung wird durch einen *Geometry Shader* durchgeführt.

Der Aufbau des Voxelgrid geschieht in zwei Phasen. Zunächst werden die Kandidaten jedes Voxels gezählt. Dies ist erforderlich um den Speicherbedarf der Kandidatenliste zu bestimmen und anschließend zu reservieren. Zusätzlich werden die Präfixsummen [Ble90] jedes Voxels mittels atomarer Addition berechnet und in einer 3D-Textur abgelegt. Abbildung 7 zeigt eine beispielhafte Berechnung der Präfixsummen. Mithilfe der Präfixsummen lässt sich für jedes Voxel der jeweils relevante Teil der Kandidatenliste lokalisieren und die Kandidatenanzahl bestimmen. Im zweiten Durchgang werden die Kandidaten dann als Kandidatenliste in einem SSBO gespeichert.

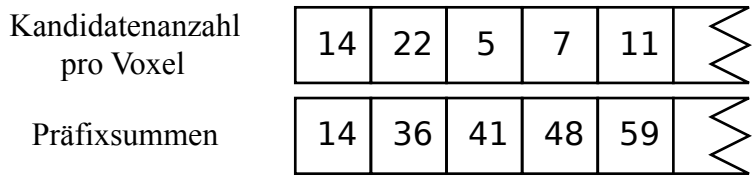

Abbildung 7: Berechnung der Präfixsummen

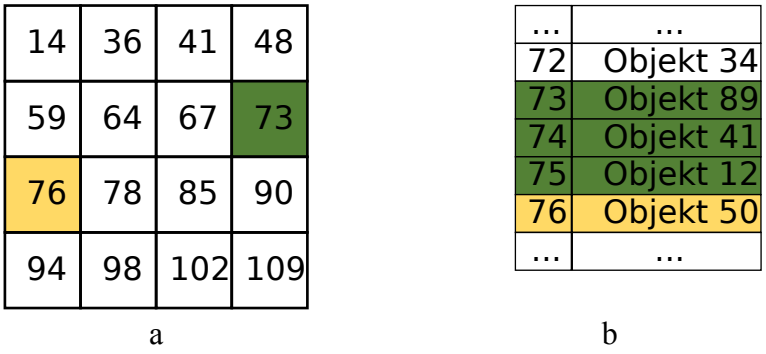

Abbildung 8: Speicherung des Voxelgrid im 2D-Fall
a: Voxelgrid Textur mit Präfixsummen
b: Kandidatenliste

Abbildung 8 zeigt den Zusammenhang von Präfixsummen und Kandidatenliste im 2D-Fall. Will man die Kandidaten des grün markierten Voxels in der Kandidatenliste abrufen liest man die Präfixsumme aus der Textur. Diese ist der Startindex des Voxels in der Kandidatenliste. Der Endindex lässt sich einfach bestimmen, indem die Präfixsumme des nächsten Voxels (gelb) aus der Textur gelesen wird.

3.2.2 Initialisierung

Die Linespace-Datenstruktur baut auf dem vorher initialisierten Voxelgrid auf und generiert für jedes nichtleere Voxel einen Linespace. Dabei wird ähnlich wie beim Voxelgrid in zwei Phasen vorgegangen. Zunächst werden alle nichtleeren Voxel gezählt und für jedes dieser Voxel ein Offset-Wert in Form einer Präfixsumme berechnet. Dabei hat jedes Voxel einen festgelegten Wert von $15 \times N^4$, da dies, wie in Abschnitt 2.3.1 gezeigt, die Anzahl der erforderlichen Shaft-Einträge pro Linespace ist. Die berechneten Präfixsummen werden in einem *VoxelToLSIndexMap* SSBO als *offset_LS* Array gespeichert.

Als nächstes werden die Kandidaten jedes Shafts gezählt und der für die Kandidatenliste erforderliche Speicher reserviert. Dazu werden die Shafts mit den Kandidaten ihrer zugehörigen Voxel geclippt. Als Clipping-Algorithmus kommt eine Erweiterung des Sutherland-Hodgman-Algorithmus [SH74] in

den 3D-Fall zum Einsatz. Die berechneten Werte werden in einem *lineSpaces* SSBO gespeichert. Der Index eines Shafts entspricht dabei der Summe aus dem vorher berechneten Voxel-Offset-Wert und der Shaft-ID, welche aus den Indizes der lokalen Arbeitsgruppe des Compute Shaders berechnet wird.

Jetzt lassen sich die Präfixsummen der Shafts pro Linespace für die Kandidatenliste berechnen. Dazu wird eine atomare Addition durchgeführt. Die Präfixsummen werden im *lineSpaces* SSBO gespeichert und überschreiben die alten Werte. Zusätzlich werden die Präfixsummen, der zu den Voxeln gehörigen Linespaces, für die Kandidatenliste berechnet und als *offset_ LSCandList* Array im *VoxelToLSIndexMap* SSBO gespeichert. Der Index entspricht dabei der Voxel-ID, welche durch die Indizes der globalen Arbeitsgruppe des Compute Shaders berechnet wird.

Im zweiten Durchgang werden die Kandidaten dann in einem SSBO als Kandidatenliste gespeichert. Der Startindex eines Shafts in der Kandidatenliste entspricht dabei, der Präfixsumme des zugehörigen Linespace addiert mit der Präfixsumme des Shafts:

```
VoxelToLSIndexMap.offset_LSCandList[VoxelID] +
lineSpaces[VoxelToLSIndexMap.offset_LS[VoxelID] + ShaftID]
```

Abbildung 9 zeigt den Zusammenhang der Buffer beispielhaft im 2D-Fall. Zusammenhängende Einträge sind dabei farblich gekennzeichnet. Das im Voxelgrid grün markierte Voxel hat einen Offset-Wert von $15 \times N^4$. Dieser Wert ist im *offset_ LS* Array unter der Voxel-ID abgelegt. Die Präfixsummen der Shafts sind pro Linespace im *lineSpaces* SSBO gespeichert. Im Falle des grünen Voxels entspricht der Startindex im *lineSpaces* Array seinem Offset-Wert von $15 \times N^4$. Zusätzlich ist die Präfixsumme, des zum grünen Voxel gehörigen Linespace, im *offset_ LSCandList* Array unter der Voxel-ID abgelegt. Die Objekte jedes Shafts werden durch die Summe aus Linespace-Präfixsumme und Shaft-Präfixsumme bestimmt. Der hellgrün markierte Shaft beinhaltet also die Kandidaten 89 und 17.

