UNIVERSITÄT
KOBLENZ · LANDAU

Fachbereich 4: Informatik

Beschleunigung von GPU-basiertem Raytracing durch den adaptiven Linespace

Bachelorarbeit

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

vorgelegt von

Pablo Delgado Krämer

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

Zweitgutachter: Kevin Keul, M.Sc.
(Institut für Computervisualistik, AG Computergraphik)

Koblenz, im Mai 2019

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig 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)

Inhaltsverzeichnis

Zusammenfassung

Bildsynthese durch Raytracing gewinnt durch Hardware-Unterstützung in Verbraucher-Grafikkarten eine immer größer werdende Relevanz. Der Linespace dient dabei als eine neue, vielversprechende Beschleunigungsstruktur. Durch seine richtungsbasierte Natur ist es sinnvoll, ihn in andere Datenstrukturen zu integrieren. Bisher wurde er in ein Uniform-Grid integriert. Problematisch werden einheitlich große Voxel allerdings bei Szenen mit variierbarem Detailgrad. Diese Arbeit führt den adaptiven Linespace ein, eine Kombination aus Octree und Linespace. Die Struktur wird hinsichtlich ihrer Beschleunigungsfähigkeit untersucht und mit dem bisherigen Grid-Ansatz verglichen. Es wird gezeigt, dass der adaptive Linespace für hohe Grid-Auflösungen besser skaliert, durch eine ineffiziente GPU-Nutzung allerdings keine optimalen Werte erzielt.

Abstract

Image synthesis using ray tracing gains importance with the recent introduction of hardware acceleration in consumer GPUs. Regarding this technique, the Linespace proves itself as a novel and promising acceleration structure. Because of its direction-based nature, it is beneficial to integrate it into other data structures. Until now, the uniform-grid has been the structure of choice. Equally sized voxels are, however, not the best solution for scenes with varying degree of detail. This thesis introduces the adaptive Linespace as a combination of an Octree and the Linespace. The structure is examined by its ability to accelerate and compared to the grid-based approach. Results suggest that it scales better with higher grid resolutions but is inherently limited by inefficient GPU usage. Thus, it is unable to achieve a higher overall acceleration.

1 Einleitung

Computerbasierte Bildsyntheseverfahren werden seit Jahrzenten für eine Vielzahl von Anwendungsfällen verwendet. Insbesondere das Erzeugen von Bildern photorealistischer Qualität erweist sich nach wie vor als richtungsweisende Problemstellung. Detailechte globale Beleuchtung, Durchsichtigkeit, Reflektionen und Refraktionen sind mit herkömmlichen Rasterisierungs-Ansätzen jedoch nur schwer oder unmöglich umzusetzen. Als geeigneteres Verfahren dient Raytracing, welches durch seine Funktionsweise allerdings vergleichsweise rechen- und zeitaufwendig ist. Durch steigende Rechenkapazitäten und neue Entwicklungen wie Hardware-Unterstützung in Verbraucher-Grafikkarten, besitzt dieses Verfahren heutzutage eine immer größer werdende Relevanz.

Der Linespace ist eine neuartige Raytracing-Beschleunigungsstruktur, welche Informationen richtungsabhängig speichert. Um zu skalieren, ist es sinnvoll, ihn in eine andere Datenstruktur zu integrieren. Die bisherige Implementierung basiert auf einer Gitter-Datenstruktur, bei der jede Zelle durch einen separaten Linespace repräsentiert wird. Ein Octree erlaubt es, den Berechnungsaufwand weiter zu senken, da durch eine Hierarchie bei der Traversierung große Teile des Gitters übersprungen werden können. Der adaptive Linespace ist die Kombination des Octrees mit dem Linespace. Untersucht wird, ob er eine höhere Beschleunigung als das Grid erreichen kann.

Im folgenden Kapitel wird eine Übersicht über relevante Arbeiten gegeben. Danach werden die Grundlagen der Raytracing-Beschleunigungsstrukturen und ihre GPU-Implementierungen erläutert. Die Voxelisierung bildet eine wichtige Konstruktionsgrundlage der Datenstruktur. Was folgt ist eine kurze Beschreibung des Konzepts des Verfahrens, welches in der Implementierung umgesetzt wird. Es wird eine zweite Octree-Datenstruktur nach Laine und Karras implementiert und untersucht. In der Evaluation werden zuerst beide Sparse Voxel Octree Implementierungen und das Grid miteinander verglichen. Daraufhin folgt die Untersuchung des adaptiven Linespaces hinsichtlich seiner Beschleunigungsfähigkeit. Im Fazit werden die Ergebnisse zusammengefasst und auf die zentrale Fragestellung bezogen.

2 Verwandte Arbeiten

Der Linespace ist ein neues und relativ unerforschtes Gebiet. Die in 2016 von K. Keul, S. Müller und P. Lemke [KML16] eingeführte CPU-basierte Implementierung setzt eine Variante des rekursiven Grids ein, einen *N-Tree*, welche durch den Linespace in der Form einer zusätzlichen Abbruchbedingung beschleunigt wird. 2017 wurde in einer folgenden Arbeit der Linespace für das Berechnen von weichen Schatten verwendet [KKM17]. Dabei kam eine auf der GPU implementierte N-Tree Datenstruktur zum Einsatz. Eine Erweiterung des Linespaces um eine grobe Vorberechnung eines Kandidatens pro

direktionalem Bereich zur Approximation von globaler Beleuchtung wurde 2018 vorgestellt [KKM18]. In diesem Jahr, 2019, wurde untersucht, inwiefern eine Kombination des gerade beschriebenen Linespaces mit *Bounding Volume Hierarchien* (BVHs) dazu benutzt werden kann, das Verfahren der indirekten Beleuchtung zu beschleunigen [Keu+19].

Eine Linespace-Variante mit Kandidatenlisten von dynamischer Größe ist nicht nur auf ungenaue indirekte Beleuchtung beschränkt, sondern kann alleinstehend oder als Teil einer Datenstruktur artefaktlos das Verfolgen jeglicher Strahlen signifikant beschleunigen. Bisher wurde diese Linespace-Variante als Teil einer Gitter-Struktur eingesetzt. In dieser Arbeit wird die Integration in einen *Sparse Voxel Octree* (SVO) untersucht.

3 Grundlagen

Abbildung 1: Schematische Darstellung einer GP104 GPU [Nvi16]. Pascal Architektur, verwendet in der GeForce GTX 1080. Es gibt vier GPC mit jeweils zehn SMs. Jeder SM enthält 128 CUDA-Kerne und 48 KB L1-Cache Speicher. Die Größe des L2-Caches beträgt 2048 KB.

3.1 GPU Programmierung

Eine GPU (*Graphics Processing Unit*, Abbildung 1) kann stark parallelisierbare Prozesse wie die Farbwertbestimmung für Pixel eines Bildes schneller und energieeffizienter bewältigen als eine CPU (*Central Processing Unit*). Die hochsimultane Ausführung homogener Arbeitsvorgänge ist auch für völlig andere Problemstellungen nutzbar. GPC (*Graphics Processing Cluster*) bilden high-level Baublöcke der Nvidia GPU-Architektur. Sie enthalten jeweils eine *Raster Engine* und mehrere *Thread Processing Cluster* (TPC), welche sich wiederum aus einer *PolyMorph Engine* und einem *Streaming Multiprocessor* (SM) zusammensetzen. Die Raster-Engine ist für das Vorbereiten der Dreiecke, Rasterisierung und Z-Culling verantwortlich. PolyMorph Engines handhaben das Abrufen von Vertices, Tessellation, Viewport Transformationen, die Einrichtung von Vertex Attributen und perspektivische Korrektion. Ein SM enthält eine Menge von CUDA (*Compute Unified Device Architecture*) Kernen und mehrere Arten von Speicher. Als eine der wichtigsten Hardware-Komponenten benötigen nahezu alle Operationen früher oder später SMs. Dort werden Berechnungen in Gruppen von 32 Threads, *Warps*, abgearbeitet.

Die Grafikkarte ist besonders für Berechnungen geeignet, bei denen zeitgleich identische Operationen ausgeführt werden. Der Grund dafür ist die Natur der Warp-Abarbeitung: nach dem SIMT (*Single Instruction, Multiple Threads*) Schema wird für jeden Thread im Warp dieselbe Instruktion ausgeführt. Der große Geschwindigkeitsvorteil kann jedoch nur erreicht werden, wenn der Kontrollfluss eines Programmes möglichst verzweigungsfrei ist. Im Worst-Case führen alle Threads alle Zweige aus und verwerfen irrelevante Ergebnisse. Warp-Divergenz tritt auf, wenn sich die Threads an unterschiedlichen Stellen des Programms befinden. Dadurch wird das Rechenpotential der Grafikkarte nicht vollständig genutzt.

OpenGL ist eine Spezifikation, die vom Grafikkartenhersteller durch Hardware und Treiber implementiert und bereitgestellt wird. Eine größtenteils geräte- und herstellerunabhängige Verwendung der GPU ist durch diese zusätzliche Abstraktionsebene möglich. Dem Entwickler stellt sie durch eine API (*Application Programming Interface*) Möglichkeiten zur Verfügung, Speicher zu manipulieren, Hardware-Funktionalität anzusprechen oder benutzerdefinierten Code auszuführen. Traditionell wird das in Abbildung 2 dargestellte Pipeline-Modell zur Bildsynthese verwendet. Durch *Compute-Shader* ist es möglich, Berechnungen unabhängig von der Pipeline auszuführen. Diese und programmierbare Stufen der Pipeline werden auf den zuvor erwähnten SMs ausgeführt. Die Rasterisierung hingegen ist ein integraler fixed-function Bestandteil und durch Hardware-Unterstützung schnell und energieeffizient.

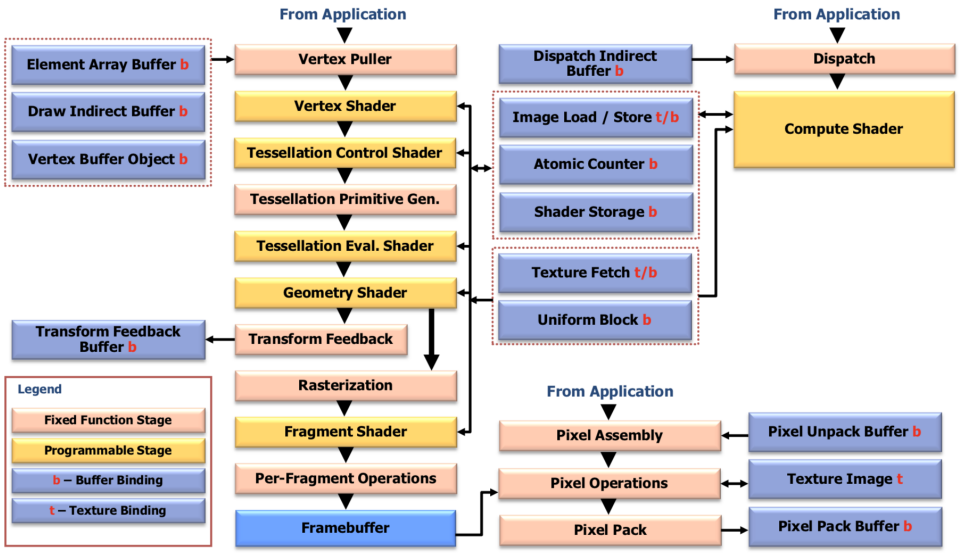


Abbildung 2: Block-Diagramm der OpenGL-Pipeline [SA19]. Vertex-Daten werden für die Rasterisierung hergerichtet. Der Framebuffer ist mehrfach gepuffert und nach einem Tausch das finale Resultat.

3.2 Raytracing

Anfänglich von Appel ’68 als *Raycasting* ins Leben gerufen [App68] und von Whitted ’79 durch Rekursivität erweitert [Whi79], erlaubt Raytracing die Approximation physikalisch korrekten Lichtverhaltens durch das Aussenden und Verfolgen von Strahlen. Für jeden Pixel werden dabei ein oder mehrere Primärstrahlen aus der virtuellen Kamera in die Szene geschickt. Es wird überprüft, ob ein Strahl ein Objekt schneidet. Mit Hilfe eines Beleuchtungsmodells kann dann ein Farbwert zugewiesen werden. Dazu müssen unter Umständen weitere Strahlen (Sekundärstrahlen) verschossen werden, um Effekte wie Transparenz, Refraktionen, Schattierungen oder indirekte Beleuchtung zu simulieren. Je nach gewünschtem Grad von Realismus ist es notwendig, diesen Vorgang viele Male rekursiv fortzuführen. Beim *Pathtracing* [Kaj86] wird somit eine Folge von Strahlen, ein Pfad, betrachtet.

