

**HOCHSCHULE  
HANNOVER**  
UNIVERSITY OF  
APPLIED SCIENCES  
AND ARTS

—  
*Fakultät IV  
Wirtschaft und  
Informatik*

# **Raycasting-basiertes Voxelrendering**

Enes Hergül

Bachelor-Arbeit im Studiengang „Angewandte Informatik“

12. April 2022



**Autor** Enes Hergül  
153 10 96  
enes.hergul215@gmail.com

**Erstprüferin:** Prof. Dr. Frauke Sprengel  
Abt. Informatik der Fak. IV  
Hochschule Hannover  
frauке.sprengel@hs-hannover.de

**Zweitprüfer:** Prof. Dr. Volker Ahlers  
Abt. Informatik der Fak. IV  
Hochschule Hannover  
volker.ahlers@hs-hannover.de

### **Selbständigkeitserklärung**

Hiermit erkläre ich, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Hannover, den 12. April 2022

Unterschrift

## Zusammenfassung

Die Arbeit hat das Ziel, Volumendaten effizient zu visualisieren, ohne diese in ein alternatives Repräsentationsformat wie Polygonnetze zu überführen. Dafür werden Voxel sowie ein Raycasting-Ansatz verwendet. Die Elemente des Volumens werden als Voxel (=volumetrische Pixel), das Volumen nachfolgend als ein dreidimensionales Array aus Voxel bezeichnet. Für jeden Pixel wird ein Strahl erzeugt, der das Array iterativ traversiert, wobei in jeder Iteration geprüft wird, ob das gegebene Volumenelement ein Datum hält oder nicht. Maßgeblich hierfür ist der von (Amanatides, Woo et al., 1987) vorgestellte Ansatz, den Strahl in solche Segmente zu zerlegen, die immer jeweils so lang wie der momentan zu untersuchende Voxel groß ist: In einer Iteration wird also immer genau ein Voxel untersucht. Die Arbeit wird diese Idee verfolgen, praktisch aber anders umsetzen und sie um eine zusätzliche Beschleunigungsstruktur ergänzen. Im Idealfall soll dieser Ansatz es erlauben, mindestens  $512^3$  Voxel in Echtzeit zu rendern. Der beschriebene Ansatz hat zusätzlich den Vorteil, dass Änderungen direkt sichtbar werden, weil für das anschließende Rendering, auf die modifizierten Daten zurückgegriffen wird.

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>7</b>
1.1 Motivation . . . . .	7
1.2 Historie . . . . .	8
1.3 Problem und Zielstellung . . . . .	9
1.4 Struktur der Arbeit . . . . .	12
<b>2 Grundlagen</b>	<b>13</b>
2.1 Volumen . . . . .	13
2.2 Voxels . . . . .	13
2.3 Beschleunigungsstrukturen . . . . .	15
2.3.1 Bounding-Volumes . . . . .	15
2.3.2 N-Trees: Quadtrees und Octrees . . . . .	17
2.3.3 Verwendung beim Raycasting . . . . .	19
2.4 Hardwarebeschleunigung . . . . .	19
2.4.1 CPU, SIMD und GPU . . . . .	20
2.4.2 DirectX und Monogame . . . . .	22
2.4.3 Shader . . . . .	23
2.5 Raycasting-Grundlagen . . . . .	23
2.5.1 Abgrenzung . . . . .	24
2.5.2 Kamera . . . . .	25
2.5.3 Ray-AABB-Intersection . . . . .	28
<b>3 Raycasting: Theoretische Grundlagen</b>	<b>31</b>
3.1 Sparse-Voxel-Octrees . . . . .	31
3.1.1 Verwendung beim Raycasting . . . . .	31
3.1.2 Vorherige Arbeit: Efficient Sparse Voxel Octrees (NVIDIA Research) . . . . .	32
3.1.3 Eigener Ansatz: Sparse-Voxel-Octree . . . . .	33
3.1.4 Überblick: Algorithmus und Pseudocode . . . . .	35
3.1.5 Bewertung: Sparse-Voxel-Octrees . . . . .	36
3.2 Voxel-Volume-Raycasting . . . . .	36
3.2.1 Vorherige Arbeit: A Fast Voxel Traversal Algorithm for Ray Tracing . . . . .	37
3.2.2 Eigener Ansatz: Voxel-Volume-Raycasting . . . . .	39
3.2.3 Bewertung: Voxel-Volume-Raycasting . . . . .	41
3.3 Beschleunigtes Voxel-Volume-Rendering . . . . .	42