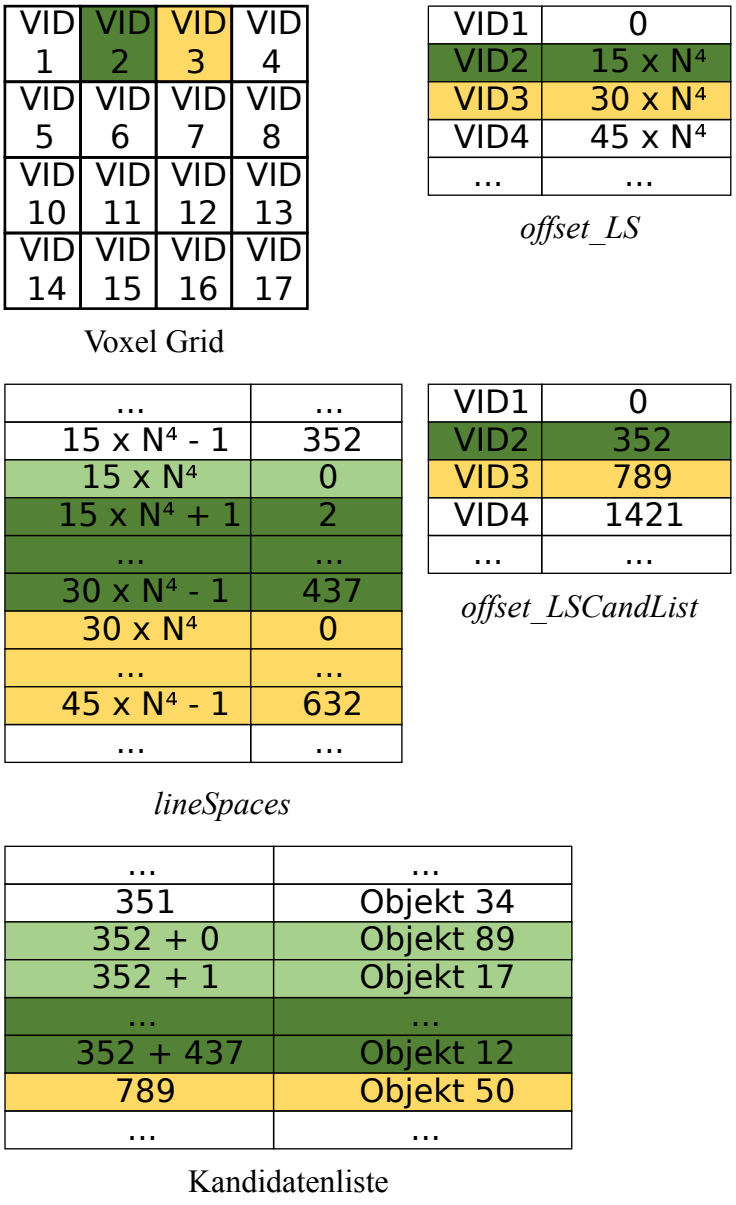

Abbildung 9: Speicherung des Linespace im 2D-Fall
VID: Voxel-ID

3.2.3 Sortierung der Kandidatenliste

Da die Linespace-Datenstruktur aus richtungsbasierten Shafts besteht, ist es möglich die Kandidaten anhand ihrer Position im Shaft zu sortieren. Beim Raytracing können dann verschiedene Strategien verwendet werden, welche anhand der Sortierung nur Teile der Kandidatenliste auf Schnittpunkte testen. Ziel ist es durch eine möglichst optimale Sortierung und ein entsprechen-

des Raytracing-Verfahren eine erhöhte Bildfrequenz bei minimaler Verringerung der Bildqualität zu erreichen. Zur Sortierung der Kandidatenliste ist ein zusätzlicher SSBO notwendig, welcher Distanzwerte speichert, anhand derer die Kandidatenliste sortiert werden kann. Die Indizes der Distanzwerte entsprechen dabei den Indizes der zugehörigen Kandidaten in der Kandidatenliste. Die Sortierung wird während der Initialisierung des Linespace durchgeführt.

Es wurden die folgenden drei Sortiermethoden implementiert:

Methode 1: Sortierung nach Schnittpunkt
Bei der Sortierung nach Schnittpunkt wird jeder Kandidat mit dem Mittelstrahl des Shafts geschnitten. Dabei sind auch Schnittpunkte mit der Dreiecksebene gültig, welche außerhalb der Dreiecksfläche liegen. Die Entfernung des Schnittpunktes zum Ursprung des Strahls wird gemessen. Diese Distanzberechnung wird für die vier Eckstrahlen des Shafts wiederholt. Das arithmetische Mittel der fünf berechneten Werte wird bestimmt und in der Distanzliste gespeichert. Die Kanidatenliste wird dann mit Selectionsort anhand der Distanzliste absteigend sortiert. Abbildung 10 stellt die Distanzberechnung eines Kandidaten mit dem Mittelstrahl dar. Die rot dargestellte Strecke ist der Distanzwert, welcher mit den vier Distanzwerten der Eckstrahlen gemittelt und in die Distanzliste eingetragen wird.

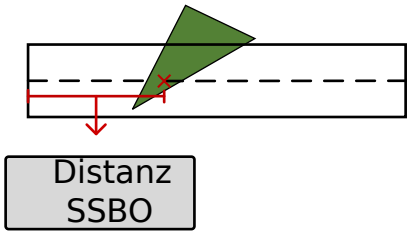

Abbildung 10: Sortierung nach Schnittpunkt

Methode 2: Sortierung nach orthogonaler Projektion
Bei der Sortierung nach orthogonaler Projektion wird jeder Kandidat zunächst mit dem Shaft geclippt. Die Punkte des resultierenden Polygons werden orthogonal auf den Mittelstrahl des Shafts projiziert und ihre Entfernung zum Ursprung des Strahls wird gemessen. Der niedrigste Wert wird als Distanzwert des Kandidaten festgelegt. Die Distanzberechnung wird für die vier Eckstrahlen des Shafts wiederholt und das arithmetische Mittel der fünf Werte in der Distanzliste gespeichert.

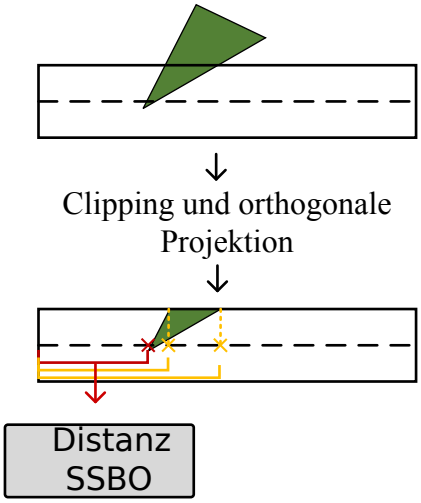

Abbildung 11: Sortierung nach orthogonaler Projektion

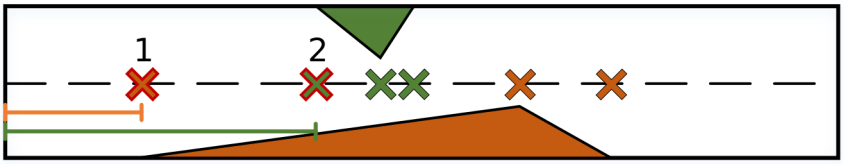

Abbildung 12: Problem bei der Sortierung nach orthogonaler Projektion

Abbildung 11 zeigt die Distanzberechnung anhand orthogonaler Projektion auf den Mittelstrahl. Die rot dargestellte Strecke ist die kürzeste und wird als Distanzwert festgelegt. Ein Nachteil dieser Methode ist, dass sie je nach Blickrichtung verschiedene Distanzwerte ermittelt. Abbildung 12 visualisiert das Problem. Die rot umrandeten Kreuze stellen die Punkte dar, welche bei einer Betrachtung von vorne die kürzeste Distanz zum Ursprung des Mittelstrahls haben. In der sortierten Kandidatenliste liegt das orangene Dreieck daher vor dem grünen Dreieck. Betrachtet man den Shaft aber von hinten, so reicht es nicht die Kandidatenliste auch von hinten zu betrachten, da dann das grüne Dreieck vor dem orangenen Dreieck liegen würde. Tatsächlich liegt aber das orangene Dreieck, auch bei einer Betrachtung von hinten, vor dem grünen. Aus diesem Grund müssen zusätzlich eine zweite Distanz- und eine zweite Kandidatenliste für die Betrachtung der Shafts von hinten erstellt werden. Der durch die Sortierung erforderliche Speicherplatz wird also verdreifacht. Beide Kandidatenlisten werden mit Selectionsort anhand der zugehörigen Distanzliste absteigend sortiert.