Durch den nicht-linearen Anstieg der Menge von Strahlen und durch große und komplexe Szenen mit vielen Objekten ist bei einem naiven Durchlaufen aller Eventualitäten die Anzahl an Schnittpunkttests so hoch, dass die Synthese des Bildes eine lange Zeitspanne beansprucht. Um die Anzahl an Tests zu senken und sogar Echtzeitfähigkeit zu ermöglichen, werden Beschleunigungsstrukturen eingesetzt.

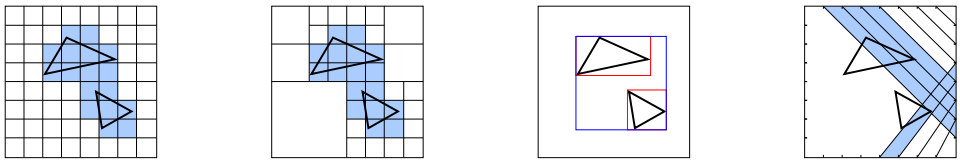


Abbildung 3: Einfache, zweidimensionale, konzeptuelle Visualisierung der für diese Arbeit relevantesten Beschleunigungsstrukturen.

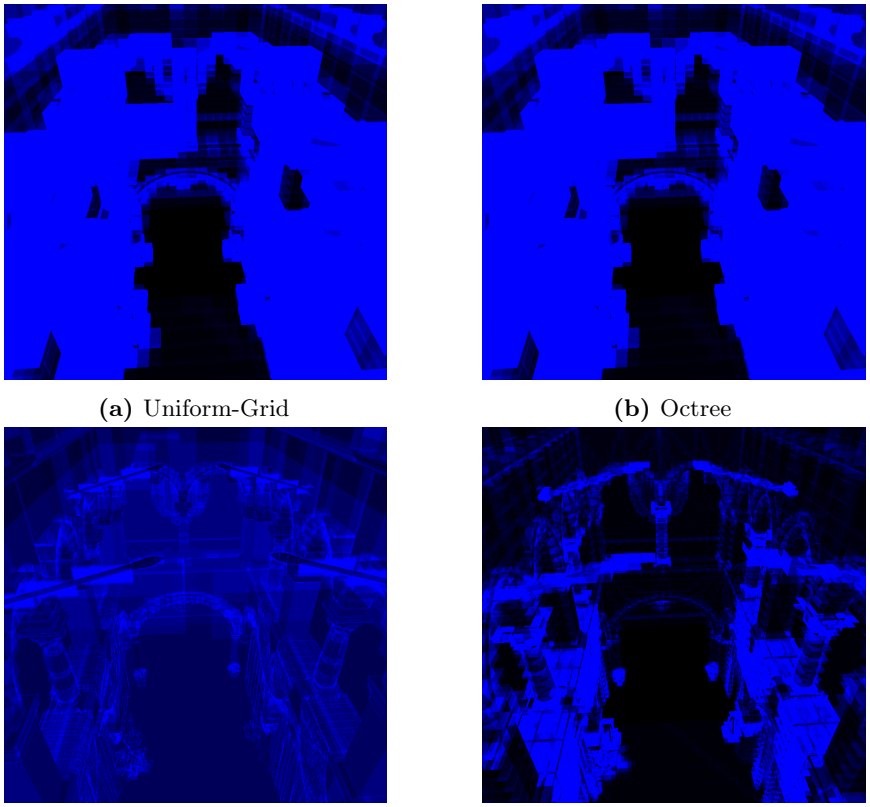
3.3 Beschleunigungsstrukturen

Wie Whitted 1979 schon bemerkte [Whi79], ist der Ursprung der langen Rechenzeit von Raytracing die Anzahl an benötigten Schnittpunkttests. Diese zu reduzieren ist eine der effektivsten Maßnahmen, um die Bildsynthese zu beschleunigen. Im Laufe der Zeit wurden für diesen Zweck Beschleunigungsstrukturen entwickelt. Vier ausgewählte Strukturen sind in Abbildung 3 dargestellt. Abbildung 4 visualisiert die Reduktion von Schnittpunkttests.

Die am häufigsten verwendete Beschleunigungsstruktur ist die Bounding Volume Hierarchie [KK86] [GS87]. Ein Binärbaum erlaubt es, die Anzahl an benötigten Schnittpunkttests von einer linearen $O(n)$ Komplexität an eine logarithmische $O(\log n)$ Komplexität anzunähern. *Node*s (Knoten) innerhalb der Baumstruktur verweisen entweder auf zwei *Child Nodes* (Kindsknoten) oder enthalten als *Leaf* (Blatt) eine Liste von Kandidaten. Im einfachsten Fall stellt dabei jede Node eine *axis-aligned Bounding Box* (AABB) dar, die möglichst genau die AABBs der Child-Nodes umfasst. Die Komplexitätsreduktion erfolgt dadurch, dass durch die Hierarchie eine binäre Suche, anstatt einer linearen Suche, vollführt werden kann.

Andere Beschleunigungsstrukturen basieren auf der Unterteilung eines diskret definierten Raumes. Uniform Grids [FTI86] benutzen ein Gitter, um nur Kandidaten zu testen, die sich in bestimmten Zellen befinden. Rekursive Gitter [JW88] und Octrees [Mea80] [Gla84] reduzieren den Traversierungsaufwand durch das Einführen einer Hierarchie. k-d-Trees sind mehrdimensionale binäre Suchbäume mit variierbaren Schnittebenen [Hav00]. Sie bilden einen speziellen Fall von *Binary Space Partitioning* (BSP) [FKN80].

Der Linespace [KML16] basiert auf der Unterteilung des Raumes in Richtungsschächte. Jede Richtung lässt sich einem einzigen Schacht zuordnen. Der Schacht kann Sichtbarkeitsinformationen, vorberechnete Farbwerte oder auch eine Liste von Kandidaten enthalten. Durch seine strikt direktionale Natur ist es sinnvoll, den Linespace in eine andere Datenstruktur zu integrieren. Diese Arbeit baut auf einem Kandidatenlisten-enthaltenden Linespace auf, welcher in ein Uniform-Grid integriert wurde.



(a) Uniform-Grid **(b)** Octree

(c) Bounding Volume Hierarchie **(d)** Uniform-Grid Linespace

Abbildung 4: Die Anzahl an benötigten Kandidatentests visualisiert als Heatmap. Schwarz bedeutet, dass kein Kandidatentest stattgefunden hat. Die Farbe Blau steht für 32 oder mehr Tests. Für die Grid-basierten Verfahren wurde eine 64^3 Auflösung verwendet. Die dargestellte Szene ist Sponza mit ~260k Dreiecken. Uniform-Grid und Octree weisen identische Farbwerte auf, da die Baum-Hierarchie letztendlich nur den Traversierungsaufwand reduziert.

3.3.1 Uniform Grid

Ein Uniform Grid unterteilt den dreidimensionalen Raum in gleichmäßig große Zellen (*Voxel, Volumetric Pixel*). Primitive befinden sich in Form einer Kandidatenliste in jedem Voxel und werden nacheinander überprüft, falls dieser von einem Strahl getroffen wird. Die Primitive in den anderen Voxeln werden nicht überprüft, was zu einer signifikanten Reduktion von Schnittpunkttests führt.

Nach der Einführung von Fujimoto et al. in 1986 [FTI86] untersuchten Amanatides und Woo [AW87] die effiziente Traversierung der Datenstruktur. In der hier vorkommenden Traversierung wird der 3D-*Digital-Differential-Analyzer* (3D-DDA) [FI85] verwendet. Zuerst wird das Grid an sich geschnitten, um die Position des initialen Voxels festzustellen. Darauf folgend werden dieser und alle weiteren dem Strahl folgenden Voxel iterativ geschnitten und auf ihren Inhalt überprüft. Sequentiell wird innerhalb eines Voxels ein Schnittpunkt gesucht und der Algorithmus endet, falls das dem Strahlursprung am nahe liegendsten geschnittene Primitiv ermittelt wurde.

Das Uniform Grid lässt sich durch Voxelisierungsalgorithmen erzeugen, worauf in Kapitel 3.4 eingegangen wird. Die Leistung der Beschleunigungsstruktur hängt stark von der Verteilung der Objekte innerhalb des Gitters ab. Ein Schlüsselfaktor ist es demnach, die Auflösung szenengerecht zu wählen, unter anderem durch das Verwenden von Heuristiken [ISP07]. Im Idealfall sind die Primitive gleichmäßig auf die Zellen verteilt. Ein Voxel mit vielen Primitiven ist aufwendig zu testen, aber eine Erhöhung der Auflösung hat eine langsamere Traversierung des Rests der Szene zur Folge. In diesem Szenario ist eine höhere Auflösung bestimmter Teilbereiche des Grids vorteilhaft.

Abhilfe schafft partielle adaptive räumliche Unterteilung: rekursive Grid-Strukturen [JW88] versuchen mit einer hierarchischen Ordnung das Problem zu lösen.

3.3.2 Sparse Voxel Octree

Octrees, zuerst von Meagher [Mea80] beschrieben und später von Glassner [Gla84] im Kontext von Raytracing angewandt, benutzen eine Hierarchie von Gittern unterschiedlicher Auflösung. Mit dieser wird jeder Voxel rekursiv in 2x2x2 Subvoxel zerlegt, die entlang des Strahls von oben nach unten traversiert werden. Octrees haben eine maximale Tiefe, die an den Stellen erreicht wird, an denen die Dichte von Primitiven am höchsten ist. Dort enthalten die Voxel keine Subvoxel, sondern eine Liste von Primitiv-Kandidaten. Diese letzte Ebene wird als *Leaf-Layer* bezeichnet. Ein Sparse Voxel Octree hat die Eigenschaft, dass leere Voxel und ihre Subvoxel keinen Speicher beanspruchen.

Der in dieser Arbeit verwendete GPU-basierte Sparse Voxel Octree Ansatz wird im Kapitel *Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer* von *OpenGL Insights* beschrieben [CG12].

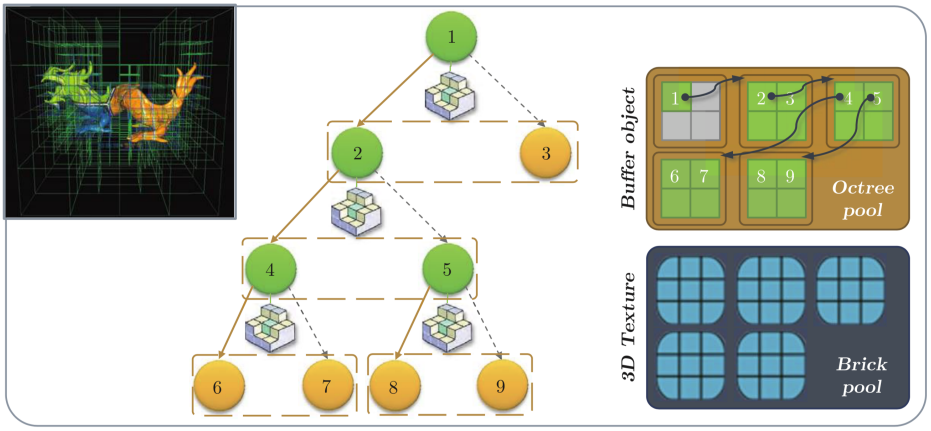


Abbildung 5: Visualisierung der Octree Datenstruktur nach Crassin und Green. [CG12]. Der *Brick Pool* repräsentiert hier das Leaf-Layer mit Voxelattributen anstelle von Primitiv-Kandidaten.

Speicherlayout Die Baumstruktur lässt sich mit Hilfe eines Buffers realisieren. Der *Octree-Pool* (auch *Node Pool* genannt) enthält Blöcke von jeweils acht 32-Bit Integern, die Buffer-Indizes der Child-Nodes repräsentieren. Bei der Traversierung nach unten wird so von Node zu Node gesprungen. Ein Wert, z.B. 0, gibt an, dass der Voxel weder Subvoxel noch Primitive enthält. Er kann deswegen übergangen werden. Anstelle von möglichen Blattknoten in oberen Ebenen der Hierarchie existieren diese in dieser Arbeit nur auf der Höhe der untersten Ebene. Es ist weiterhin anzumerken, dass Crassin und Green Voxelattribute wie Farbe innerhalb der Blätter speichern. Der hier implementierte Ansatz benutzt stattdessen eine Kandidatenliste. Abbildung 5 veranschaulicht die Datenstruktur im Speicher.

Konstruktion Der Konstruktionsalgorithmus nimmt ein Uniform Grid als Eingabe an und baut den Octree *top-down* auf. Er ist in Abbildung 6 dargestellt. Das Grid ist dabei die unterste Ebene mit der maximalen Auflösung, welche später als Leaf-Layer verwendet wird. Ein Hilfsbuffer, die *Voxel Fragment List* (1), welche aus dem Uniform Grid generiert wird und die Voxel mit ihren Positionen enthält, erlaubt uns, die anderen Ebenen zu generieren ohne mehrmals Voxelisieren zu müssen.