<b>4 Implementierung</b>	<b>43</b>
4.1 Implementierung: Ray-AABB-Schnittpunktberechnung . . . . .	43
4.1.1 Vorbemerkung zur Syntax . . . . .	43
4.1.2 Code . . . . .	44
4.1.3 Erläuterung . . . . .	44
4.2 Implementierung: Voxel-Volume-Raycasting . . . . .	45
4.2.1 Code . . . . .	45
4.2.2 Erläuterung . . . . .	47
4.2.3 Komplexitätsanalyse . . . . .	50
4.2.4 Divergenzanalyse . . . . .	51
4.3 Erzeugung einer Octree in der GPU . . . . .	51
4.3.1 Code . . . . .	51
4.3.2 Erläuterung . . . . .	53
4.3.3 Komplexitätsanalyse . . . . .	56
4.4 Rekursionsfreie Traversierung eines Octree . . . . .	57
4.4.1 Code . . . . .	58
4.4.2 Erläuterung . . . . .	60
4.4.3 Komplexitätsanalyse . . . . .	64
4.4.4 Divergenzanalyse . . . . .	65
4.5 Denoising-Strategie: Medianfilter als Postprocessing-Shader . . . . .	66
4.5.1 Median-Filter . . . . .	66
4.5.2 Code . . . . .	66
4.5.3 Erläuterung . . . . .	67
4.5.4 Komplexitätsanalyse . . . . .	68
<b>5 Ergebnisse und Diskussion</b>	<b>69</b>
5.1 Ergebnisse . . . . .	69
5.1.1 Blockartefakte und Rauschreduktion . . . . .	69
5.1.2 Beschleunigungsstruktur . . . . .	70
5.1.3 Raytracing . . . . .	72
5.2 Performance . . . . .	73
5.2.1 Hardwarespezifikation . . . . .	73
5.2.2 Tabellen . . . . .	74
5.3 Diskussion: Probleme und Lösungsansätze . . . . .	75
5.3.1 Raycasting ohne Beschleunigungsstruktur . . . . .	75
5.3.2 Raycasting mit Beschleunigungsstruktur . . . . .	75
5.3.3 Redundanz . . . . .	76
5.3.4 Beleuchtungsprobleme . . . . .	77
5.3.5 Texturemapping . . . . .	78
5.3.6 Schatten . . . . .	78
5.3.7 Flackern bei Modifikation zusammenhängender Voxel-Daten . . . . .	79

*Inhaltsverzeichnis*

---

<b>6 Fazit und Ausblick</b>	<b>81</b>
Literatur . . . . .	83

# 1 Einleitung

In diesem Kapitel soll die Motivation der Bachelor-Arbeit erklärt, das Thema historisch eingeordnet, darauffolgend grundlegende Probleme konventioneller Renderingtechniken erklärt, das angestrebte Ziel dieser Arbeit erörtert und die Struktur der Arbeit dargelegt werden.

## 1.1 Motivation

Kennzeichnend für das 21. Jahrhundert ist insbesondere der technologische Fortschritt, der sich in der fortschreitenden nationalen sowie internationalen Vernetzung zuerst von Computern, dann von Mobilgeräten und künftig sogar von Fahrzeugen, aber auch in der visuellen Darstellung digitaler Inhalte in Form von interaktiven Programmen, Virtual-Reality und Videospielen niederschlägt.

Im Allgemeinen nutzen Videospiele für die Darstellung dreidimensionaler Modelle sogenannte Polygone. Dies liegt unter anderem - wie im nächsten Abschnitt weiter ausgeführt - historisch darin begründet, dass die ersten Grafikkarten auf die Darstellung von Polygonen (besonders Dreiecke) ausgelegt waren. Dabei wird ein dreidimensionales Modell in Simplizes (häufig Dreiecke) zerlegt, die Ecken (Vertizes) jedes Simplex' der Grafikkarte übergeben, dort transformiert und anschließend gerastert. Dieser Prozess wird unter dem Begriff „Rendering-Pipeline“ subsumiert.

Wichtige Bestandteile der Rendering-Pipeline sind sogenannte Shader. Shader sind programmierbare Module, die von der Grafikkarte in korrespondierenden Pipeline-Stufen mit entsprechenden Parametern aufgerufen werden. Im Shader-Modul können die Parameter beliebig modifiziert und die Ergebnisse anschließend in die nächste Pipeline-Stufe übergeben werden.