Methode 3: Sortierung nach gemittelter orthogonaler Projektion

Die Sortierung nach gemittelter orthogonaler Projektion funktioniert grundsätzlich wie die Sortierung nach orthogonaler Projektion. Anstatt die kürzeste Entfernung der projizierten Punkte als Distanzwert festzulegen, wird aber das arithmetische Mittel der Distanzwerte aller Projektionen berechnet und in der Distanzliste gespeichert. Dadurch dass der Distanzwert gemittelt wird, sind die zusätzlichen Kandidaten- und Distanzlisten nicht mehr notwendig.

Abbildung 13 zeigt die Distanzberechnung mittels gemittelter orthogonaler Projektion auf den Mittelstrahl. Das Objekt wird mit dem Shaft geclippt und die entstehenden Punkte werden auf den Mittelstrahl projiziert. Die Entfernung der Punkte zum Ursprung des Strahls wird gemessen und gemittelt in der Distanzliste abgelegt.

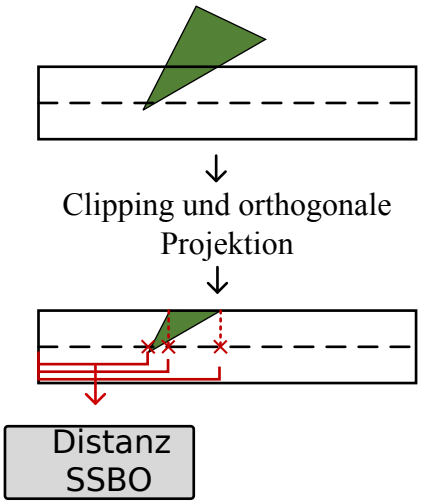

Abbildung 13: Sortierung nach gemittelter orthogonaler Projektion

3.3 Raytracer

3.3.1 Aufbau

Der Raytracer besteht aus den zwei Hauptkomponenten *Trace* und *Shade*. Zunächst wird für jeden Pixel des Bildes ein Strahl generiert und die Methode Trace aufgerufen. Diese verfolgt und testet den Strahl auf Schnittpunkte mit der Szenengeometrie und bestimmt im Fall mehrerer Schnittpunkte den vordersten Schnittpunkt. Dazu wird zunächst das Voxelgrid mittels des 3D-Digital Differential Analyzer *(3D-DDA)* [AW87] Algorithmus traversiert. Dieser bestimmt zu Beginn das Voxel durch welches der Strahl in das Voxelgrid eintritt. Die darauf folgenden Voxel werden anhand der Austrittpunkte des vorherigen Voxels bestimmt. Für jedes durch den Strahl traversierte Vo-

xel wird anhand des Eintritts- und Austrittspunktes, der zugehörige Shaft bestimmt. An dieser Stelle lässt sich auch bestimmen, ob der Shaft von vorne oder von hinten traversiert wird. Der Strahl wird dann auf Schnittpunkte mit den Kandidaten des Shafts getestet. Wird ein Schnittpunkt gefunden, wird die Methode Shade aufgerufen, ansonsten wird eine vorher definierte Hintergrundfarbe als Farbe des Pixels festgelegt.

Aufgabe der Shade-Methode ist es den Pixel zu beleuchten. Hierzu iteriert der Raytracer über alle Lichtquellen der Szene und generiert die zugehörigen Schattenstrahlen, welche dann auf Schnittpunkte mit der Szenengeometrie getestet werden. Die Linespace-Datenstruktur wird dabei wie bei der Trace-Methode traversiert, es wird aber nur ein beliebiger Schnittpunkt und nicht der vorderste Schnittpunkt gesucht. Ist kein Schnittpunkt vorhanden, die Lichtquelle also nicht blockiert, wird der entsprechende Lichtwert mit der Materialfarbe des Objektes multipliziert und die resultierende Farbe als Wert des Pixels festgelegt.

3.3.2 Anpassung der Trace-Methode

Damit der Raytracer einen Vorteil aus der sortierten Kandidatenliste ziehen kann, muss die Trace-Methode entsprechend angepasst werden. Im Folgenden werden drei Vorgehensweisen erläutert. Alle Methoden haben gemeinsam, dass für jeden Shaft unterschieden werden muss, ob dieser vom Strahl von vorne oder von hinten traversiert wird. Dies liegt daran, dass der Linespace wie in Abschnitt 2.3.1 gezeigt, aus Speichergründen jeden Shaft nur von einer Richtung speichert. Wird der Shaft von hinten traversiert muss daher je nach Sortiermethode entweder die Kandidatenliste von hinten betrachtet, oder die zweite Kandidatenliste verwendet werden.

Methode 1: Erster Kandidat
Bei der *Erster Kandidat* Methode wird beim Ausführen der Trace- und Shade-Methoden jeweils nur der erste Kandidat jedes Shafts betrachtet.

Methode 2: Erste n/k Kandidaten
Die *Erste n/k Kandidaten* Methode betrachtet bei einer Kandidatenliste mit n Einträgen nur die ersten n/k Kandidaten. Gilt $n/k < 1$ so wird nur der erste Kandidat betrachtet.

Methode 3: Erster Schnittpunkt
Bei der *Erster Schnittpunkt* Methode wird die vollständige Kandidatenliste betrachtet, die Trace-Methode sucht aber nicht den vordersten Schnittpunkt, sondern stoppt die Schnittpunktsuche, sobald ein Schnittpunkt gefunden wurde. Die Shade-Methode ändert sich nicht, da die Schattenfühler bereits nach diesem Prinzip arbeiten.

Ebenenbetrachtung der Kandidaten
Die Trace-Methoden 2 und 3 lassen sich alternativ so anpassen, dass die Schnittpunktberechnungen anhand der Dreiecksebenen der Kandidaten durchgeführt werden. Der Vorteil dieser Variante ist, dass durch die erzwungenen Schnittpunkte keine Löcher im gerenderten Bild entstehen können, es sei denn, der Strahl liegt parallel zur Ebene. Ein Nachteil ist jedoch, dass die Linespace-Datenstruktur in Form von Blockartefakten zum Vorschein kommt. Methode 3 kann in der gleichen Art modifiziert werden, die Funktionsweise ist dann aber dieselbe wie bei Trace-Methode 1, da der erste Kandidat immer in einem Schnittpunkt resultieren würde.

4 Evaluation

Der folgende Teil befasst sich mit der Evaluation der in den vorigen Kapiteln genannten Methoden zur Kandidatensortierung und Traversierung des Linespace.

4.1 Testumgebung

4.1.1 Testsystem

Alle Tests wurden auf einem Rechner mit Intel Core i7 6700 CPU, 32 GB Arbeitsspeicher und Nvidia GTX Titan X GPU ausgeführt. Der Prozessor verfügt über 4 Kerne mit einer maximalen Taktfrequenz von 4 GHz. Die Grafikkarte ist mit 3072 CUDA Kernen, mit einer maximalen Taktfrequenz von 1075 MHz und 12 GB GDDR5 Grafikspeicher ausgestattet. Getestet wurde auf Windows 10 Pro 64bit mit Nvidia Treiberversion 376.53.

4.1.2 Testaufbau

Jede Kombination aus Sortier- und Trace-Verfahren wird anhand von fünf ausgewählten Szenen, unterschiedlicher Komplexität, evaluiert. Die Linespace-Auflösung liegt bei allen Szenen bei $N = 5$. Die Auflösung des Voxelgrids ist für jede Szene individuell festgelegt, sodass die Performance bei Erweiterung durch das Linespace-Verfahren mit unsortierter Kandidatenliste möglichst optimal ist. Die Render-Auflösung beträgt 850×850.