Im *Flagging*-Vorgang (2a), versucht für jeweils einen Voxel des Leaf-Layers ein Thread den Octree zu traversieren. Da dieser jedoch noch nicht vollständig erzeugt wurde, stoßen die Threads auf ihrem Weg zur untersten Ebene auf nicht initialisierte Nodes. Diese enthalten den Wert, mit dem der Buffer nach dem Erstellen überschrieben wurde. Sobald solch eine Node gefunden wird, kann der Octree nicht weiter traversiert werden. Stattdessen wird sie als besucht markiert und somit mit einem speziellen Wert wie 0 überschrieben.

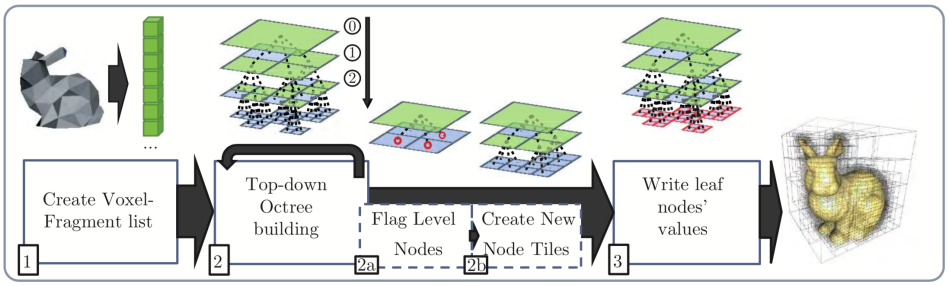


Abbildung 6: Übersicht über die Schritte der Sparse Voxel Octree Generierung (angelehnt an [CG12]).

Das Flagging merkt besuchte und somit nur die benötigten Nodes für das Erstellen vor. In dem darauf folgenden Schritt (2b) werden markierte Nodes gefunden und Offsets zu ihren neu reservierten Speicherbereichen zugewiesen. Dazu wird wie im vorherigen Schritt ein Thread per Node, der den Octree traversiert, benutzt. Bei dem Erreichen einer markierten Node wird ein *Shared Atomic Counter* inkrementiert, welcher die Anzahl der bis jetzt erstellten Nodes zurückgibt. Dadurch lässt sich der Offset der neuen Node im Buffer berechnen und der markierten Node zuweisen.

Flaggen und Erstellen findet für jede Ebene des Sparse Voxel Octrees statt. Anschließend wird ein weiterer Schritt ausgeführt, der den Leaf Nodes einen Offset in einen Kandidaten-Hilfsbuffer zuweist. Dieser enthält die Anzahl an Kandidaten und den Offset des ersten Primitivs.

Traversierung Algorithmus 1 fasst den implementierten Traversierungs-Algorithmus als Pseudocode zusammen. Er ist in der Literatur nicht aufzufinden, geht wie in den meisten Verfahren jedoch *depth-first* mit Hilfe eines Stacks vor. Dieser enthält aus Speichergründen lediglich Node-IDs.

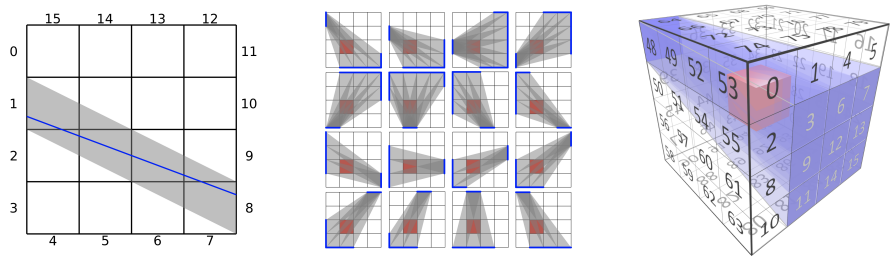
Zuerst wird die Bounding Box des Sparse Voxel Octrees geschnitten, um frühzeitig Strahlen zu verwerfen. Danach werden in den Zeilen 5–7 Variablen die initialen Zustände zugewiesen. Es folgt die Hauptschleife von Zeile 8 bis 31, die erst abbricht, wenn der Strahl den Octree wieder verlässt oder ein Primitiv geschnitten wird. Innerhalb der Schleife wird eine neue Node aus dem Node Pool gelesen und in einem der beiden Zweige verarbeitet.

Falls es sich bei der Node um kein Leaf handelt, sie nicht leer ist und der Child Index *cIndex* nicht ungültig ist, muss der Octree ab Zeile 11 weiter nach unten traversiert werden. Dazu wird der Index der momentanen Node auf den Stack gelegt und der Index der neuen Node, die im nächsten Durchgang behandelt wird, berechnet. Das erfolgt durch das Aktualisieren und Schneiden der neuen Voxel Bounding Box, resultierend in den *tMin* und *tMax* Werten. Sie werden für die Berechnung des nächsten Child Index und auch für eine effizientere Kandidaten-Überprüfung gebraucht.

Algorithmus 1 SVO Traversierung

```
 1: procedure CLOSESTINTERSECTION(Ray ray)
 2:     ensure ray.direction ≠ 0
 3:     if ray does not hit bounding box then
 4:         return false
 5:     level ← 0, stack[0] ← 0
 6:     nodePos ← (0, 0, 0), voxelSize ← 1
 7:     nodeIdx ← 0, cIndex ← intersect middle planes(ray)
 8:     while level ≥ 0 do
 9:         nodeIdx ← nodePool[nodeIdx]
10:         if nodeIdx is not leaf and not empty and cIndex.x ≠ −1 then
11:             level++
12:             nodeIdx ← nodeIdx + cIndex.x + cIndex.y × 2 + cIndex.z × 4;
13:             stack[level] ← nodeIdx
14:             nodePos ← nodePos × 2 + cIndex
15:             voxelSize ← voxelSize × 2
16:             minPos, maxPos ← voxel bounding box(nodePos, nodeSize)
17:             tMin, tMax ← ray-box intersection(ray, minPos, maxPos)
18:             cIndex ← compare pos to middle planes(ray, tMin, voxelSize)
19:         else
20:             if nodeIdx is leaf then
21:                 if candidate is hit(ray, nodeIdx, tMin, tMax) then
22:                     return true
23:             level−−
24:             nodeIdx ← stack[level]
25:             cIndex ← nodePos & 0x1
26:             nodePos ← nodePos ÷ 2
27:             voxelSize ← voxelSize ÷ 2
28:             tStep ← intersect voxel(ray, cIndex, nodePos, voxelSize)
29:             cIndex ← advance to parent sibling(tStep)
30:             if cIndex exceeds range then
31:                 cIndex.x ← -1
32:     return false
```

In dem anderen Zweig der Node-Behandlung, Zeilen 20–31, wird zunächst überprüft, ob es sich um ein Leaf handelt. In diesem Fall werden die Kandidaten mit dem Strahl getestet und der Algorithmus terminiert, falls ein Schnittpunkt gefunden wurde. Andernfalls muss zurück zur übergeordneten Node gegangen werden. Dazu wird in Zeile 24 der Index vom Stack genommen und Voxel-Eigenschaften wie Position und Größe werden neu berechnet. Der neue Child Index wird auf den des Nachbarvoxels des Parent Voxels gesetzt. Für den Fall, dass der Nachbar außerhalb des Parents liegt, wird in Zeile 31 der dreidimensionale Child Index durch einen negativen Wert als ungültig markiert. Das erwingt im nächsten Durchlauf wieder ein nach-oben Traversieren. Falls *level* durch konsekutives Aufsteigen wieder den Wert 0 erreichen sollte, terminiert der Algorithmus.



(a) (b) (c)

Abbildung 7: (a) Darstellung eines Strahls in einem leeren, zweidimensionalen Linespace. Durch den Start-Index 1 und den End-Index 8 kann die Shaft-ID berechnet werden. (b) Alle Shafts, die das rote Objekt abdecken. (c) Das Szenario in drei Dimensionen. Alle Shafts, die Start-Index 37 entspringen und das Objekt schneiden sind blau dargestellt. Entnommen aus [KML16].

3.3.3 Linespace

Der Linespace wurde 2016 von K. Keul, S. Müller und P. Lemke eingeführt [KML16]. Integriert wurde er in einen N-Tree einer CPU-basierten Applikation. Darauf folgende Arbeiten [KKM17] [KKM18] (siehe Kapitel 2) verwendeten eine GPU-Implementierung zur Berechnung von weichen Schatten und globaler Beleuchtung. Zuletzt wurde die Kombination von Bounding Volume Hierarchien mit Linespace-Nodes untersucht [Keu+19].

Bisher benannte Verfahren speichern pro direktionalem Bereich entweder eine binäre Sichtbarkeitsinformation oder eine einzelne Kandidaten-Information. Für eine ungenaue Berechnung von Beleuchtung mag dieser Ansatz valid sein. Eine Liste von Kandidaten hingegen erlaubt uns eine Beschleunigung des Verfolgens von Primär- und Sekundärstrahlen ohne visuelle Einbußen. In dieser Arbeit wird auf einem Kandidatenlisten-basierten Linespace als Teil einer Uniform-Grid Datenstruktur aufgebaut.

Aufbau Der Linespace teilt einen Quader in direktionale Schächte, sogenannte *Shafts* auf (Abbildung 7). Dazu wird auf jede Seite ein $N \times N$ dimensionales Gitter projiziert. Eine Zelle des Gitters wird als *Patch* bezeichnet. Ein Shaft wird als Verbindung zweier Patches s und e definiert und enthält Sichtbarkeitsinformationen über enthaltene Primitive. Das Verbinden der Patches jeder Seite mit den Patches jeder anderen führt zu $36N^4$ einzigartigen Shafts, die alle möglichen Richtungen repräsentieren.

Offensichtlich ist der Linespace mit höherer Parametrisierung von N zunehmend speicherlastig. Drei Eigenschaften der Datenstruktur erlauben es jedoch, die benötigte Speicherkapazität zu reduzieren:

1. *LS(s; e) = LS(e; s)*: Der Linespace ist symmetrisch und nur die Hälfte der Shafts werden benötigt.

2. *LS(s; s) = 0*: Es gibt degenerierte Shafts mit leerem Volumen.

3. *Komplanarität*: Shafts zwischen komplanaren Seiten sind entartet und müssen ebenfalls nicht berücksichtigt werden.

Letztendlich werden $15N^4$ Shafts benötigt. Für die binäre Variante lässt sich die Information, ob ein Shaft leer ist oder nicht, als Bit kodieren. Der Kandidatenlisten-Ansatz hingegen benötigt für jeden Shaft eine Zahl und ist somit deutlich speicherintensiver. Der Speicherverbrauch ist einer der größten Nachteile des Linespaces.

Generierung Die grobe Shaft-Richtungsunterteilung wird bei Szenen mit vielen Primitiven und Verdeckungen zum Verhängnis, da die Shafts zu viele Kandidaten enthalten. Deswegen ist es notwendig, den Linespace mit einer anderen Datenstruktur zu kombinieren. Eine Uniform-Grid basierte Struktur erlaubt nicht nur eine Beschleunigung zur Laufzeit, sondern auch eine Verringerung des Konstruktionsaufwands.

Für jeden Kandidaten enthaltenden Voxel des Uniform-Grids wird einmalig ein Linespace generiert. Das Erzeugen der Shaft-Kandidatenlisten erfolgt in zwei separaten Phasen mit dem Bilden einer Präfixsumme, auf die im Kontext der Voxelisierung (Kapitel 3.4) genauer eingegangen wird.

Grundlegend wird für jeden Shaft und jeden Kandidaten überprüft, ob diese sich schneiden oder ob der Kandidat vollständig enthalten ist. Dazu wird ein erweiterter 3D Sutherland-Hodgman Algorithmus [SH74] verwendet. In der ersten Phase des Verfahrens wird so die Anzahl an Kandidaten pro Shaft gezählt. Es folgt das Anwenden der Präfixsumme, die Voraussetzung für das effiziente Bilden von Listen variabler Größe ist. Als Resultat sind nun die Offsets der Kandidatenlisten bekannt und können im zweiten Durchlauf dazu benutzt werden, um die eigentlichen Kandidaten-IDs in den Speicher zu schreiben. Durch die hohe Anzahl an Shafts und der vielen Schnittpunkttests ist der Konstruktionsvorgang in der Regel nicht echtzeitfähig.

Traversierung Der Strahl wird mit der Bounding Box des Linespaces getestet. Dadurch ergeben sich Eintritts- und Austrittsseite sowie die betroffenen 2D-Koordinaten, an denen die jeweiligen Seiten durchdrungen werden. Mit Hilfe eines Algorithmus (Listing 1) wird die Shaft-ID bestimmt, die dazu benutzt wird, um aus einem Hilfsbuffer die Anzahl der Kandidaten und den Offset in die Kandidatenliste zu extrahieren. Falls die Kandidatenliste nicht leer ist, wird sequentiell jeder Kandidat mit dem Strahl getestet. Sobald das Primitiv mit dem am nächstliegendsten Schnittpunkt gefunden wurde, terminiert der Algorithmus.