Zwei wichtige solcher Shader sind der sogenannte Vertex-Shader, der die Ecken (sogenannte Vertizes) eines Dreiecks für die Weiterverarbeitung manipuliert sowie der Pixel-Shader, der während der Rasterisierung für jeden Pixel/jedes Fragment aufgerufen wird.

Grundlegendes Charakteristikum von auf Polygonnetzen basierenden Modellen ist, dass diese „hohl“ sind. Konkret heißt das, dass Polygonmodelle nur die äußere Form eines

volumetrischen (also dreidimensionalen) Modells abbilden. Das ist auch insofern plausibel, weil die Grafikkarte auf diese Weise keine Daten prozessieren muss, die prinzipiell nicht sichtbar sind.

Problematisch ist dies aber deshalb, weil dadurch volumetrische Informationen nicht adäquat erfasst werden können, Nutzerinteraktion mit dreidimensionalen Modellen also nur beschränkt möglich, Zerstörung von Komponenten des Modells praktisch unmöglich ist. Liegt exemplarisch ein dreidimensionales Terrain als Polygontopographie vor, kann dieses Terrain nicht beliebig verwüstet, weiter ausgebaut, oder grundlegend verändert werden, sodass das Spiel im schlimmsten Fall starr, streng mechanisch und gar langweilig wirkt.

Es gilt daher, nachfolgend einen Ansatz zu erarbeiten, das volumetrische Daten effizient visualisiert, statt nur die Oberfläche dieser mittels Polygone anzunähern. Ein solcher Ansatz wäre, volumetrische Daten in einem dreidimensionalen Array von Voxel abzuspeichern und diese auf Grundlage der Raycasting-Technik zu rendern.

## 1.2 Historie

Während noch in den 1940er Jahren Computer ganze Räume füllten, erschien mit dem Beginn von Apple I die Nutzung von Computern im Alltag immer greifbarer, was schließlich in die Serienproduktion sogenannter Heimcomputer kulminierte und damit eine wichtige Grundlage für den in den 1990er Jahren beginnenden digitalen Zeitalters gelegt wurde.

Schon bald war ein wichtiger Bestandteil des Heimcomputers der Bildschirm, der visuelle Elemente darstellen und die Nutzerinteraktion mit dem Computer erleichtern sollte. Nachfolgende Generationen von Bildschirmen ermöglichen es, Animationen darzustellen, bessere Prozessoren schließlich die Echtzeitinteraktion, sodass erste Spiele realisiert werden konnten.

In den 1970er Jahren konnten Spiele zunächst nur durch primitive Grafikelemente wie Linien, oder Rechtecke schmeicheln. Weniger als ein Jahrzehnt später aber, wurde mit dem Videospiel für Arcade-Automaten „I, Robot“ (“Game That Makes Full Use of 3D CG – Future Space War: Interstellar from Funai.”, 1983) Polygone eingeführt, die die Darstellung dreidimensionaler Objekte in Echtzeit ermöglichte.

Das Interesse an Polygonen stieg insbesondere in der sogenannten „fünften Generation von Videospielen“, die um das Jahr 1993 begann, und ungefähr 2001/2 endete (Medin & Persson, 2009).

Die (als solche vermarktete) erste Grafikkarte, war die Geforce 256 von Nvidia (1999), die eigenen Angaben zufolge „zehn Millionen Polygone in einer Sekunde“ gleichzeitig verarbeiten könne (*Graphics Processing Unit (GPU)*, o. J.). Nachfolgend waren alle Grafikkarten so ausgelegt, dass diese besonders viele Vertizes (grob: Ecken eines Dreiecks)

parallel prozessieren und Dreiecke besonders schnell rendern konnten. Rendering bezeichnet in diesem Kontext die Rasterisierung von Primitiven, besonders Dreiecke, bei der seit jeher der sogenannte Scanline-Algorithmus zum Einsatz kommt (Bouknight, 1970).

In der Videospiel-Branche erfuhren Voxel zunächst wenig Beachtung (wobei es vereinzelt doch zum Einsatz kam, etwa im Spiel „Terra Nova: Strike Force Centauri“ (1996)), dafür aber mehr in der Medizin, etwa bei der Visualisierung von durch das MRT-Verfahren ermittelte Daten. In diesem Zusammenhang leisteten Hanspeter Pfister et. al. (1999) einen wichtigen Beitrag zum Rendering von Voxel. In seiner Arbeit (Pfister, 1999) wird ein für das auf Raycasting basierende Rendering von Voxels ausgelegte Hardware-Einheit vorgestellt, die 256<sup>3</sup> Voxel bei 30 FPS (frames per seconds) gleichzeitig auf den Bildschirm bringen könne.