Es wurden die folgenden fünf Testszenen gewählt:

Cornell Box: 60 Eckpunkte, 34 Dreiecke.

Teapot: 3872 Eckpunkte, 6402 Dreiecke.

Bunny: 35947 Eckpunkte, 69451 Dreiecke.

Dragon: 566098 Eckpunkte, 1132830 Dreiecke.

Sponza: 145185 Eckpunkte, 262267 Dreiecke.

4.1.3 Bewertungskriterien

Die Bewertung der Algorithmen erfolgt anhand der folgenden drei Kriterien:

Initialisierung und Speicherbedarf
Die Initialisierungszeit der Datenstruktur wird gemessen. Dazu zählt zum einen die Generierung des Voxelgrids und zum anderen die Initialisierung des Linespace einschließlich der Sortierung der Kandidatenliste. Zusätzlich wird der erforderliche Speicher auf der GPU gemessen. Hierzu zählen alle

für die Datenstruktur erforderlichen Buffer. Da die genutzte Trace-Methode keinen Einfluss auf die Generierungszeit und den Speicherverbrauch hat, wird hierbei nur zwischen den drei verschiedenen Sortiermethoden unterschieden.

Bildqualität
Die Qualität der gerenderten Szene wird bewertet. Als Vergleichswert dient das unsortierte Linespace-Verfahren, welches ein fehlerfreies Bild erzeugt. Besonders an Stellen mit vielen Dreiecken ist es wichtig, dass die richtigen Kandidaten beleuchtet werden. Falls Bildartefakte entstehen, sollen diese möglichst unauffällig sein.

Performance
Um die Echtzeitfähigkeit der implementierten Sortier- und Trace-Algorithmen zu bewerten wird die Bildfrequenz in Bildern pro Sekunde *(fps)* gemessen. Ziel ist es eine höhere Bildfrequenz als das unsortierte Linespace-Verfahren zu erreichen.

4.2 Initialisierung und Speicherbedarf

Tabelle 1 zeigt die Generierungsdauer und den Speicherverbrauch der unsortierten Linespace-Datenstruktur. Die Tabellen 2 bis 4 zeigen die Generierungsdauer und den Speicherverbrauch der verschiedenen Sortierverfahren. Zusätzlich bilden sie die Änderungsrate zum unsortierten Linespace-Verfahren in Prozent ab. Klar erkennbar ist, dass sowohl Generierungsdauer als auch Speicherverbrauch grundsätzlich von der Komplexität der Szene und den zugehörigen Voxelgrid- und Linespace-Auflösungen abhängen. So braucht die Dragon Szene mit einer Initialisierungsdauer von etwa 1,7 min deutlich länger als die Teapot Szene mit nur 1,5 s. Entsprechend liegt auch der Speicherverbrauch bei der Dragon Szene mit 4,3 GB stark über dem der Teapot Szene mit nur 199 MB.

	Generierungsdauer (∼ms)	Speicherverbrauch (MB)
Cornell Box	40	4
Teapot	1549	199
Bunny	10862	682
Dragon	99551	4268
Sponza	39157	2443

Tabelle 1: Linespace Initialisierung

4.2.1 Sortierung nach Schnittpunkt

Wie an Tabelle 2 erkennbar, ist die Generierungsdauer der Linespace-Datenstruktur mit einer Sortierung der Kandidatenliste nach Schnittpunkt in allen

Szenen, mit Ausnahme der Cornell Box Szene, $30-60\%$ höher als bei der Berechnung des unsortierten Linespace. Grund hierfür sind die zusätzlich erforderlichen Schnittpunkttests. Hinzu kommt das Sortieren der Kandidatenliste mittels Selectionsort, welches ebenfalls zusätzliche Zeit in Anspruch nimmt. Der Speicherverbrauch steigt ebenfalls deutlich mit $19-67\%$ je nach Szene. Dies hängt mit dem benötigten Distanz SSBO zusammen, welcher den gleichen Speicherbedarf wie die Kandidatenliste besitzt.

	Generierungsdauer (∼ms)		Speicherverbrauch (MB)	
Cornell Box	40	+0%	5	+19%
Teapot	2051	+32%	290	+46%
Bunny	16151	+49%	1111	+63%
Dragon	155532	+56%	7451	+75%
Sponza	62260	+60%	4070	+67%

Tabelle 2: Linespace-Initialisierung mit Sortierung nach Schnittpunkt

4.2.2 Sortierung nach orthogonaler Projektion

Tabelle 3 zeigt Initialisierungsdauer und Speicherverbrauch beim Sortieren der Kandidatenliste nach orthogonaler Projektion. Im Vergleich zur Sortierung nach Schnittpunkt (Tabelle 2) ist ein höherer Anstieg der Generierungsdauer und des Speicherverbrauchs erkennbar. Die Generierungsdauer bei den komplexeren Dragon und Sponza Szenen steigt um 85% bzw. 90%. Auch in der weniger komplexen Teapot Szene steigt die Generierungsdauer mit 53% stark. Noch deutlicher ist der Unterschied beim benötigten Speicherplatz. Dort liegt bei der Dragon Szene ein Anstieg von 224% vor. Dies kommt durch das in Abschnitt 3.2.3 erläuterte Problem zustande, welches eine zusätzliche Kandidaten- und Distanzliste erfordert. In Hinblick auf die Initialisierung ist diese Sortiermethode daher die zeit- und ressourcenaufwendigste.

	Generierungsdauer (∼ms)		Speicherverbrauch (MB)	
Cornell Box	46	+15%	6	+56%
Teapot	2375	+53%	472	+137%
Bunny	19015	+75%	1968	+189%
Dragon	184210	+85%	13817	+224%
Sponza	74775	+90%	7326	+200%

Tabelle 3: Linespace-Initialisierung mit Sortierung nach orthogonaler Projektion

4.2.3 Sortierung nach gemittelter orthogonaler Projektion

Tabelle 4 zeigt die Generierungszeit und den Speicherverbrauch bei Sortierung des Linespace nach gemittelter orthogonaler Projektion. Verglichen

mit den Tabellen 2 und 3 ist erkennbar, dass die Initialisierungsdauer dieser Sortiermethode zwischen den beiden anderen vorgestellten Verfahren liegt. Da sowohl hier als auch bei einer Sortierung nach Schnittpunkt nur eine Kandidaten- und eine Distanzliste erforderlich sind, ist der Speicherverbrauch bei beiden Verfahren identisch.

	Generierungsdauer (∼ms)		Speicherverbrauch (MB)	
Cornell Box	44	+10%	5	+19%
Teapot	2256	+45%	290	+46%
Bunny	17482	+61%	1111	+63%
Dragon	169735	+71%	7451	+75%
Sponza	67605	+73%	4070	+67%

Tabelle 4: Linespace-Initialisierung mit Sortierung nach gemittelter orthogonaler Projektion

4.3 Performance

Tabelle 5 zeigt die Bildfrequenz mit der unsortierten Linespace-Datenstruktur in fps. Die Tabellen 6 bis 10 zeigen die Bildfrequenz der verschiedenen Trace-Verfahren in Verbindung mit den drei implementierten Sortiermethoden. Die Änderungsrate im Vergleich zum unsortierten Linespace ist in Prozent angegeben.