In Kombination mit dem Uniform-Grid wird zunächst mit dem 3D-DDA über die Voxel iteriert (siehe Abbildung 8). Eine Besonderheit ist, dass die Ausgangsseite eines getroffenen Voxels zugleich die Startseite des nächsten

Voxels bildet. Dadurch muss pro Voxel nur eine Seite, die Ausgangsseite, getestet werden, was Rechenleistung einspart. Dieser Trick kann auch für die Berechnung der Linespace-Parameter verwendet werden. Diese müssen demnach pro Voxel nur einmal, anstatt zweimal, berechnet werden. Die Grid-Traversierung fährt fort, falls der Linespace-Test nicht erfolgreich ist.

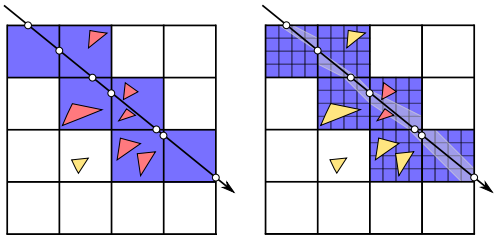


Abbildung 8: (a) Ein Uniform-Grid. Die blau markierten Voxel werden traversiert. Rot gefärbte Dreiecke müssen auf einen Schnittpunkt getestet werden. Gelbe Dreiecke werden nicht getestet. (b) Traversierung des Uniform-Grids mit Linespace-Voxeln. Die Überprüfung von Shafts reduziert die Anzahl an benötigten Tests.

```
uint computeShaftId(
    vec2 sDim, vec2 eDim, uint startFace, uint endFace)
{
    // Calculate linear index of patch on start side.
    int u_index = max(0, min(int(sDim.x * N), N - 1));
    int v_index = max(0, min(int(sDim.y * N), N - 1));
    uint index_s = N * v_index + u_index;

    // Calculate linear index of patch on end side.
    u_index = max(0, min(int(eDim.x * N), N - 1));
    v_index = max(0, min(int(eDim.y * N), N - 1));
    uint index_e = N * v_index + u_index;

    // Use shaft symmetry to reduce memory usage.
    uint block, shaftID;
    if (startFace > endFace) {
        block = faceToBlockMap[endFace][startFace];
        shaftID = N * N * N * N * block +
            N * N * index_e + index_s;
    } else {
        block = faceToBlockMap[startFace][endFace];
        shaftID = N * N * N * N * block +
            N * N * index_s + index_e;
    }
    return shaftID;
}
```

Listing 1: Berechnung der Shaft-ID. Als Eingabe werden Start- und Endseite sowie die korrespondierenden 2D-Intersektionskoordinaten verwendet. N ist die Auflösung des Patch-Gitters. *faceToBlockMap* ist eine 2D LUT, mit der die Shaft-Symmetrie ausgenutzt wird.

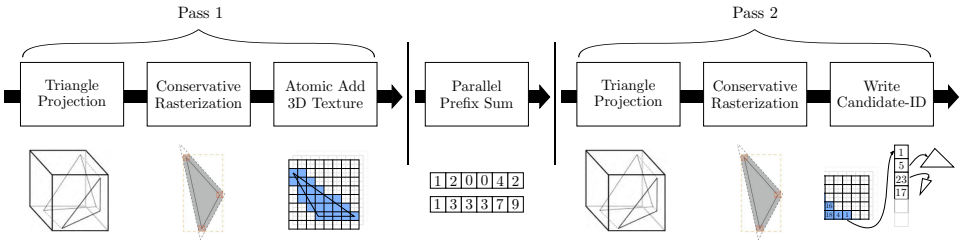


Abbildung 9: Eine grobe Übersicht über die Voxelisierungs-Pipeline (basiert auf [CG12]). Es werden zwei Durchgänge benötigt, um die Kandidatenlisten zu erzeugen.

3.4 Voxelisierung

Das Uniform-Grid ist das Resultat der Voxelisierung einer in der Regel dreiecksbasierten Szene. Bei echtzeitfähiger Anwendung mag es von Nöten sein, die Voxel-Repräsentation mehrmals in der Sekunde neu zu generieren. Um dies zu ermöglichen, schlagen Crassin und Green [CG12] ein Verfahren vor, welches sich die Hardware-Rasterisierung zu Nutze macht. Fixed-function Funktionalität mag nicht nur schneller sein als generische Compute-Ansätze, sondern ist auch energieeffizienter. Der Ansatz wurde auf Kandidatenlisten erweitert und bildet zusammen mit dem Uniform-Grid und dem Sparse Voxel Octree eine wichtige Grundlage des adaptiven Linespaces.

Dreiecks-Projektion Grundlegend funktioniert der Voxelisierungs-Ansatz so, dass Dreiecke orthogonal projiziert, rasterisiert und die entstehenden Fragmente den korrespondierenden Voxeln zugewiesen werden. Alle benötigten Operationen sind Teil einer Shader Pipeline (Abbildung 12) und lassen sich mit einem einzigen Draw-Call ausführen. Zuerst wird dazu die dominante Achse der Normalen gesucht. Das Dreick wird entlang dieser orthogonal projiziert, wodurch gewährleistet wird, dass die Fläche und somit die Anzahl an erzeugten Fragmenten maximal ist. Es wird die Viewport-Auflösung auf die des gewünschten Grids gesetzt. Danach findet die Hardware-Rasterisierung statt, die durch *2D scan conversion* Fragmente erzeugt. Anstelle von Farb- oder Tiefenwerten, die üblicherweise ausgegeben werden, wird nun der Fragment-Shader dazu benutzt, um mit Hilfe der interpolierten Position die betroffenen Voxel zu berechnen.

Konservative Voxelisierung Es wird zwischen zwei Arten der Oberflächenvoxelisierung unterschieden: *thin voxelization* [Hua+98] und *conservative voxelization* (siehe Abbildung 11). Während thin voxelization weniger rechenaufwendig ist, ist es doch möglich, dass unter bestimmten Situationen Löcher entstehen. Da das generierte Voxel-Grid im Kontext des Raytracings verwendet wird, führt dies zu sichtbaren Artefakten, die es zu vermeiden gilt.

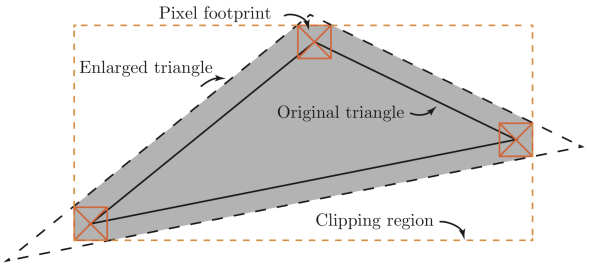


Abbildung 10: Visualisierung der Dilatation durch den Geometry Shader und des Clippings durch den Fragment Shader [CG12].

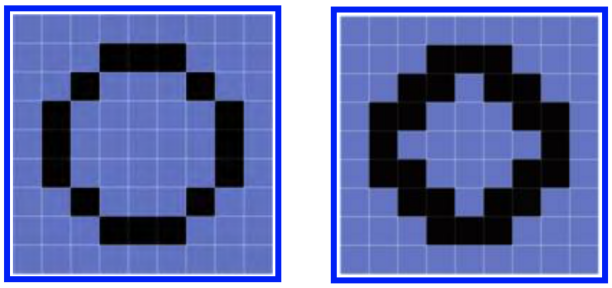


Abbildung 11: Beispiel für *thin* (links) und *conservative* (rechts) 2D Linien-Voxelisierung [CG12]. Analog auf eine 3D-Overflächenvoxelisierung übertragbar.

Die OpenGL-Rasterisierung arbeitet standardgemäß nicht konservativ und führt nur Fragment-Shader für Pixel aus, deren Mittelpunkt sich innerhalb der auf den Screen-Space projizierten Fläche eines Primitivs befindet. Um diese Überdeckung zu erreichen, wird mittels eines Geometry Shaders das Dreiecks-Primitiv dilatiert (Abbildung 10). Überflüssige Fragmente außerhalb der erweiterten Bounding Box des Primitivs werden verworfen. Neuerdings mag eine OpenGL-Extension wie GL_NV_conservative_raster für eine einfache konservative Rasterisierung verwendet werden.

Vollständige Konservative Voxelisierung Die bis jetzt beschriebene konservative Voxelisierung und somit der Ansatz, im Geometry Shader die Primitive zu dilatieren, ist für grobe Resultate ausreichend. Allerdings existiert die Problematik, dass pro Fragment nur ein Tiefenwert betrachtet wird, obwohl das Primitiv mehrere Voxel in der Tiefe schneiden kann. Da Voxel-basiertes Raytracing eine möglichst genaue Darstellung benötigt, muss der Ansatz erweitert werden. Bei der vollständigen konservativen Voxelisierung (auch *watertight voxelization*) werden zusätzlich die in der Tiefenrichtung benachbarten Voxel betrachtet. Welche Voxel das Primitiv durchdringt, wird mit dem Akenine-Möller Test [Ake02] überprüft. Für kubische Voxel ist es möglich, dass bis zu drei Nachbarn in der Tiefe betroffen sind.

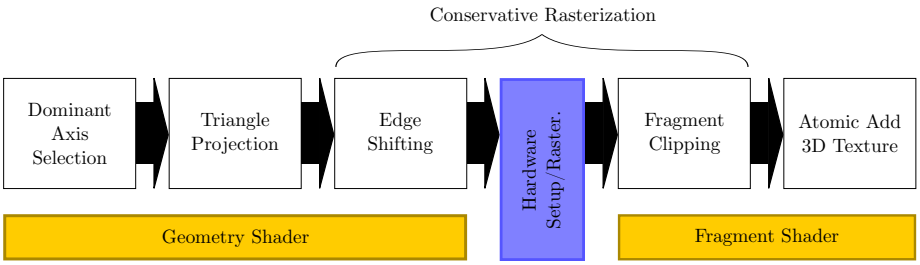


Abbildung 12: Mapping des ersten Voxelisierungs-Passes auf die GPU und ihr Shader-Modell (Kapitel 3.1). Angelehnt an [CG12].

Präfixsummenbildung Die Resultate der Voxelisierung werden nicht wie von Crassin und Green beschrieben als Voxel-Attribute in eine 3D-Textur geschrieben. Stattdessen besitzen die Voxel eine Liste von Primitiven, die geschnitten werden oder enthalten sind. Um dies umzusetzen, muss die Szene zweimal voxelisiert werden (siehe Abbildung 9). Zwischen den beiden Durchgängen wird eine Präfixsumme angewandt, angelehnt an [KLZ12].

Das Resultat der ersten Voxelisierung ist die Größe der Kandidatenliste. Es wird für jeden betroffenen Voxel der auf Null initialisierte Wert der korrespondierenden Zelle innerhalb der 3D-Textur inkrementiert. Das Anwenden der Präfixsumme läuft so ab, dass die Anzahlen in festgelegter Reihenfolge vom Voxel mit dem kleinsten Index bis zu dem Voxel mit dem größten Index parallel akkumuliert werden. Der letzte Voxel enthält somit die Gesamtanzahl an Primitiven in den Kandidatenlisten. Die nun aufaddierten Werte im 3D-Grid geben nicht nur den Offset in die Kandidatenliste an, sondern ermöglichen es auch, durch Betrachtung des Offsets des linear vorangehenden Voxels, die Anzahl an Primitiven in jeder Teil-Liste zu berechnen. Im zweiten Durchlauf werden die Primitiv-IDs in die Kandidatenlisten geschrieben.

4 Konzeption

Das Uniform-Grid enthält für jede Zelle einen separaten Kandidatenlisten-Linespace. Da die Datenstruktur mit komplexen Szenen schlecht skaliert, wird eine adaptive Struktur, ein Octree, in Betracht gezogen. Die Idee ist es, den Linespace nur auf dem Leaf-Layer der Baumstruktur einzusetzen. Durch die Octree-Traveriserung sind weniger Iterationen notwendig, um die vom Strahl getroffenen Linespaces zu erreichen. Anstelle einer Kandidantenliste wird so, bei dem Abfragen einer SVO Leaf-Node, ein Linespace-Shaft überprüft. Es bietet sich an, den Sparse Voxel Octree und die Uniform-Grid Linespace Datenstruktur unabhängig voneinander zu generieren und in der Traversierung miteinander zu verbinden. Um das zu realisieren, muss in erste Linie die Handhabung der SVO Leaf-Nodes verändert werden.

Diese Arbeit baut auf einem Framework der Arbeitsgruppe Computergrafik der Universität Koblenz-Landau auf, dem CVK, welches GPU-basierte Voxelisierung, den Kandidatenlisten-Linespace und eine Sparse Voxel Octree Beschleunigungsstruktur implementiert. Die Verfahren und ihre Eigenarten wurden in den Grundlagen bereits beschrieben. Nach einer Integration des Linespaces in die bestehende Sparse Voxel Octree Datenstruktur wird untersucht, ob ein neues Speicherlayout und ein neuer Traversierungsalgorithmus nach Laine und Karras [LK10a] die Struktur noch weiter optimieren können.