Besonders bekannt wurden Voxel durch „Minecraft“ (2011), welches sich bis heute noch hoher Beliebtheit erfreut (Hofmann, 2019). Minecraft verwendet für die Darstellung aber keine Raycasting-Technik, sondern Polygone.

Mit dem Spiel „Teardown“ (2020) war das Potential von Voxel, insbesondere in der Videospielindustrie, greifbarer geworden: Nicht nur sind die Spielwelten und Entitäten zerstörbar, sie werden auch physikalisch simuliert, indem etwa ein Hausdach einstürzt, wenn die Balken zerstört werden. Teardown verwendet für die Visualisierung von Voxel-Daten den Ansatz von (Amanatides et al., 1987), an den diese Arbeit auch anknüpfen, aber praktisch anders umsetzen wird.

## 1.3 Problem und Zielstellung

Prinzipiell haben auf Polygone basierende Modelle den Vorteil, Deformationen gut darstellen zu können. Ein Polygon ist im Grunde ein geschlossener Streckenzug (=Polygonzug), der durch ein Satz von Punkten festgelegt wird. Die Punkte dieses Streckenzugs können als Ecken interpretiert werden, sodass die Form des Polygons allein von der Lage der Punkte abhängt. Wird eine Ecke (also ein Punkt) des Polygons (=Streckenzugs, Polygonzugs) beliebig verschoben, passt sich die Form des Polygons entsprechend an.



(a) Querschnitt eines dreidimensionalen Modells.  
(b) Die Silhouette des Querschnitts

Abbildung 1.1: Die Abbildung rechts stellt die Silhouette der Abbildung links - Polygonnetze setzen genau dort an: Diese haben nämlich das Ziel, die (dreidimensionale) Oberfläche (oder hier: die zweidimensionale Silhouette) eines volumetrischen Objekts mit beliebig vielen Simplizes (etwa Dreiecke, oder Linien) anzunähern. Volumetrische Eigenschaften werden dabei ignoriert.

Die Silhouette (b) kann durch eine beliebige Anzahl von Linien angenähert werden, erfasst damit aber grundsätzlich keine „volumetrischen“ Informationen (Abbildung links (a)). Polygone ermöglichen es also, die Oberfläche eines Körpers (oder etwa die Silhouette einer Figur) mit einfachen geometrischen Formen (Simplizes: Dreiecke, Linien) darzustellen. Soll das Modell aber zerstört, volumetrische Querschnitte dieser betrachtet oder beliebig erweitert werden, führt an alternativen Darstellungsmöglichkeiten volumetrischer Daten kein Weg vorbei.

Es existieren zwar eine Reihe von Möglichkeiten, Oberflächen von Volumendaten ermitteln und diese zu triangulieren, etwa Marching Cubes (Lorensen & Cline, 1987), das Isoflächen (griechisch *íisos*, gleich: Flächen also, die gleiche (oder ähnliche) Eigenschaften von Punkten im Raum zusammenfassen) mit Polygon-Netzen anzunähern versucht, indem der Algorithmus durch das Volumen iteriert, in jeder Iteration jeweils an acht Ecken die Volumendaten evaluiert und ausgehend davon jeweils Polygone generiert, die eine solche Isofläche (und damit die Oberfläche des Volumens) repräsentieren sollen. Problematisch an diesem Ansatz ist aber, dass zwei verschiedene Repräsentation derselben Daten vorliegen: Die Volumendaten auf der einen und das Polygonnetz, das die Oberfläche dieser annähert, auf der anderen Seite. Wird das Volumen modifiziert, muss unter Umständen das gesamte Polygonnetz neu generiert werden, was viele Rechenzyklen und Transfer von Daten von der CPU zur GPU zur Folge hat.

Grundsätzlich gilt es, in jedem Frame so wenig Daten wie nur möglich von der CPU zur GPU zu transferieren.