	Bildfrequenz (fps)
Cornell Box	1000
Teapot	272
Bunny	81
Dragon	21
Sponza	45

Tabelle 5: Bildfrequenz mit unsortiertem Linespace-Verfahren

4.3.1 Erster Kandidat Methode

Die *Erster Kandidat* Trace-Methode erzielt in Kombination mit allen Sortierverfahren, eine hohe Steigerung der Bildfrequenz. Dabei ist deutlich, dass die Methode bei steigender Objektanzahl, effektiver wird. So steigt die Bildfrequenz bei der Teapot Szene um $20\% - 24\%$, während bei der Dragon Szene eine Steigerung von $210\% - 224\%$ vorliegt. Eine Ausnahme stellt die Cornell Box Szene dar, welche trotz einer geringen Zahl an Objekten eine Steigerung von 50% erfährt. Grund dafür ist die gering gewählte Voxelgrid-Auflösung, durch die ein Shaft trotz geringer Objektdichte mehrere Kandidaten enthält.

Das Verfahren ist in Kombination mit Sortiermethode 3 am effektivsten, während Sortiermethoden 1 und 2 etwas langsamere Ergebnisse erzielen.

fps	Methode 1	Methode 2	Methode 3
Cornell Box	1500 +50%	1500 +50%	1500 +50%
Teapot	329 +21%	326 +20%	336 +24%
Bunny	139 +72%	138 +70%	143 +77%
Dragon	66 +214%	65 +210%	68 +224%
Sponza	127 +182%	130 +189%	129 +187%

Tabelle 6: Bildfrequenz mit der *Erster Kandidat* Trace-Methode

Schnittpunktberechnung mit Dreiecksebene
Bei einer Ebenenbetrachtung der Kandidaten, steigt die Bildfrequenz in allen Testszenen mit Ausnahme der Cornell Box Szene noch stärker. Grund für die höhere Bildfrequenz ist, dass mehr Strahlen auf Geometrie treffen und dadurch weniger Linespaces bzw. Shafts traversieren müssen. Da die Cornell Box Szene durch die niedrige Voxelgrid-Auflösung und Objektdichte sehr schnell traversiert werden kann, ergibt sich in diesem Fall kein Vorteil. Die gewählte Sortiermethode hat keine Auswirkung auf die Performance.

fps	Methode 1	Methode 2	Methode 3
Cornell Box	1500 +50%	1500 +50%	1500 +50%
Teapot	368 +35%	368 +35%	369 +36%
Bunny	148 +83%	149 +84%	149 +84%
Dragon	69 +229%	69 +229%	69 +229%
Sponza	219 +387%	221 +390%	220 +390%

Tabelle 7: Bildfrequenz mit der *Erster Kandidat* Trace-Methode und Betrachtung der Kandidaten als Ebenen

4.3.2 Erste n/k Kandidaten Methode

Tabelle 8 zeigt die Bildfrequenz der *Erste n/k Kandidaten* Trace-Methode für $k = 2$. Im Vergleich zur *Erster Kandidat* Trace-Methode sind hierbei deutlich geringere Steigerungen der Bildfrequenz messbar. Besonders auffällig ist, dass die Performance in den Teapot und Bunny Szenen in Kombination mit den Sortiermethoden 1 und 2 sogar leicht sinkt. Sortiermethode 2 liefert mit einer Änderungsrate von -3% bei der Teapot Szene und -7% bei der Bunny Szene die schlechtesten Ergebnisse. Nur in Kombination mit Sortiermethode 3 sind bei allen Szenen Steigerungen der Bildfrequenz messbar.

fps	Methode 1		Methode 2		Methode 3	
Cornell Box	1100	+10%	1040	+4%	1140	+14%
Teapot	270	−1%	265	−3%	280	+3%
Bunny	79	−2%	75	−7%	84	+4%
Dragon	25	+19%	23	+10%	27	+29%
Sponza	55	+22%	54	+20%	57	+27%

Tabelle 8: Bildfrequenz mit der *Erste n/k Kandidaten* Trace-Methode und $k = 2$

Schnittpunktberechnung mit Dreiecksebene
Bei einer Ebenenbetrachtung der Kandidaten können mit der *Erste n/k Kandidaten* Methode und $k = 2$ deutlich höhere Bildfrequenzen erreicht werden. So steigt etwa die Änderungsrate bei der Sponza Szene von durchschnittlich +23% auf +184%. Wie bei der *Erster Kanidat* Methode hat das Sortierverfahren, bei einer Betrachtung der Kandidaten als Ebenen, keinen Einfluss auf die Geschwindigkeit.

fps	Methode 1		Methode 2		Methode 3	
Cornell Box	1100	+10%	1100	+10%	1100	+10%
Teapot	349	+28%	348	+28%	348	+28%
Bunny	132	+63%	132	+63%	132	+63%
Dragon	56	+167%	54	+157%	56	167%
Sponza	128	+184%	128	+184%	128	+184%

Tabelle 9: Bildfrequenz mit der *Erste n/k Kandidaten* Trace-Methode, $k = 2$ und Betrachtung der Kandidaten als Ebenen

4.3.3 Erster Schnittpunkt Methode

Die *Erster Schnittpunkt* Trace-Methode liefert in Kombination mit allen Sortiermethoden positive Änderungsraten der Bildfrequenz. Diese halten sich aber bei allen Szenen, mit Ausnahme der Dragon Szene, gering. Die Ergebnisse der verschiedenen Sortierverfahren liegen eng beieinander. Die besten Ergebnisse können in Kombination mit Sortiermethode 2 erzielt werden.

fps	Methode 1		Methode 2		Methode 3	
Cornell Box	1050	+5%	1055	+6%	1090	+9%
Teapot	273	+0,5%	278	+2%	277	+2%
Bunny	83	+2%	85	+5%	85	+5%
Dragon	25	+19%	25	+19%	25	+19%
Sponza	46	+2%	50	+11%	49	+9%

Tabelle 10: Bildfrequenz mit *Erster Schnittpunkt* Trace-Methode

4.4 Bildqualität

4.4.1 Erster Kandidat Methode

Bei Verwendung der *Erster Kandidat* Methode sinkt die Bildqualität unabhängig von der genutzten Sortiermethode stark. Abbildung 14 zeigt das Renderergebnis der Bunny und Sponza Szenen. Teapot, Bunny und Dragon Szenen haben gemeinsam, dass die Modelle von vielen großen Löchern durchzogen und nur noch als Punktwolken erkennbar sind. Grund dafür sind Shafts, in denen viele kleine Dreiecke liegen. Da selbst bei einer optimalen Sortierung nur eines dieser Dreiecke pro Shaft betrachtet wird, treffen viele Strahlen keine Geometrie. Durch eine höhere Linespace- oder Voxelgrid-Auflösung lässt sich dieses Problem zwar etwas verbessern, allerdings sinkt die Bildfrequenz dadurch so stark, dass sich das Verfahren nicht mehr lohnt. Auffällig ist auch, dass die Bodenebene eine Linie aufweist, an welcher viele kleine Löcher vorhanden sind. Diese kommen dadurch zustande, dass der Strahl an diesen Stellen Shafts traversiert, welche beide Dreiecke der Bodenebene beinhalten. Dadurch dass aber alle Strahlen nur mit einem dieser Dreiecke geschnitten werden, entstehen dort zwangsläufig Löcher. Die Sponza Szene weist im Vergleich kaum Löcher auf, es ist allerdings deutlich zu sehen, dass an nahezu allen Stellen die falschen Dreiecke beleuchtet werden. Grund hierfür ist, dass die Sponza Szene von vielen großen Flächen umschlossen ist. Traversiert ein Strahl die Szene ohne ein Objekt zu treffen, wird er in der Regel durch eine dieser großen Flächen gestoppt, da diese die zugehörigen Shafts komplett ausfüllen. Auch die Cornell Box Szene weist durch die niedrige Voxelgrid-Auflösung viele Artefakte auf.