5 Implementierung

Das besagte Framework wurde um eine CMake-Integration erweitert, um mehrere Betriebssysteme unterstützen zu können. Dadurch lassen sich mögliche Auswirkungen des Systems und der Treiber besser vergleichen. Es wird OpenGL 4.6 mit *Direct State Access* (DSA), SSBOs und Compute-Shadern verwendet, sowie ein No-Error OpenGL-Kontext. Eine type-safe API beseitigt häufige Fehlerquellen. Rückgabewerte die Fehler signalisieren werden direkt aufgegriffen. OpenGL Debug-Logging gibt zusätzliche Hinweise bezüglich Fehlern und warnt vor Performance-Einbußen.

5.1 SVO Integration

Der Linespace wurde in die SVO-Traveriserung nach Algorithmus 3 integriert. Durch Unterschiede in den Traversierungsverfahren erfolgte dies fundamental anders als die bisherige Integration in das Uniform-Grid. Zuerst wird das Leaf-Handling der Uniform-Grid basierten Struktur mit Hilfe von Algorithmus 2 beschrieben. Danach wird erläutert, inwiefern sich die Integration in den Sparse Voxel Octree von dieser unterscheidet.

Das Uniform-Grid profitiert von der einheitlichen Voxel-Größe. Wie in den Grundlagen beschrieben, erlaubt diese es, nach der Traversierung eines Voxels die Ausgangsseite als die neue Eingangsseite zu verwenden (Zeile 19). Dieser Vorteil kann in Zeile 12 auch für die aufwendige Berechnung der Linespace Seiten-Indizes und der 2D-Intersektionskoordinaten genutzt werden. Sie erfolgt somit nur einmal pro Voxel, in Zeile 7. Die Berechnung wird dabei in allen Fällen ausgeführt, auch wenn sich der Linespace als leer erweist. Die Abfrage eines Linespace Shafts benötigt nur wenige zusätzliche Operationen, die in den Zeilen 10 bis 11 ausgeführt werden. Durch wenige und kurze Zweige im Kontrollfluss wird die parallele Natur der GPU besser ausgenutzt. Ein weiterer Vorteil der Uniform-Grid Traversierung ist, dass kein Stack benutzt werden muss. Das reduziert die Anzahl an Speicherzugriffen und den benötigten Cache-Speicher. Der große Nachteil des Verfahrens ist nach wie vor die Anzahl der zu traversierenden Voxel. Im schlimmsten Fall terminiert der Algorithmus erst, wenn der letzte Voxel vor Grid-Ende überprüft wurde.

Algorithmus 2
Uniform-Grid LS Leaf-Handling

```
 1: procedure INTERSECT(Ray ray)
 2:     init params
 3:     intersect start side
 4:     intersect end side
 5:     calc LS start side params
 6:     while true do
 7:         calc LS end side params
 8:         calc voxel index
 9:         if LS is not empty then
10:             calc shaft id
11:             get shaft count, offset
12:         start side LS params
                ← end side LS params
13:         if shaft count > 0 then
14:             if test candidates then
15:                 return true
16:         calc next voxel
17:         if next voxel outside grid then
18:             return false
19:         start side ← end side
20:         end side ← next side
21:     return false
```

Algorithmus 3
Adaptive LS Leaf-Handling

```
 1: procedure INTERSECT(Ray ray)
 2:     init params
 3:     while in octree do
 4:         node ← nodePool
 5:         if node is valid subnode then
 6:             traverse downwards
 7:             intersect start side
 8:         else
 9:             if node is leaf then
10:                 intersect end side
11:                 calc voxel index
12:                 calc LS start side params
13:                 calc LS end side params
14:                 calc shaft id
15:                 get shaft count, offset
16:                 if shaft count > 0 then
17:                     if test candidates then
18:                         return true
19:             go up and to neighbor
20:             if outside parent then
21:                 go up next time
22:     return false
```

Der Sparse Voxel Octree besitzt mehrere Ebenen mit Voxeln unterschiedlicher Größe. Dadurch lassen sich die Seiten nicht tauschen und eine teure, explizite Leaf-Behandlung muss in den Zeilen 9–18 durchgeführt werden. In dieser werden für jede Leaf-Überprüfung in den Zeilen 12–13 die Linespace-Parameter der Strahleintritts- und Strahlaustrittsseite berechnet. Die rechenintensiven Schritte lassen sich kaum anders in die SVO-Traverierung integrieren. Der Stack enthält schließlich Parent-Indizes von Voxeln über dem Leaf-Layer und kann somit nicht um Linespace-Parameter erweitert werden. Die Parameter einer Seite können auch nicht an den Nachbar-Voxel übergeben werden, da, um ihn zu erreichen, einmal nach oben und wieder nach unten traversiert wird. Die Logik für das Herein- und Heraustraversieren wurde in Algorithmus 1 bereits ausführlich beschrieben.

Es ist ersichtlich, dass, verglichen zu der Uniform-Grid Implementierung, mehrere große Verzweigungen existieren. Anhand des Kontrollflusses kann gemutmaßt werden, dass der Algorithmus sich weniger gut auf die GPU-Architektur abbilden lässt, als die Uniform-Grid Variante. Inwiefern sich dies auf die Leistung auswirkt und wie genau der Shader vom Treiber kompiliert, verändert und auf der Grafikkarte gesheduled wird, lässt sich nur durch eine spätere Analyse bestimmen. Schließlich sollte die Skalierbarkeit des Sparse Voxel Octrees mit einer steigenden Grid Auflösung die Nebeneffekte der scheinbar ineffizienteren Integration ausgleichen können.

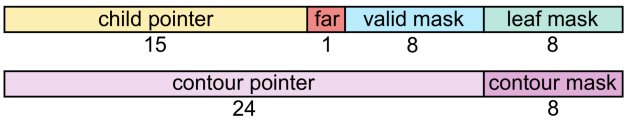


Abbildung 13: Ein 64-Bit *child-descriptor* von Laine und Karras [LK10a]. Konturen werden zur Beschleunigung der Traversierung verwendet. Das *Far-Bit* wird dazu benutzt, um mit einem zusätzlichen 64-Bit Offset auf weit entfernt liegende Nodes zu verweisen. Durch die *Leaf Mask* kann sich ein Leaf auf allen Ebenen befinden. Die *Valid Mask* gibt an, ob ein Voxel leer ist oder nicht.

5.2 Octree nach Laine und Karras

Laine und Karras präsentierten 2010 eine effiziente GPU Sparse Voxel Octree Implementierung [LK10a]. Diese ist für diese Arbeit aus zwei Gründen relevant:

1. Es wird ein kompaktes Speicherlayout vorgeschlagen, welches nicht nur den benötigten Speicher, sondern auch die Anzahl an benötigten Speicherzugriffen um den Faktor acht reduziert. Durch die Verwendung des Linespaces mit dem Speicheraufwand $15N^4$ und des Octrees mit dem konstanten Verzweigungs-Faktor von acht, ist eine große Limitation der Datenstruktur der vorhandene Speicher. Jede Minimierung erlaubt höhrere Auflösungen und resultiert zugleich in einer besseren Nutzung des Caches.

2. Ein hochoptimierter Traversierungsalgorithmus auf CUDA-Basis wird vorgestellt. Es steht genau fest, welche Variablen sich in welchem Speicher-Typ befinden und mehrere Tricks werden angewandt, um die Anzahl an Instruktionen und Speicherzugriffen zu senken.

Zuerst wird der in 3.3.2 beschriebene GPU SVO Konstruktionsalgorithmus so verändert, dass der generierte Octree besagtes kompaktes Speicherlayout besitzt. Danach folgt die Implementierung des Traversierungsalgorithmus in GLSL.

5.2.1 Kompaktes Speicherlayout

Im vorgestelltem Ansatz wird die meiste Information über eine Node innerhalb des Parents gespeichert. Dazu wird ein Integer aufgeteilt, sodass dieser einen Child Pointer und mehrere Bit-Masken enthält. Ob ein Subvoxel leer ist oder nicht, lässt sich nun anhand eines einzelnen Bits feststellen. Der Vorteil ist ein kompaktes Speicher-Layout, welches nicht nur weniger Speicher beansprucht, sondern auch die Anzahl an Speicherzugriffen und die verwendete Bandbreite senkt. Abbildungen 13 und 14 visualisieren das von Laine und Karras implementierte Speicherlayout.

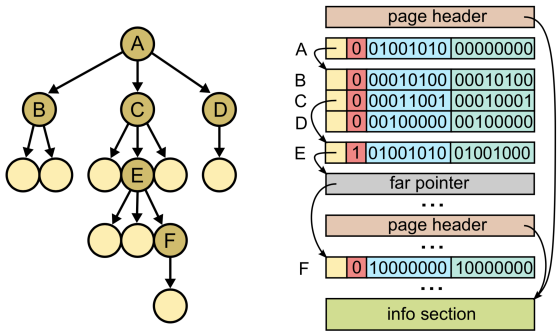


Abbildung 14: Links: Beispiel-Hierarchie eines Octrees. Rechts: Child-Descriptors im Buffer. Es werden Speicher-Seiten verwendet [LK10a].

Für diese Arbeit wurde eine starke Vereinfachung des Prinzips implementiert. Es wird kein Speicher-Seiten basierter Ansatz verwendet, womit das Far-Bit weg fällt. Konturen können immer noch später hinzugefügt werden, weswegen die Child Descriptor Größe auf 32-Bit reduziert wird. Schlussendlich gibt es ein Leaf-Layer und keine Leaf-Voxel in höheren Ebenen. Das heißt, dass die Leaf-Maske ebenfalls überflüssig ist. Was übrig bleibt ist ein 32-Bit Deskriptor mit einem 24-Bit Base Child Offset und einer 8-Bit Empty Bitmaske (Abbildung 15). Mit 24 Bit lassen sich 16 777 216 Werte darstellen, was für niedrige Auflösungen mehr als ausreichend ist. Listing 2 enthält den angepassten GLSL Code des Node-Creation Schritts der SVO Konstruktion.

```
#version 460 core

layout(local_size_x=1, local_size_y=1, local_size_z=1) in;

layout(binding=27, std430) restrict buffer nodePool_buffer
{
    uint nodePool[];
};

uniform uint nodeOffset;
uniform uint nodeCount;

void main()
{
    uint accumOffset = nodeOffset + nodeCount;

    for (uint i = 0; i < nodeCount; ++i)
    {
        nodePool[nodeOffset + i] |= (accumOffset << 8);
        accumOffset +=
            bitCount(nodePool[nodeOffset + i] & 0x000000FF);
    }
}
```

Listing 2: Der modifizierte *CreateNodes* Compute-Shader. Allen Nodes werden Child Pointer zugewiesen, die sich aus einem Ebenen-Offset und der Anzahl der Children zuvor bearbeiteter Nodes zusammensetzen.

child pointer	empty mask
000000000000000000000100	11111111
24	8

Abbildung 15: Der in dieser Implementierung verwendete Child-Descriptor.

5.2.2 Effiziente Traversierung

Da die GPU keinen *Call-Stack* besitzt und dadurch Funktions-Rekursion nicht möglich ist, muss zur Traversierung der Hierarchie ein Stack verwendet und depth-first vorgegangen werden. In jeder Iteration werden zur Auswahl des nächsten Voxels drei Fälle betrachtet:

- PUSH: Fahre mit dem Voxel fort, der als erstes vom Strahl geschnitten wird.
- ADVANCE: Fahre mit dem nächsten Nachbarvoxel fort.
- POP: Fahre mit dem nächsten Nachbarvoxel des Parent-Voxels fort, den der Strahl verlässt.

Der komplette Pseudocode des Algorithmus ist in Abbildung 16 zu sehen. In der Initialisierung wird das aktive Intervall des Strahls t_{min} bis t_{max} auf das Minimum und Maximum gesetzt. Der momentane Voxel ist durch *parent* und den Child Slot Index *idx* dargestellt. Der erste Voxel wird durch den Schnitt der Mittelflächen des Root-Voxels bestimmt. In Zeilen 8–39 wird dann so lange über die Voxel iteriert, bis der Strahl den Octree verlässt oder ein Primitiv gefunden wird. Zeilen 9–17 testen den Voxel und überprüfen, ob PUSH ausgeführt oder mit dem nächsten Voxel fortgefahren werden soll. Terminiert wird, falls sich der Voxel in Zeile 18 als Leaf herausstellt. Andernfalls werden *parent* und t_{max} in Zeilen 19–25 auf den Stack gepusht und die Werte auf den neuen Subvoxel angepasst. h ist dabei eine Optimierung, um unnötige Push-Operationen zu vermeiden. Für den Fall, dass der Voxel nicht getroffen und somit auch kein PUSH ausgeführt wird, findet der ADVANCE Vorgang statt. Dabei wird der nächste Nachbarvoxel in Betracht gezogen, unter der Annahme, dass der Strahl im selben Parent-Voxel bleibt. Falls sich diese Annahme als falsch herausstellt, wird in Zeilen 31–38 POP ausgeführt und der Nachbarvoxel des Parents weiter traversiert.