Dies liegt einerseits in dem sogenannten Von-Neumann-Flaschenhals (Zou, Xu, Chen, Yan & Han, 2021) begründet: Auf das Bus-System kann immer nur eine Hardware-Einheit schreibend zugreifen (*2.1.2.2 von-Neumann-Flaschenhals*, o. J.). Werden zu viele Daten geschoben, wirkt sich das auf das Laufzeitverhalten des Programms negativ aus, auch deshalb, weil die Bandbreite des Bus-Systems wesentlich geringer ist, als die Geschwindigkeiten mit der die CPU und GPU arbeiten. Anderseits ist das Rendern von Polygonnetzen an sogenannte **Draw()-Calls** gekoppelt. Analog gilt es, die Anzahl dieser in jedem Frame zu reduzieren und Gebrauch vom sogenannten **Batching** zu machen (Wloka, 2003). Die Gründe dafür sind vielschichtig, besonders aber liegt es daran, dass Grafikkarten Batches schneller verarbeiten können, als die CPU einen Batch (das was zu Rendern ist) vorbereiten kann. Die GPU liegt bei vielen kleinen Batches häufig also (unnötig) brach, während die CPU bei der Vorbereitung vieler Batches hinterherhinkt.

Die Arbeit wird sich im Folgenden also insbesondere mit der Frage beschäftigen, wie Volumendaten zur Laufzeit **fortlaufend** modifiziert und gleichzeitig für das anschließende Rendering **effizient** zur Verfügung gestellt werden können. Effizient impliziert hierbei hauptsächlich die Lokalität der Daten. Das heißt, dass das Rendering genau dann effizient ist, wenn die Daten nicht erst in verschiedene Repräsentationsformate überführt werden müssen, sondern die **schreibende** Einheit (exemplarisch der Physik-Solver) und die **lesende** Einheit (exemplarisch der Renderer) auf die selben Daten zugreifen.

Eine möglicher Lösungsansatz lässt sich direkt aus der Abbildung 1.1 ableiten: Die Daten liegen im Grunde bereits als ein Array von Bildelementen (Pixels) vor und können damit ohne alternative Repräsentationsformate direkt modifiziert und visualisiert werden. Die Figur in der Abbildung korrespondiert aber mit einem zweidimensionalen Querschnitt eines dreidimensionalen Körpers. Die Arbeit erweitert diese Idee und führt deshalb volumetrische Pixel ein, kurz: Voxel. Volumetrische Objekte werden im Speicher also, analog zu Bildern, die im Speicher als ein Array von Pixel vorliegen, als ein Array von Voxel abgespeichert. Das Ziel der Arbeit ist es jetzt, Voxeldaten (also die Rohdaten volumetrischer Objekte) effizient zu visualisieren und damit den Datentransfer zwischen CPU und GPU auf ein Minimum zu reduzieren.

Ist das Array von Voxel (exemplarisch im Form von dreidimensionalen Texturen) einmal in den Grafikspeicher geladen worden, kann der in dieser Arbeit vorgestellte Ansatz genutzt werden um a) Änderungen im Volumen in einem Shader-Modul wie dem Compute-Shader durchzuführen (hierbei findet praktisch gar kein Datentransfer statt), oder um b) kleine Bereiche des Volumens CPU-seitig zu modifizieren und dann zu übertragen (was ebenfalls unproblematisch ist, solange die Anzahl der Änderungen nicht wesentlich größer sind, als das Bus-System Daten übertragen kann).

Der letzte Punkt ist besonders vor dem Hintergrund zu betrachten, dass Arrays initial redundant sind, diese Redundanz aber auch vorteilhafte Eigenschaften hat: Jedes Arrayelement (=Voxel) kann exemplarisch als Integer-Wert (32 Bits, 4 Bytes), oder gar

als ein einzelnes Byte (8 Bits) dargestellt werden. Die Position muss nicht explizit abgespeichert werden, weil diese bereits implizit in der Art und Weise wie die Elemente des Arrays angeordnet sind, vorliegt. Zudem ist die Größe eines Voxels uniform und konstant, sodass diese ebenso nicht abgespeichert werden muss.

Bei einem Polygon-Ansatz hingegen, muss nach jeder Änderung das Polygonnetz neu generiert, also ein neuer Vertex-Buffer instanziert und die neuen Vertizes zur Grafikkarte transferiert werden, wobei die Ecke jeweils eines Dreiecks allein mindestens 96 (drei Floats für eine Vektor-Struktur =  $3 \cdot 32$  Bits um die Position anzugeben) Bits groß ist.