Betrachtung der Kandidaten als Ebenen
Abbildung 15 zeigt die *Erster Kandidat* Trace-Methode mit einer Betrachtung der Kandidaten als Ebenen. Dadurch dass Schnittpunkte durch die Ebenenbetrachtung gewährleistet sind, weisen die Modelle keine Löcher mehr auf, die Linespace-Datenstruktur kommt aber in Form von rechteckigen Artefakten zum Vorschein. Dadurch lässt sich zwar die grobe Form der Modelle erkennen, Details und Beleuchtungseffekte sind aber kaum erkennbar. Zwischen den verschiedenen Sortierverfahren sind keine signifikanten Unterschiede in der Bildqualität erkennbar.

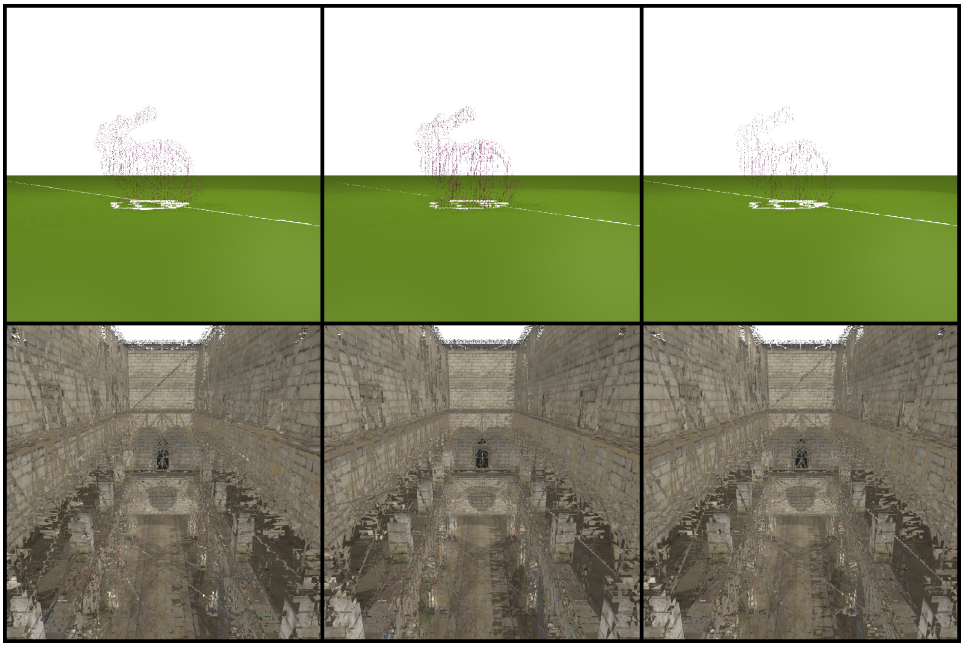

Abbildung 14: Bunny und Sponza Szene mit dem *Erster Kandidat* Trace-Verfahren.
Sortierverfahren v. l. n. r: Schnittpunkt, orthogonale Projektion, gemittelte orthogonale Projektion

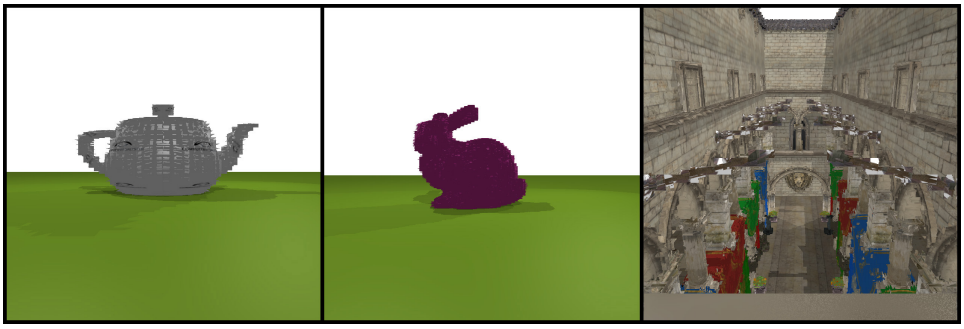

Abbildung 15: Teapot, Bunny und Sponza Szene mit dem *Erster Kandidat* Trace-Verfahren und einer Betrachtung der Kandidaten als Ebenen. Die Kandidatenliste ist nach gemittelter orthogonaler Projektion sortiert.

4.4.2 Erste n/k Kandidaten Methode

Die *Erste n/k Kandidaten* Methode liefert ähnliche Ergebnisse wie die *Erster Kandidat* Methode, abhängig von k sind aber weniger Löcher vorhanden, da mehr Kanidaten pro Shaft betrachtet werden. Abbildung 16 zeigt das Ergebnis der Bunny und Sponza Szene für $k = 2$. Obwohl durchschnittlich mehr

Kandidaten pro Shaft betrachtet werden, sind auch bei dieser Methode Löcher in der Bodenebene der Bunny Szene sichtbar. Da $k = 2$ gilt, wird an diesen Stellen wie bei der *Erster Kandidat* Methode nur eines der Dreiecke betrachtet und ein Loch entsteht. Das Problem lässt sich lösen, indem die Minimalanzahl der betrachteten Kandidaten, auf Kosten der Performance, von eins auf zwei erhöht wird. Die Bildqualität der Sponza Szene ist im Vergleich zur erster Kandidat Methode deutlich besser, so sind die Banner sichtbar, welche vorher nicht gerendert wurden. Insgesamt ist das Bild dennoch stark fehlerbehaftet und Details, wie der Löwenkopf, nicht erkennbar. Die Cornell Box Szene weist auch hier aufgrund der geringen Voxelgrid-Auflösung viele Artefakte auf. Zwischen den drei Sortierverfahren sind keine offensichtlichen Qualitätsunterschiede erkennbar.

Betrachtung der Kandidaten als Ebenen

Abbildung 17 zeigt die *Erste n/k Kandidaten* Trace-Methode mit einer Betrachtung der Kandidaten als Ebenen. Das Verfahren liefert in diesem Fall ähnliche Ergebnisse wie die *Erster Kandidat* Trace-Methode mit Ebenenbetrachtung. Abhängig von k ist die Anzahl an Linespace-Artefakten aber etwas geringer. Details und Beleuchtung sind trotzdem nur schlecht erkennbar.