Optimierungen Die vorgestellte Traveriserung beherbergt einige Tricks, um Rechnungen und Speicherzugriffe zu vermeiden. Auffällig ist, dass der Schnittpunkttest nicht der Form $(p - o)/d$ entspricht, wobei p die zu schneidenden Flächen, o der Ursprung des Strahls und d die Richtung des Strahls sind. Er besitzt die Form $p * t_{coef} - t_{bias}$, wobei t_{coef} und t_{bias} in der Initialiserungsphase berechnet werden. Zum einen wird dadurch die Division durch eine Multiplikation ersetzt, was in der Regel schneller ist. Zum anderen besitzt

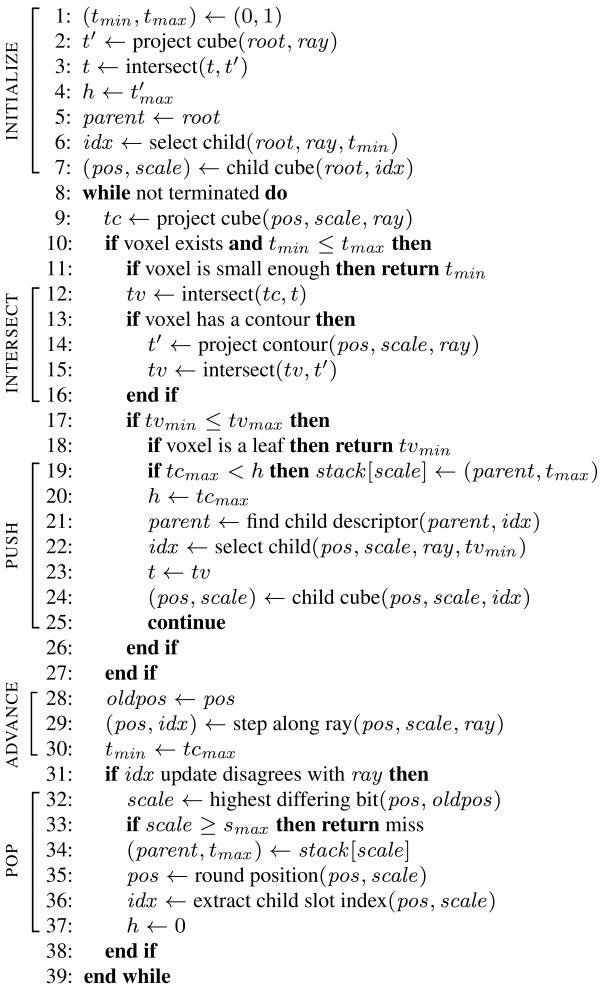


Abbildung 16: SVO-Traverierung nach Laine und Karras [LK10b]. Die über 270 Zeilen CUDA-Code sind im Appendix des Papers zu finden.

der Test nun die Form einer einzigen GPU MAD-Instruktion. Ein weiterer Trick mit größeren Auswirkungen ist die Verschiebung des gesamten Octrees in den Bereich $[1, 2]$. Das wird getan, weil durch geschickte Bitshift- und Interpretierungs-Operationen auf Integern und Floats im POP Vorgang aus der Position des Voxels die Skalierung errechnet werden kann. Abbildung 18 veranschaulicht den Vorgang. Dabei ist anzumerken, dass der Octree zusätzlich an bestimmten Seiten gespiegelt ist, da die Schnittpunktsuche positive Komponenten voraussetzt [RUL00]. Um Speicherzugriffe zu reduzieren, muss der Child-Descriptor nur aus dem Speicher geladen werden, wenn er vorher invalidiert wurde. Des Weiteren wird in PUSH nicht auf den Stack gepusht, wenn der Strahl beim nächsten POP Voxel und Parent zugleich verlässt.

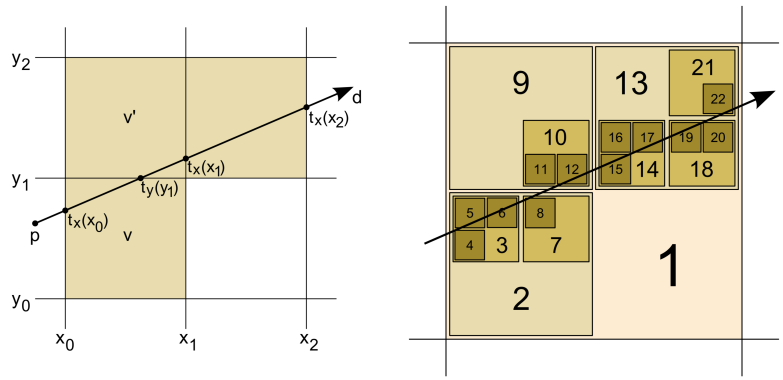


Abbildung 17: Links: Die Uniform-Grid Traversierung. Strahlaustrittsseite von Voxel v ist zugleich die Strahleintrittsseite von Voxel v'. Rechts: Die SVO-Traversierung. Der Algorithmus startet bei Voxel 2, einer Subnode des Root Voxels. PUSH wird zweimal ausgeführt um Voxel 3 und Leaf 4 zu erreichen. Indem ADVANCE zweimal ausgeführt wird, werden die Nachbarvoxel 5 und 6 erreicht. Da der Strahl aus Parent-Voxel 3 austritt, wird POP sowie ADVANCE aufgerufen und Voxel 7 traversiert. Der Algorithmus terminiert nach Leaf 22, da der Strahl aus dem Root-Voxel austritt. Entnommen [LK10b].

Implementierung Einige Änderungen sind notwendig, damit der Algorithmus mit dem Raytracing-Framework, GLSL und dem vereinfachten Speicherlayout nutzbar ist. Der SVO von Laine und Karras befindet sich auf allen Achsen im Bereich $[1, 2]$. Das erlaubt es, die Skalierung aus der Position zu extrahieren. Damit beide SVO-Implementierungen verglichen werden können, müssen die erzeugten Bilder jedoch pixelgenau übereinstimmen. Dazu ist es vonnöten, vor der Traversierung den Ursprung und die Richtung des Strahls zu verändern. Zuerst wird der Ursprung um den Startpunkt der Bounding Box versetzt. Der resultierende Vektor wird durch die Längen der Box dividiert und somit auf den Bereich $[0, 1]$ normalisiert. Zuletzt findet eine Verschiebung um eine Einheit in alle Richtungen statt. Das Ergebnis dieser Rechenoperationen ist ein neuer Aufpunkt des Strahls, welcher für die Traversierung des SVOs verwendet wird. Die Richtung wird ebenfalls durch die Maße der Bounding Box geteilt und auf die Länge von Eins normalisiert. Die Transformation des Raumes muss auch bei der Leaf-Behandlung beachtet werden. Dabei spielt auch die Spiegelung bestimmter Achsen eine Rolle, die Voraussetzung für effiziente Schnittpunkttests ist. Nach der Rücktransformation ist es mit Hilfe der Voxel-Position in Weltkoordinaten möglich, mit der Leaf-Überprüfung fortzufahren. Der Stack-Zugriff wurde modifiziert, um beliebige Stack-Größen zu erlauben und somit den benötigten Speicher zu reduzieren. Die Leaf-Behandlung wurde analog zu Kapitel 5.1 auf eine Linespace-Abfrage umgeschrieben.

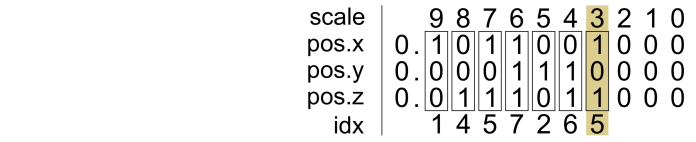


Abbildung 18: Verbindung zwischen Position, Child-Index und der Skalierung [LK10a]. Jedes Bit korrepondiert zu einem Skalierungs-Wert. Werden die drei Bits in Verbindung mit der Skalierung interpretiert, lässt sich der Index extrahieren. Bits links von der markierten Reihe definieren besagte Werte für höhere Octree-Layer, bis zur Wurzel.

5.3 Dynamische Buffergröße

Der Node-Buffer des SVOs muss vor dem Konstruktionsvorgang allokiert werden. Allerdings ist da noch nicht die Anzahl der Nodes und somit die Größe des Buffers bekannt. Die bisher implementierte Lösung war es, die theoretisch maximal verbrauchbare Menge an Speicher zu allokieren. Allerdings skaliert diese Lösung für einen Sparse Voxel Octree schlecht – bei einer Auflösung von 512^3 werden so mehr als 1GB Speicher allokiert, obwohl in der Realität nur 7MB von Nodes belegt sind. Wegen des speicherhungrigen Linespaces ist es notwendig, die Buffergröße zu reduzieren.

SSBOs können nicht vergrößert oder verkleinert werden. Sie können nur erstellt oder gelöscht werden. Deswegen ist eine Lösung, nach jedem Flagging-Create Schritt des Konstruktionsvorgangs ein neues SSBO mit anderer Größe anzulegen und den Inhalt des alten SSBOs zu kopieren. Der Nachteil ist, dass dadurch nahezu doppelt so viel Speicher temporär allokiert wird. Ein anderer Ansatz ist es, die Struktur zweimal zu generieren. Nach dem ersten Durchgang ist die byte-genaue Größe bekannt und und kann im zweiten Durchgang für die finale Buffer-Größe verwendet werden. Der offensichtliche Nachteil ist die lange Konstruktionszeit.

GL_ARB_sparse_buffer Die hier untersuchte und implementierte Methode benutzt die GL_ARB_sparse_buffer Extension. Diese fügt ein neues *glBufferStorage* Usage Flag, SPARSE_STORAGE_BIT_ARB hinzu. Mit diesem Parameter initialisierte Buffer operieren zuerst auf virtuellem Speicher. Deswegen ist es möglich, eine Buffer-Größe von 16GB oder mehr zu allokieren, selbst wenn die GPU keine derart hohe Speicherkapazität aufweist. Mit einem Kommando, *glBufferPageCommitmentARB*, lassen sich nun geräteabhängig große Speicherseiten in den physikalischen Speicher der Grafikkarte übertragen. Das erlaubt es, den Buffer wachsen zu lassen. Integriert wird das Wachsen des Buffers in den Konstruktionsvorgang, indem per geflaggter Node ein Shared Atomic Counter erhöht wird. Nach dem Flaggen wird der Counter ausgelesen und die Gesamtanzahl an Nodes in eine Buffergröße umgerechnet. Code-Segment 3 enthält die implementierte Buffer-Größenänderung.

```
void CVK::SparseSSBO::resize(GLsizeiptr sizeInBytes)
{
  assert(sizeInBytes > 0);
  sizeInBytes =
    (sizeInBytes + pageSize_ - 1) / pageSize_ * pageSize_;

  glBindBuffer(GL_SHADER_STORAGE_BUFFER, id_);

  // Commit pages from start to new size.
  GLsizeiptr commitStart = 0;
  GLsizeiptr commitSize = sizeInBytes;
  glBufferPageCommitmentARB(GL_SHADER_STORAGE_BUFFER,
      commitStart, commitSize, GL_TRUE);

  // Uncommit pages from new size to old size.
  GLsizeiptr uncommitStart = commitSize;
  GLsizeiptr uncommitEnd = sizeInBytes_;
  if (uncommitStart < uncommitEnd) {
    glBufferPageCommitmentARB(GL_SHADER_STORAGE_BUFFER,
        uncommitStart, uncommitEnd, GL_FALSE);
  }

  sizeInBytes_ = sizeInBytes;
}
```

Listing 3: Größenänderung eines Buffers mit der GL_ARB_sparse_buffer Extension. *glBufferPageCommitmentARB* nimmt als Argumente das Buffer Target, den Speicheroffset, die Anzahl an Bytes und ob die betroffenen Seiten commited oder uncommited werden. Offset und Byte-Anzahl müssen ein Vielfaches der Seitengröße sein. DSA führt an dieser Stelle zu Problemen und wird absichtlich nicht verwendet.

6 Evaluation

Zuerst werden das Uniform-Grid und beide Sparse Voxel Octree Implementierungen hinsichtlich Beschleunigungsfähigkeit und Speicherverbrauch getestet. Es wird schlussgefolgert, welche der Optionen unter welchen Umständen als Linespace-Kombination geeignet ist. Durch konventionelle nicht-Linespace Beschleunigungsstrukturen lassen sich höhere Grid-Auflösungen erreichen und Folgerungen bezüglich der weiteren Skalierung der Verfahren ziehen. Zusätzlich dazu lassen sich auf den niedrigen Auflösungen die Auswirkungen der Linespace-Integration deutlicher erkennen.