## 1.4 Struktur der Arbeit

Im nachfolgenden Kapitel **Grundlagen**, werden Volumen sowie Voxels präzisiert, Beschleunigungsstrukturen eingeführt und Raycasting-Grundlagen erörtert. Darauf folgt das Kapitel **Raycasting: Theoretische Grundlagen**, in das für die Implementierung notwendigen **theoretischen** Grundlagen gelegt und damit insbesondere die zu realisierenden Ziele konkretisiert werden. Den Kern der Arbeit bildet das Kapitel **Implementierung**, in welches ein eigener Raycasting-Ansatz implementiert, diese um eine Beschleunigungsstruktur ergänzt und eine Denoising-Strategie gezeigt wird. Auf das Kapitel **Implementierung** folgt das Kapitel **Ergebnisse und Diskussion**, wo Ergebnisse gezeigt, wichtige Querschnittsdaten tabellarisch dargestellt und Problematiken sowie Lösungen diskutiert werden. Im abschließenden Kapitel **Fazit und Ausblick** werden die Ergebnisse zusammengefasst, bewertet ob die Ziele angemessen umgesetzt wurden und künftige Einsatzmöglichkeiten in Ausblick gestellt.

# 2 Grundlagen

In diesem Kapitel sollen die Grundlagen der Arbeit erarbeitet werden. Zuerst sollen Volumina und Voxel mathematisch präzisiert, Beschleunigungsstrukturen präsentiert und anschließend Raycasting-Grundlagen erörtert werden.

## 2.1 Volumen

**Definition 1** (Volumen, Volumenelement). *Mit Volumen wird ein abgeschlossenen Quader im  $\mathbb{R}^3$ , also  $V \subset \mathbb{R}^3$  bezeichnet,*

$$V = \{(x_1, x_2, x_3) \in \mathbb{R}^3 \mid \text{left} \leq x_1 \leq \text{right}, \text{bottom} \leq x_2 \leq \text{top}, \text{front} \leq x_3 \leq \text{back}\}.$$

*Gegeben sei ferner eine multivariate Funktion  $f : V \rightarrow \mathbb{R}$ , die Volumenkoordinaten auf ein Skalar abbildet.*

*Die Werte (Skalare) der Funktion  $f$  an den Stellen  $\vec{x} \in V$  werden als Volumenelemente bezeichnet.*

Die Definition mag insofern irreführend wirken, dass der Begriff **Volumenelement** den Anschein weckt, dass damit die Elemente des als abgeschlossene Teilmenge des  $\mathbb{R}^3$  definierten Volumens gemeint sind. In der Arbeit ist dies aber keineswegs gemeint: Der Begriff Volumenelement bezeichnet hier die Werte der Funktion  $f$  an der Stelle  $\vec{x} \in V, V \subset \mathbb{R}^3$ . Der Skalar kann hierbei für alle möglichen Größen wie etwa Temperatur, Dichte, oder dergleichen stehen. Die Differenzierung ist wichtig, weil die Arbeit Voxel über die multivariate Funktion  $f$  definieren wird.

## 2.2 Voxels

**Definition 2** (Minimum/Maximum in  $\mathbb{R}$ ). *Für eine abgeschlossene und beschränkte Menge  $M \subset \mathbb{R}$  ist das Minimum bzw. Maximum wie gewöhnlich definiert als*

$$\min M = \min_{x \in M} x,$$

$$\max M = \max_{x \in M} x.$$

Um Voxels präziser zu definieren, muss die obige Definition noch für Mengen  $V \subset \mathbb{R}^n$  angepasst werden, weil auf  $n$ -Tupel keine Ordnungsrelation definiert ist. Dafür werden die Funktionen min sowie max für abgeschlossene Mengen  $V \subset \mathbb{R}^n$  komponentenweise anwenden.

**Definition 3** (komponentenweises Minimum/Maximum in  $\mathbb{R}^3$ ). *Für eine abgeschlossene und beschränkte Menge  $M \subset \mathbb{R}^3$  ist das komponentenweise Minimum bzw. komponentenweise Maximum definiert als*

$$\min M = (\min_{\vec{x} \in M} x_1, \min_{\vec{x} \in M} x_2, \min_{\vec{x} \in M} x_3),$$

$$\max M = (\max_{\vec{x} \in M} x_1, \max_{\vec{x} \in M} x_2, \max_{\vec{x} \in M} x_3).$$

**Definition 4** (Ab-/Aufrundungsfunktion in  $\mathbb{R}$ ). *Außerdem benötigt werden die Abrundungsfunktion  $\lfloor \cdot \rfloor : \mathbb{R} \rightarrow \mathbb{Z}$*

$$\lfloor x \rfloor = \max\{z \in \mathbb{Z} \mid z \leq x\}$$

*und die Aufrundungsfunktion  $\lceil \cdot \rceil : \mathbb{R} \rightarrow \mathbb{Z}$*

$$\lceil x \rceil = \min\{z \in \mathbb{Z} \mid z \geq x\}.$$

Auch diese Definition muss für Elemente der Menge  $V \subset \mathbb{R}^n$  angepasst werden, indem die Funktion  $\lfloor \vec{x} \rfloor$  wieder komponentenweise definiert wird.