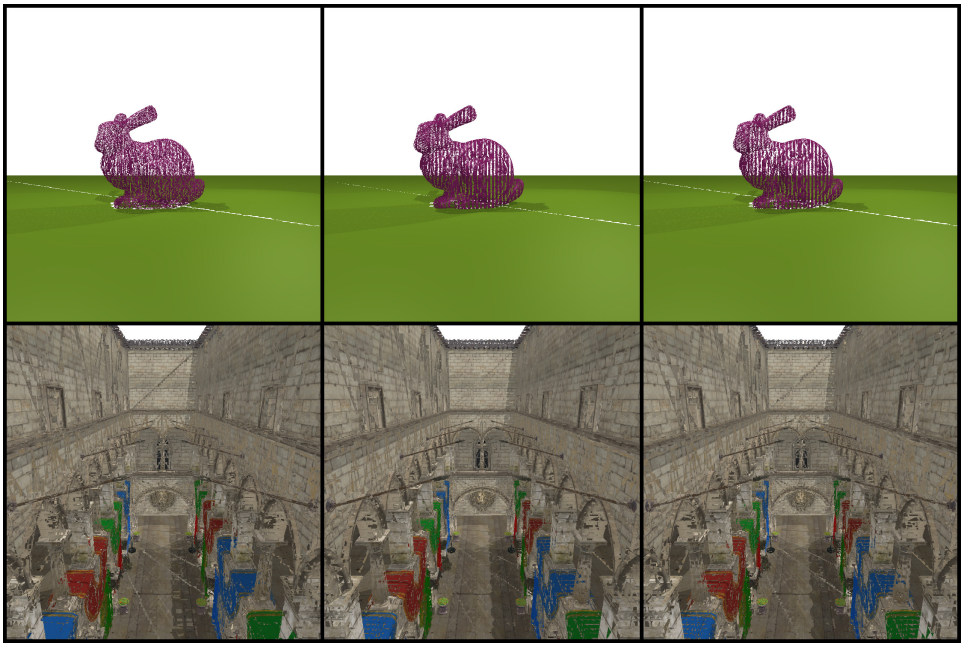

Abbildung 16: Bunny und Sponza Szene mit dem *Erste n/k Kandidat* Trace-Verfahren für $k = 2$
Sortierverfahren v. l. n. r: Schnittpunkt, orthogonale Projektion, gemittelte orthogonale Projektion

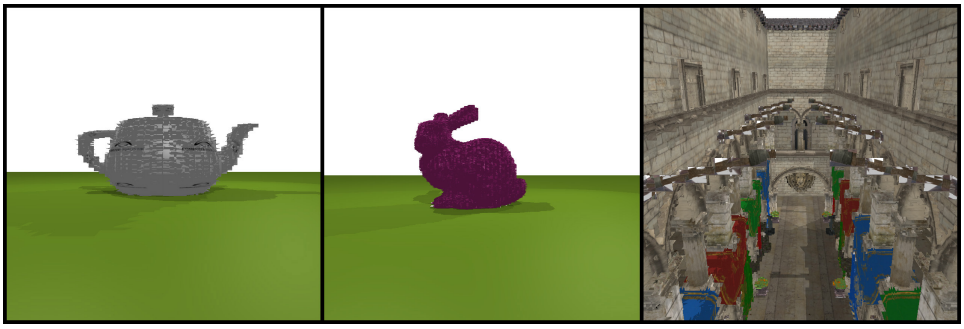

Abbildung 17: Teapot, Bunny und Sponza Szene mit dem *Erste n/k Kandidaten* Trace-Verfahren für $k = 2$ und einer Betrachtung der Kandidaten als Ebenen. Die Kandidatenliste ist nach gemittelter orthogonaler Projektion sortiert.

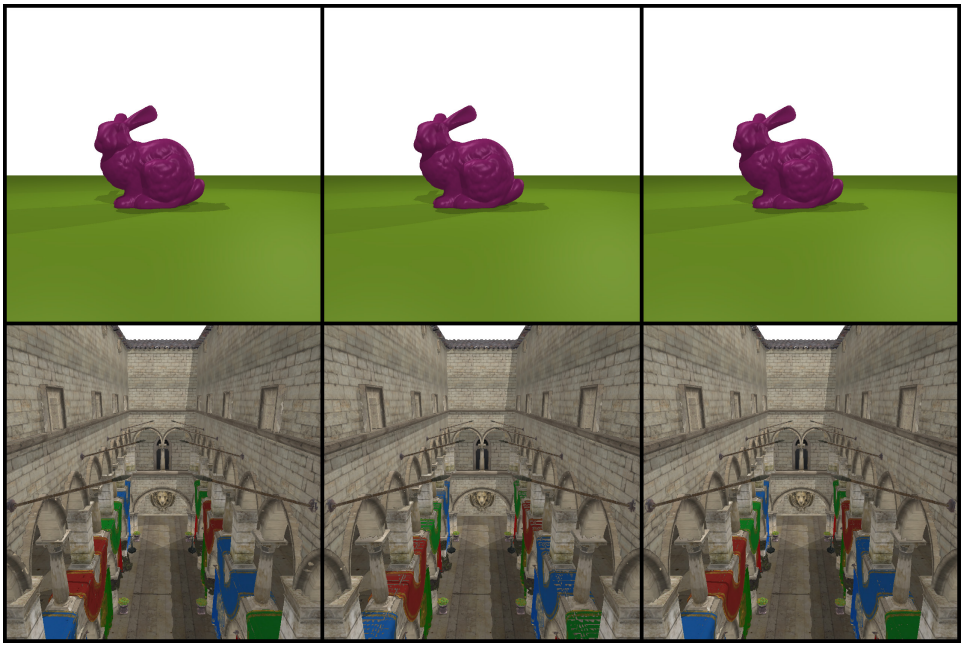

Abbildung 18: Bunny und Sponza Szene mit dem Erster Schnittpunkt Trace-Verfahren
Sortierverfahren v. l. n. r: Schnittpunkt, orthogonale Projektion, gemittelte orthogonale Projektion

4.4.3 Erster Schnittpunkt Methode

Abbildung 18 zeigt das Renderergebnis der Bunny und Sponza Szene mit der *Erster Schnittpunkt* Trace-Methode. Die Teapot, Bunny und Dragon Szenen weisen auf den ersten Blick keine offensichtlichen Fehler auf. Dadurch dass der Raytracer so lange Schnittpunkttests ausführt bis ein Schnittpunkt gefunden wird oder alle Kandidaten abgearbeitet sind, sind keine Löcher mehr

in den Modellen vorhanden. Bildfehler beschränken sich bei diesem Verfahren auf die Beleuchtung falscher Kandidaten durch Fehler in der Sortierung. Abbildung 19 zeigt die Teapot und Sponza Szenen in der Nahaufnahme. Die Teapot Szene weist bei allen Sortierverfahren Fehler am Übergang von Kannenkörper zu Tülle auf, besitzt abgesehen davon aber keine offensichtlichen Artefakte. Bei einer Sortierung nach orthogonaler Projektion sind die Artefakte am kleinsten. Etwas schlechter ist das Bild der Sponza Szene, welches in allen Verfahren Fehler an Kanten und den Bannern aufweist. Hierbei sind deutliche Unterschiede zwischen den drei Sortierverfahren sichtbar. Bei einer Sortierung nach Schnittpunkt sind an den Bannern nur wenige Bildfehler vorhanden. An Kanten sind jedoch viele kleinere Artefakte präsent. Bei einer Sortierung der Kandidatenliste nach orthogonaler Projektion sind an den Bannern viele Löcher vorhanden, Kanten weisen jedoch etwas weniger Fehler auf. Die insgesamt beste Qualität liefert die Sortierung nach orthogonaler Projektion. Dort sind sowohl bei den Bannern als auch an Kanten nur kleinere Artefakte vorhanden. Die Cornell Box Szenen weist im Vergleich zu den anderen Trace-Methoden weniger aber dennoch viele Artefakte auf.