Im zweiten Teil werden die Linespace-basierten Datenstrukturen hinsichtlich ihrer Beschleunigungsfähigkeit miteinander vergleichen. Dabei kommen unterschiedliche Voxel-Grid Auflösungen und Linespace-Parametrisierungen zum Einsatz. Eine eingehende Analyse erklärt Geschwindigkeits-Diskrepanzen.

(a) Sponza, 262k Dreiecke

(b) Gallery, 998k Dreiecke

Abbildung 19: Renderergebnisse der verwendeten Test-Szenen.

6.1 Testablauf

Die Tests wurden auf einem Windows 10 64-Bit System mit einer Nvidia GeForce GTX 1060 GPU mit 3 GB GDDR5 VRAM, 1152 CUDA-Kernen und einer Taktrate von 1569 MHz durchgeführt. Auf CPU-Seite kam ein Ryzen 2700X mit 8 Kernen und einer Taktfrequenz von bis zu 4.3 GHz zum Einsatz. Es wurden 16 GB DDR4 RAM verwendet, von denen 8 GB durch *Shared Memory* mit der Grafikkarte geteilt wurden. Der Nvidia Game Ready Treiber 430.39 wurde benutzt.

Zwei Szenen wurden für die Tests verwendet, welche in Abbildung 19 zu sehen sind. Zum einen Sponza, eine architekturelle Szene mit 262 267 Dreiecken sowie Texturen. Durch das Atrium gibt es einen Leerraum. Säulen sowie Vorhänge besitzen einen hohen Detailgrad. Die andere Szene ist eine Gallerie [McG17] mit 998 941 Dreiecken. Beide Szenen wurden in der Auflösung 1920x1080 Pixel (Full HD) mit jeweils genau einem Primärstrahl pro Pixel gerendert. Die FPS wurde über mehrere Sekunden gemessen.

Scene	Res.	Uniform-Grid	SVO CVK	SVO Laine
Sponza	512^3	7	31	36
	256^3	11	27	24
	128^3	10	20	15
	64^3	5	10	7
	32^3	2	5	3
Gallery	512^3	10	19	23
	256^3	15	17	15
	128^3	16	11	8
	64^3	8	6	4
	32^3	3	2	2

Tabelle 1: Performance der Datenstrukturen in FPS.

Scene	Res.	Uniform-Grid	SVO CVK	SVO Laine
Sponza	512^3	569.97	3254.32	85.88
	256^3	78.04	413.59	23.6
	128^3	12.86	54.8	11.81
	64^3	3.41	8.65	3.05
	32^3	1.76	2.4	1.81
Gallery	512^3	566.54	3250.89	50.04
	256^3	82.12	417.67	20.08
	128^3	17.53	59.48	10.44
	64^3	7.58	12.83	6.88
	32^3	5.52	6.17	5.53

Tabelle 2: Speicherverbrauch der Datenstrukturen in MB.

6.2 Sparse Voxel Octree

Aus der Tabelle 1 lässt sich klar entnehmen, dass ein Sparse Voxel Octree basierter Ansatz mit steigender Auflösung zu einer höheren Raytracing-Beschleunigung führt als ein Uniform-Grid. Der Schwellwert ist szenenabhängig. Bei Sponza genügt schon eine niedrige Auflösung für eine höhere FPS und bei der Gallerie Szene scheint eine Beschleunigung ab einer Auflösung von 256^3 zu erfolgen.

Während die neue Octree Implementierung nach Laine für die höchsten Auflösungen die besten Ergebnisse erzielt, scheint die bisherige CVK SVO Implementierung in den niedrigeren Auflösungen überlegen zu sein. Der Grund dafür mag eine schnelle Traversierung durch das einfach gehaltene Speicherlayout sein. Durch die dynamische Buffergröße und das neue SVO-Layout verbraucht der Laine SVO für hohe Auflösungen signifikant weniger Speicher als die bisherige Implementierung. Das ist vorteilhaft, da somit mehr Speicher für die Linespace-Nodes zur Verfügung steht und die Speicherzugriffe kohärenter sind. Weiter ist aus Tabelle 2 abzulesen, dass der Speicherbedarf des CVK SVOs bei niedrigen Auflösungen noch im erträglichen Bereich liegt.

Durch den hohen Speicherverbrauch der CVK SVO Implementierung bei einer Auflösung von 512^3 Voxeln wird die VRAM-Grenze gesprengt und teilweise Shared Memory benutzt. Da sich der DDR4 und GDDR5 Speichertyp und die Speicheranbindung unterscheiden, wurden die besagten Ergebnisse mit einer GTX 1080 und 8 GB VRAM reproduziert und bestätigt. Aufgrund dessen ist anzunehmen, dass sich der Laine SVO für höhere Auflösungen wie 1k, 2k oder 4k weiter durchsetzen wird. Für die niedrigeren Auflösungen hingegen, wie sie bei die Linespace-Kombination erzwungen werden, scheint die CVK SVO Implementierung geeigneter zu sein.

Scene	LS	Res.	Uni.-LS	Adap.-LS Laine	Adap.-LS CVK
Sponza	N=2	256^3	31	32	32
		128^3	51	32	35
		64^3	46	22	27
		32^3	24	11	15
	N=3	256^3	-	-	-
		128^3	47	33	33
		64^3	60	29	36
		32^3	40	20	25
	N=4	256^3	-	-	-
		128^3	-	-	-
		64^3	56	32	38
		32^3	50	25	33
Gallery	N=2	512^3	5	16	16
		256^3	13	17	19
		128^3	19	16	19
		64^3	22	11	18
		32^3	12	7	10
	N=3	512^3	-	-	-
		256^3	-	-	-
		128^3	20	18	21
		64^3	23	16	20
		32^3	16	10	14
	N=4	512^3	-	-	-
		256^3	-	-	-
		128^3	-	-	-
		64^3	27	20	22
		32^3	21	14	18

Tabelle 3: FPS der Linespace-basierten Beschleunigungsverfahren mit unterschiedlicher Parametrisierung. N steht für die Linespace Shaft-Konstante.

6.3 Adaptiver Linespace

Aus Tabelle 3 lassen sich mehrere Folgerungen ziehen. Zum einen ist erkennbar, dass der adaptive Linespace für Auflösungen wie 256^3 oder 512^3 tatsächlich zu einer höheren Beschleunigung führt als der Uniform-Grid basierte Ansatz. Die Annahme, dass die adaptive Struktur besser skaliert, ist somit bestätigt.

Zum anderen fällt auf, dass die Strukturen durch die Integration des Linespaces unterschiedlich stark beschleunigt werden. Dabei scheinen die Ergebnisse nahezu widersprüchlich zu Tabelle 1 auszufallen. Da die Linespace-Abfrage bei allen Strukturen dieselbe ist und die Traversierung der SVOs laut Tabelle 1 für die meisten Auflösungen deutlich schneller ist, fällt der Verdacht auf die eigentliche Integration des Linespaces. Die Annahme, dass das Linespace Leaf-Handling bei den SVO-Strukturen weniger effizient auf die GPU abzubilden ist, als bei dem Uniform-Grid basierten Linespace, wurde in Kapitel 5.1 schon geäußert. Die genauen Gründe für die Performance-Einbußen werden in einer später folgenden Analyse näher untersucht.

Andere Messwerte hingegen entsprechen den Erwartungen. Es ist erkennbar, dass der Uniform-Grid basierte Linespace mit höherer Auflösung und somit zunehmendem Traversierungsaufwand weniger gut beschleunigt. Auch bei dem SVO lässt sich ab einer bestimmten Auflösung ein Leistungsverlust verzeichnen. In diesen Fällen ist die Auflösung zu hoch für die Szene. Voxel enthalten nur noch wenige oder keine Kandidaten und es ist günstiger diese zu testen, anstatt weitere Octree Ebenen nach unten zu traversieren. Beide Sparse Voxel Octree Implementierungen verhalten sich untereinander ähnlich zu den Tests in Tabelle 1. Die CVK SVO Implementierung ist dem Octree nach Laine und Karras in nahezu allen Fällen überlegen.

Auflösung des Voxel-Grids Anders als angenommen, beschränken numerische Grenzen und nicht der Speicher die Voxel-Grid Auflösung in Tabelle 3. Der Knackpunkt scheint die Anzahl an Linespace Shafts zu sein. Um diese Beschränkungen zu umgehen, mag es notwendig sein, große Änderungen an der bisherigen Implementation zu vollführen. 64-Bit Buffer-Indizes in der Linespace Implementierung könnten besagte Probleme beheben, würden allerdings den Speicherverbauch deutlich erhöhen. Für höhere Auflösungen wie 1k oder 2k muss zusätzlich das Voxelisierungs-Verfahren erweitert werden.

Bezüglich der Voxel-Grid Auflösung ist auch anzumerken, dass das Uniform-Grid variierbar groß sein kann. Der Sparse Voxel Octree muss auf allen Achsen dieselbe Auflösung besitzen, welche zusätzlich auf eine Potenz von Zwei beschränkt ist. Das führt auf nicht-kubischen Szenen zu einer ineffizienten und speicherverbrauchenden Grid-Auflösung, welche insbesondere in Kombination mit dem speicherintensiven Linespace zu einem großen Nachteil wird. Es ist allerdings möglich, durch das Erzwingen einer quadratischen Bounding Box, auf Kosten von Traversierungs-Iterationen, Speicher einzusparen.

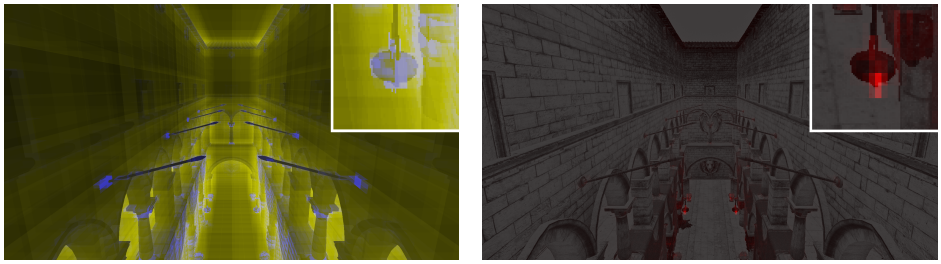
	GPU Time [ns]	Top SOL SM [%]	Top SOL TEX [%]	SM Active [%]	SM Active Min/Max Δ [%]	SM Warp Stall LSB [%]	L2 Hit Rate [%]	TEX Hit Rate [%]
Uniform-Grid	232.03	26.6	15.7	98.3	2.6	57.2	73.5	85.5
SVO CVK	125.82	32.1	19.8	94.6	5.8	54.8	85.0	81.8
SVO Laine	196.67	28.3	17.8	91.4	4.3	54.6	82.7	80.8
Uniform-LS	19.60	32.7	14.9	98.3	6.9	37.9	78.8	89.1
Adap.-LS CVK	32.80	36.2	12.2	89.1	14.5	37.5	90.1	74.0
Adap.-LS Laine	41.75	29.0	11.3	87.1	16.6	39.2	89.7	77.1

Tabelle 4: Die aussagekräftigsten Nsight Profiling Werte. Es wurde Sponza mit einer 64^3 Voxel-Grid Auflösung und Shaft-Konstante $N = 3$ analog zu den vorherigen Tests gemessen. SOL steht für *Speed of Light* und bezeichnet den maximalen theoretischen Durchsatz. LSB bedeutet *Long Scoreboard* und denotiert die TEX Latenz. TEX steht für den Texture-Cache. Auf der Pascal Architektur ist dieser zugleich der L1 Cache.

Analyse mit Nsight Um die Diskrepanzen zwischen dem Uniform-Grid Linespace und dem SVO-basierten Ansatz zu erklären, wurden mit Nvidia Nsight [Nvi19] Statistiken über die Traversierung der Strukturen auf der GPU gesammelt. Die relevantesten erhobenen Daten sind in Tabelle 4 aufgeführt.

Top SOL steht für die Auslastung der GPU-Einheit prozentual zu dem theoretischen Maximum. Von allen GPU-Einheiten werden die SMs am höchsten beansprucht, allerdings signalisieren Werte im Bereich von 26% bis 36%, dass diese bei weitem nicht ausgelastet sind. Gründe dafür können eine ineffiziente Arbeitsbelastung und Speichereffekte sein.