**Definition 5** (komponentenweises Ab-/Aufrunden in  $\mathbb{R}^3$ ). *Komponentenweises Ab-/Aufrunden ist dann definiert als*

$$\lfloor \vec{x} \rfloor = (\lfloor x_1 \rfloor, \lfloor x_2 \rfloor, \lfloor x_3 \rfloor),$$

$$\lceil \vec{x} \rceil = (\lceil x_1 \rceil, \lceil x_2 \rceil, \lceil x_3 \rceil).$$

Schließlich können Voxel mathematisch präzisiert werden:

**Definition 6.** *Ein Voxel bezeichnet die Auswertung der Funktion  $f: V \rightarrow U$  an äquidistanten Stellen  $\vec{i}$ , wobei  $\vec{i} \in T$  und  $T \subset \mathbb{N}^3$  ist, spezifischer  $T = \{\vec{i} \in \mathbb{N}^3 \mid \lceil \min V \rceil \leq \vec{i} \leq \lfloor \max V \rfloor\}$ . Insgesamt bezeichnen Voxel also die Menge  $M = \{f(\vec{i}) \mid \vec{i} \in T\}$ .*

Mit  $\min V$  und  $\max V$  werden jeweils die komponentenweise minimalen und maximalen Raumkoordinaten (Elemente also) des  $V$  bezeichnet, die jeweils auf ein Integer auf -beziehungsweise abgerundet werden. Dies liegt darin begründet, dass jeweils der nächst größere Integer des Minimums und der nächst kleinere Integer des Maximums des Volumens, in jedem Fall im Volumen liegt. Das Minimum des Volumens  $V$  ist der vordere, linke und untere Eckpunkt ( $\text{Corner}_{min}$ ), wobei analog das Maximum des Volumens der hintere, rechte und obere Eckpunkt ( $\text{Corner}_{max}$ ) ist.

Es sind alternative Definitionen möglich, etwa dass ein Voxel selbst wiederum über eine Aggregatfunktion  $F(\{X_i\})$  definiert ist, die eine Menge  $X_i$  von Volumenelementen in einem Bereich fester Größe auf einen Skalar abbildet. Eine solche Funktion  $F(\{X_i\})$  kann exemplarisch der Durchschnitt, oder das Maximum der Menge  $X_i$  sein.

Die obige Definition eines Volumen lässt Volumengrenzen kleiner als eins zu. Dies ist insofern problematisch, weil der Begriff Voxel über die ganzen Zahlen definiert wurde. Es liegt also in diesem Kontext nahe, Volumengrenzen ganzzahlig zu definieren, oder existierende Volumina so zu skalieren, dass sie der in dieser Arbeit vorgestellten Definition von Voxel genügen. Die Arbeit wird sich insbesondere mit Volumina der Größe  $512^3$  befassen.

## 2.3 Beschleunigungsstrukturen

**Definition 7.** *Mit Beschleunigungsstrukturen (engl. Accelerator Structures) werden solche Strukturen bezeichnet, die beliebige geometrische Figuren durch einfache Formen annähern, um Operationen auf diese zu beschleunigen.*

Die einfachste Form einer solchen Beschleunigungsstruktur sind sogenannte Bounding-Volumes.

### 2.3.1 Bounding-Volumes

**Definition 8.** *Gegeben sei eine Figur  $F_1$ , dann ist ihr Bounding-Volume dasjenige Volumen, dessen kleinste Ecke mit den komponentenweise kleinsten Koordinate der Figur sowie analog die größte Ecke mit den komponentenweise größten Koordinaten der Figur korrespondiert.*

Die beiden Eckpunkte, die diagonal zueinander liegen, werden jeweils  $\text{Corner}_{min}$  und  $\text{Corner}_{max}$  bezeichnet. Bezeichnet  $U \subset \mathbb{R}^3$  die Menge aller Punkte, die die Figur  $F_1$  festlegt, dann lässt sich  $\text{Corner}_{min}$  (und  $\text{Corner}_{max}$  analog) mit dem komponentenweise definierten  $\min U$  berechnen.