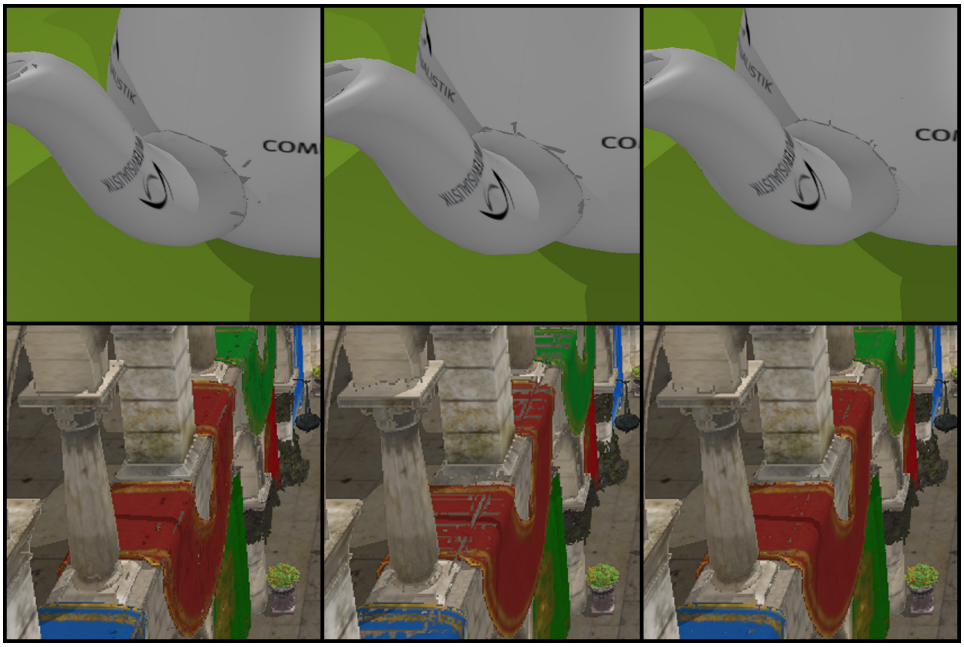

Abbildung 19: Nahaufnahme der Teapot und Sponza Szenen mit dem *Erster Schnittpunkt* Trace-Verfahren

4.5 Fazit

In dieser Arbeit wurden unterschiedliche Methoden zur Sortierung und Traversierung der Linespace-Datenstruktur vorgestellt. Das Ziel, eine Methode zu finden, welche in einer Performance-Steigerung ohne offensichtliche Minderung der Bildqualität resultiert, konnte allerdings nicht erfüllt werden.

Mit der *Erster Kandidat* Trace-Methode lässt sich in Kombination mit allen Sortierverfahren eine hohe Steigerung der Bildfrequenz verzeichnen. Die resultierende Bildqualität ist jedoch sehr gering, da viele Strahlen keine Geometrie treffen. Das Verfahren bietet dadurch keinen sinnvollen Ansatz zur Strahlenverfolgung mit der Linespace-Datenstruktur. Bei Verwendung der *Erste n/k Kanidaten* Trace-Methode ist die Bildqualität in der Regel besser, jedoch immer noch stark fehlerbehaftet. Zusätzlich resultiert das Verfahren in einer nur geringeren Steigerung der Bildfrequenz und in einigen Fällen sogar in einer Performance-Senkung. Bei einer Ebenenbetrachtung der Kandidaten können mit beiden Methoden hohe Bildraten erreicht werden, allerdings kommt die Linespace Geometrie in Form von Blockartefakten zum Vorschein.

Die beste Kombination aus Bildfrequenz und Bildqualität konnte mit dem *First Intersection* Trace-Verfahren und einer Sortierung der Kandidatenliste nach gemittelter orthogonaler Projektion erreicht werden. Dort sind nur wenige, aber dennoch bemerkbare, Bildartefakte vorhanden. Die Bildfrequenz steigt je nach Szene um 2% bis 19%, wobei größere Szene eine höhere Leistungssteigerung aufweisen. Zur Verfolgung von Primärstrahlen ist das Verfahren aufgrund der entstehenden Artefakte dennoch nicht geeignet. Bei einer Implementierung zur Verfolgung von Sekundärstrahlen könnte die Bildqualität ausreichend sein, ob der dadurch resultierende Leistungszuwachs lohnenswert ist, muss aber noch getestet werden. Dadurch dass Bildqualität und Leistungsteigerung zu großen Teilen szenenabhängig sind, ist es erforderlich das Verfahren mit weiteren und auch größeren Szenen zu testen.

Weitere Probleme sind der zusätzlich erforderliche Speicherplatz und die verlängerten Ladezeiten, welche durch die Sortierung der Kandidatenliste entstehen. Bei einer zukünftigen Implementation ließe sich der Speicherverbrauch im Falle der *Erster Kandidat* Methode stark reduzieren, indem nur die jeweils vordersten und hintersten Kandidaten der Shafts gespeichert werden. Die Generierungsdauer ließe sich in allen Verfahren durch einen effizienteren Sortieralgorithmus verringern.

Abschließend kann festgestellt werden, dass eine Sortierung der Linespace-Datenstruktur, in der hier vorliegenden Form, keine erhebliche Performance-Steigerung liefern kann, ohne gleichzeitig eine starke Reduktion der Bildqualität hervorzurufen.

Literatur

[App68] Appel, Arthur: Some Techniques for Shading Machine Renderings of Solids. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. New York, NY, USA : ACM, 1968 (AFIPS '68 (Spring)), 37–45

[AW87] Amanatides, John ; Woo, Andrew: A Fast Voxel Traversal Algorithm for Ray Tracing. In: *In Eurographics '87*, 1987, S. 3–10

[Ble90] Blelloch, Guy E.: Prefix sums and their applications. (1990)

[CG12] Crassin, Cyril ; Green, Simon: Octree-based sparse voxelization using the GPU hardware rasterizer. In: *OpenGL Insights* (2012), S. 303–318

[Gla84] Glassner, Andrew S.: Space subdivision for fast ray tracing. In: *IEEE Computer Graphics and Applications* 4 (1984), Oct, Nr. 10, S. 15–24. http://dx.doi.org/10.1109/MCG.1984.6429331. – DOI 10.1109/MCG.1984.6429331. – ISSN 0272–1716

[Gla89] Glassner, Andrew S. (Hrsg.): *An Introduction to Ray Tracing*. London, UK, UK : Academic Press Ltd., 1989. – ISBN 0–12–286160–4

[KK86] Kay, Timothy L. ; Kajiya, James T.: Ray Tracing Complex Scenes. In: *SIGGRAPH Comput. Graph.* 20 (1986), August, Nr. 4, 269–278. http://dx.doi.org/10.1145/15886.15916. – DOI 10.1145/15886.15916. – ISSN 0097–8930

[KLM16] Keul, Kevin ; Lemke, Paul ; Müller, Stefan: Accelerating spatial data structures in ray tracing through precomputed line space visibility. In: *Proceedings of the 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2016, S. 17–26

[PF05] Pharr, Matt ; Fernando, Randima: *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005

[RW80] Rubin, Steven M. ; Whitted, Turner: A 3-dimensional Representation for Fast Rendering of Complex Scenes. In: *SIGGRAPH Comput. Graph.* 14 (1980), Juli, Nr. 3, 110–116. http://dx.doi.org/10.1145/965105.807479. – DOI 10.1145/965105.807479. – ISSN 0097–8930

[SH74] Sutherland, Ivan E. ; Hodgman, Gary W.: Reentrant Polygon Clipping. In: *Commun. ACM* 17 (1974), Januar, Nr. 1, 32–42. http://dx.doi.org/10.1145/360767.360802. – DOI 10.1145/360767.360802. – ISSN 0001–0782

[Whi80] Whitted, Turner: An Improved Illumination Model for Shaded Display. In: *Commun. ACM* 23 (1980), Juni, Nr. 6, 343–349. http://dx.doi.org/10.1145/358876.358882. – DOI 10.1145/358876.358882. – ISSN 0001–0782