Eine aussagekräftige Metrik ist *SM Active*, was für den über die SMs gemittelten Prozentanteil von GPU Zyklen steht, in denen mindestens ein Warp für jeden SM aktiv ist. Ablesbar ist hier, dass diese für die adaptiven Linespace Implementierungen vergleichsweise gering ausfällt. Die Differenz zwischen dem minimalen und maximalen Prozentanteil ist umso höher. Daraus lässt sich folgern, dass die SMs in mehr Zyklen weniger gut mit Warps ausgelastet sind und die GPU dementsprechend weniger effektiv genutzt wird. Da auf Speicher wartende Warps als aktiv betrachtet werden, lässt sich vermuten, dass es ein paar wenige Warps gibt, die sehr viel Zeit benötigen. Es mag sein, dass die Threads innerhalb des Warps divergieren. Da die SVO Traversierung insbesondere drei teure Zweige besitzt, werden im Worst-Case alle hintereinander ausgeführt. Bei dem Uniform-Grid Linespace wirkt sich dieser Effekt durch den GPU-freundlicheren Kontrollfluss nicht derart stark aus. Dies lässt sich auch schon bei den nicht-Linespace Strukturen beobachten.



(a) **(b)**

Abbildung 20: (a) Eine kombinierte Heatmap. Der gelbe Farbkanal zeigt die Traversierungs-Iterationen des adaptiven Linespaces auf 172 normalisiert. Blaue Farbe steht für die Anzahl an Kandidatentests, auf 128 normalisiert. Es handelt sich um die Sponza Szene mit einer Auflösung von 64^3 und $N = 3$. (b) Eine GPU Clock Cycle Heatmap in rot, auf 8 000 000 Zyklen normalisiert. Dieselbe Sponza Szene, als Graubild und mit 5% Deckkraft.

Werte zwischen 10–20% bei *Top SOL TEX* und der Fakt, dass der primäre Warp Stall Faktor *SM Active Warp Stall Long Scoreboard* mit 35–60% ist, lassen vermuten, dass die L1 Speicherlatenz eine sekundäre Limitation darstellt. Diese ist bei den Linespace-Strukturen weniger stark ausgeprägt. Die letzten beiden Metriken sind respektitiv die L2- und L1 Cache Hit Raten. Es ist erkennbar, dass die SVOs nach der Integration eine schlechtere L1-Cache Nutzung aufweisen, während die des Uniform-Grids gestiegen ist. Es ist davon auszugehen, dass die Unterschiede von 10%-15% ebenfalls zu einer bemerkbaren Verlangsamung des adaptiven Linespaces führen. Schlechte Cache-Nutzung kann insbesondere bei Warps mit hoher Divergenz auftreten und zusätzlich für eine längere Laufzeit sorgen.

Heatmaps Es wurden drei Heatmaps angefertigt, die in Abbildung 20 zu sehen sind. Besonders interessant ist dabei die GPU Clock Cycle Heatmap (b), in der mit Hilfe der GL_ARB_shader_clock OpenGL-Extension die Ausführungszeiten der Threads gemessen wurden. Es ist erkennbar, dass diese alles andere als homogen sind, wie es zum Beispiel bei dem Uniform-Grid der Fall wäre. Stattdessen scheint es, als würden einige wenige Warps die gesamte Synthese des Bildes in die Länge ziehen. Das bestätigt den Verdacht der ineffizienten SM-Nutzung, welche die Nsight-Werte bereits suggeriert haben. Die in der Abbildung sehbaren roten Stellen sind, wie aus (a) entnehmbar ist, detailliert und enthalten vergleichsweise viele Kandidaten. Es lässt sich auch sehen, dass an den betroffenen Stellen viele Iterationen stattfinden. Im Worst-Case müssen Leaf-Voxel mehrmals vom Warp überprüft werden, da die Threads diese in unterschiedlichen Iterationen erreichen. Dabei werden jedes Mal alle Linespace-Parameter vom gesamten Warp neu berechnet. Die Cache-Latenz und schlechte Hit-Raten beeinflussen den Vorgang zusätzlich.

Schlussfolgerungen Es wurde mit Hilfe von Benchmarks gezeigt, dass der adaptive Linespace zwar für hohe Auflösungen eine bessere Beschleunigung verzeichnet, allerdings keine optimalen Ergebnisse liefert. Eine eingehende Untersuchung mit Nsight und Heatmaps haben die Annahme bewiesen, dass die GPU-Ressourcen weniger effizient genutzt werden. Grund dafür scheint die aufwendige Linespace-Abfrage als Teil einer für die Grafikkarte suboptimalen Arbeitsbelastung zu sein.

7 Fazit

Der kandidatenbasierte Linespace wurde in zwei Sparse Voxel Octree Beschleunigungsstrukturen integriert. Es wurden Benchmarks mit unterschiedlichen Szenen und Parametern durchgeführt, Heatmaps angefertigt und eine Analyse mit Nsight betrieben. Die Ergebnisse zeigen, dass sich der adaptive Linespace deutlich schlechter auf die Grafikkarte abbilden lässt, als der Uniform-Grid basierte Linespace. In den meisten Fällen werden keine optimalen Ergebnisse erreicht, da sich die Vorteile der Hierarchie erst ab sehr hohen Auflösungen ergeben.

Durch die Integration des Linespaces konnte die Beschleunigungsfähigkeit der Sparse Voxel Octrees weiter erhöht werden. Auch konnte bewiesen werden, dass der adaptive Linespace verglichen zum Uniform-Grid Linespace für hohe Auflösungen besser skaliert.

Dennoch ist der adaptive Linespace nicht dazu in der Lage, die Beschleunigung des Uniform-Grid basierten Linespaces zu übertreffen. Die parallele Natur der Grafikkarte wird durch die aufwendige Traversierung weniger gut ausgenutzt. Zusätzlich muss durch die Octree-Grundstruktur die Grid-Auflösung kubisch und eine Potenz von zwei sein. Das führt bei Szenen mit nicht-kubischen Maßen zu einer ineffizienten Unterteilung und einem erhöhten Speicherverbrauch.

Die Überprüfung des Linespaces ist aufwendig und eignet sich wenig zur Integration in eine komplexe, adaptive GPU Beschleunigungsstruktur. Nichtsdestotrotz werden mit steigenden Anforderungen adaptive Ansätze relevanter. Es wäre deswegen interessant zu untersuchen, inwiefern sich der Linespace in Strukturen anderer Art integrieren lässt. Auch ist eine Kombination mit dem Traversierungs-beschleunigenden binären Linespace nicht auszuschließen.

Literatur

[Ake02] Thomas Akenine-Möller. „Fast 3D Triangle-box Overlap Testing“. In: *J. Graph. Tools* 6.1 (Jan. 2002), S. 29–33. ISSN: 1086-7651. DOI: 10.1080/10867651.2001.10487535.

[App68] Arthur Appel. „Some Techniques for Shading Machine Renderings of Solids“. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. AFIPS '68 (Spring). Atlantic City, New Jersey: ACM, 1968, S. 37–45. DOI: 10.1145/1468075.1468082.

[AW87] John Amanatides und Andrew Woo. „A Fast Voxel Traversal Algorithm for Ray Tracing“. In: *EG 1987-Technical Papers*. Eurographics Association, 1987. DOI: 10.2312/egtp.19871000.

[CG12] Cyril Crassin und Simon Green. „Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer“. In: *OpenGL Insights*. Hrsg. von Patrick Cozzi und Christophe Riccio. CRC Press, Juli 2012. Kap. 22, S. 303–319. ISBN: 978-1439893760.

[FI85] A. Fujimoto und K. Iwata. „Accelerated Ray Tracing“. In: *Kunii T.L. (eds) Computer Graphics* (1985), S. 41–65.

[FKN80] Henry Fuchs, Zvi M. Kedem und Bruce F. Naylor. „On Visible Surface Generation by a Priori Tree Structures“. In: *SIGGRAPH Comput. Graph.* 14.3 (Juli 1980), S. 124–133. ISSN: 0097-8930. DOI: 10.1145/965105.807481.

[FTI86] A. Fujimoto, T. Tanaka und K. Iwata. „ARTS: Accelerated Ray-Tracing System“. In: *IEEE Computer Graphics and Applications* 6.4 (Apr. 1986), S. 16–26. ISSN: 0272-1716. DOI: 10.1109/MCG.1986.276715.

[Gla84] A. S. Glassner. „Space subdivision for fast ray tracing“. In: Bd. 4. 10. Okt. 1984, S. 15–24. DOI: 10.1109/MCG.1984.6429331.

[GS87] J. Goldsmith und J. Salmon. „Automatic Creation of Object Hierarchies for Ray Tracing“. In: *IEEE Computer Graphics and Applications* 7.5 (Mai 1987), S. 14–20. ISSN: 0272-1716. DOI: 10.1109/MCG.1987.276983.

[Hav00] Vlastimil Havran. „Heuristic Ray Shooting Algorithms“. Diss. Faculty of Electrical Engineering, Czech Technical University Prague, Nov. 2000.

[Hua+98] Jian Huang u. a. „An Accurate Method for Voxelizing Polygon Meshes“. In: *Proceedings of the 1998 IEEE Symposium on Volume Visualization*. VVS '98. Research Triangle Park, North Carolina, USA: ACM, 1998, S. 119–126. ISBN: 1-58113-105-4. DOI: 10.1145/288126.288181.

[ISP07] Thiago Ize, Peter Shirley und Steven Parker. „Grid Creation Strategies for Efficient Ray Tracing“. In: Okt. 2007, S. 27–32. ISBN: 978-1-4244-1629-5. DOI: 10.1109/RT.2007.4342587.

[JW88] David Jevans und Brian Wyvill. „Adaptive voxel subdivision for ray tracing“. In: 1988.

[Kaj86] James T. Kajiya. „The Rendering Equation“. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, S. 143–150. ISBN: 0-89791-196-2. DOI: 10.1145/15922.15902.

[Keu+19] Kevin Keul u. a. „Combining Two-level Data Structures and Line Space Precomputations to Accelerate Indirect Illumination“. In: Jan. 2019, S. 228–235. DOI: 10.5220/0007345902280235.

[KK86] Timothy L. Kay und James T. Kajiya. „Ray Tracing Complex Scenes“. In: *SIGGRAPH Comput. Graph.* 20.4 (Aug. 1986), S. 269–278. ISSN: 0097-8930. DOI: 10.1145/15886.15916.

[KKM17] Kevin Keul, Nicolas Klee und Stefan Müller. „Soft shadow computation using precomputed line space visibility information“. In: *Journal of WSCG* 25 (Jan. 2017), S. 97–106.

[KKM18] Kevin Keul, Tilman Koß und Stefan Müller. „Fast Indirect Lighting Approximations using the Representative Candidate Line Space“. In: *Journal of WSCG* 26 (Jan. 2018). DOI: 10.24132/JWSCG.2018.26.1.2.

[KLZ12] Pyarelal Knowles, Geoff Leach und Fabio Zambetta. „Efficient Layered Fragment Buffer Techniques“. In: *OpenGL Insights*. Hrsg. von Patrick Cozzi und Christophe Riccio. CRC Press, Juli 2012. Kap. 20, S. 279–292. ISBN: 978-1439893760.

[KML16] Kevin Keul, Stefan Müller und Paul Lemke. „Accelerating Spatial Data Structures in Ray Tracing through Precomputed Line Space Visibility“. In: Juni 2016.

[LK10a] Samuli Laine und Tero Karras. „Efficient Sparse Voxel Octrees“. In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: ACM, 2010, S. 55–63. ISBN: 978-1-60558-939-8. DOI: 10.1145/1730804.1730814.

[LK10b] Samuli Laine und Tero Karras. *Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation*. NVIDIA Technical Report NVR-2010-001. NVIDIA Corporation, Feb. 2010.

[McG17] Morgan McGuire. *Computer Graphics Archive*. Juli 2017. URL: https://casual-effects.com/data (besucht am 22. 04. 2019).

[Mea80] Donald Meagher. „Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer“. In: (Okt. 1980).

[Nvi16] Nvidia Corporation. *Nvidia GeForce GTX 1080. Gaming Perfected*. Mai 2016. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (besucht am 22. 04. 2019).

[Nvi19] Nvidia Corporation. *Nsight Visual Studio Edition*. Version 2019.1. 25. Feb. 2019. URL: https://developer.nvidia.com/tools-overview (besucht am 22. 04. 2019).

[RUL00] J. Revelles, C. Ureña und M. Lastra. „An Efficient Parametric Algorithm for Octree Traversal“. In: *Journal of WSCG*. 2000, S. 212–219.

[SA19] Mark Segal und Kurt Akeley. *The OpenGL Graphics System: A Specification. (Version 4.6 (Core Profile) - February 2, 2019)*. Version 2.0. Feb. 2019. URL: https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf (besucht am 22. 04. 2019).

[SH74] Ivan E. Sutherland und Gary W. Hodgman. „Reentrant Polygon Clipping“. In: *Commun. ACM* 17.1 (Jan. 1974), S. 32–42. ISSN: 0001-0782. DOI: 10.1145/360767.360802.

[Whi79] Turner Whitted. „An Improved Illumination Model for Shaded Display“. In: *SIGGRAPH Comput. Graph.* 13.2 (Aug. 1979), S. 14–. ISSN: 0097-8930. DOI: 10.1145/965103.807419.