Ein Bounding-Volume (oder Bounding-Rectangle) umfasst demnach also eine beliebige Figur wie folgt:

## 2 Grundlagen

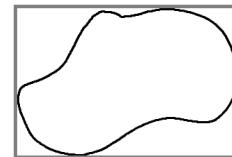


Abbildung 2.1: Hier: Bounding-Rectangle, das eine komplexe Figur  $F_1$  umfasst.

Möchte man exemplarisch die Kollision zweier komplexer Figuren berechnen, wäre es ineffizient, die Figuren zu Beginn auf eine Kollision zu prüfen. Es liegt daher nahe erst zu ermitteln, ob sich zwei komplexe Figuren überhaupt ansatzweise schneiden: Dafür genügt es zu prüfen, ob das Bounding-Volume der Figur 1 eine Überschneidung mit dem Bounding-Volume der Figur 2 hat. Erst wenn dies zutrifft, können komplexere Kollisionsberechnung erfolgen. Auf diese Weise werden wichtige Rechenzyklen gespart.

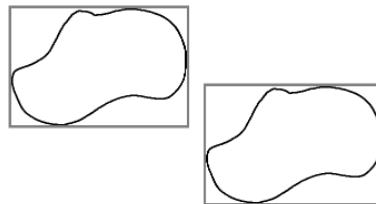


Abbildung 2.2: Hier: Bounding-Rectangles der Figuren überschneiden sich nicht, demnach müssen auch keine komplexen Kollisionsberechnungen durchgeführt werden.

Ein wesentliches Problem dieser Beschleunigungsstruktur ist die fehlende Präzision. Wie in der Abbildung 2.2 sichtbar, werden auch Bereiche vom Bounding-Volume umfasst, die über keine Punkte der Figur verfügen.

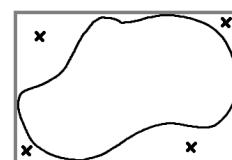


Abbildung 2.3: Hier: Bounding-Rectangle, das die Figur zwar komplett, nicht aber präzise umfasst.

Eine Datenstruktur, die die Idee des Bounding-Volumes erweitert, höhere Präzision liefert und Operationen auf Figuren ebenso beschleunigt, sind sogenannte N-Trees, die nachfolgend behandelt werden sollen.

### 2.3.2 N-Trees: Quadtrees und Octrees

Ein N-Tree ist die Verallgemeinerung von Binärbäumen, Quadtrees und Octrees. Gemein ist, dass ein Elternknoten jeweils  $2^d$  Kindknoten hat. Die Ganzzahl  $d$  bezeichnet dabei die Dimension (Binärbäume  $d = 1$ , Octrees  $d = 3$ , et cetera).

Im Kontext der Computergrafik, wird diese Datenstruktur unter anderem durch die oben gezeigte Abbildung 2.4 motiviert. Die Abbildung illustriert die Problematik, dass das Bounding-Rectangle (Erklärung gilt analog für Bounding-Volumes) auch solche Bereiche umfasst, die keinen Informationsgehalt zur Figur liefern.

Es liegt also nahe, den Bounding-Rectangle in der Mitte horizontal sowie vertikal zu halbieren:

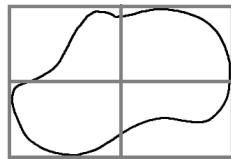


Abbildung 2.4: Hier: Bounding-Rectangle wird in der Mitte vertikal und horizontal halbiert.

Wieder existieren Bereiche, die keine Punkte der Figur umfassen, sodass die vier Quadranten jeweils nochmal halbiert werden:

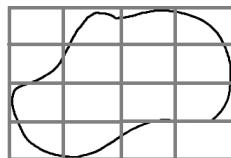


Abbildung 2.5: Die vier Quadranten wurden in vier gleichgroße, kleinere Quadranten geviertelt.

Der Abbildung ist zu entnehmen, dass zwei Quadranten keine Punkte der Figur umfassen: Bei einer vereinfachten Kollisionserkennung, können jetzt sogenannte „false positives“ besser ausgeschlossen werden.

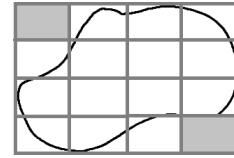


Abbildung 2.6: Die zwei Quadranten, die keine Teilmenge der Menge  $U$ , die die Figur  $F_1$  bildet, umfassen, wurden ausgegraut: Diese können bei der vereinfachten Kollisionserkennung ignoriert werden.

Iteriert man diesen Halbierungsprozess  $n - mal$ , so entsteht ein Baum der Tiefe  $n$ , wobei jeder Elternknoten, den die Figur umfasst, jeweils vier Kindknoten hat:

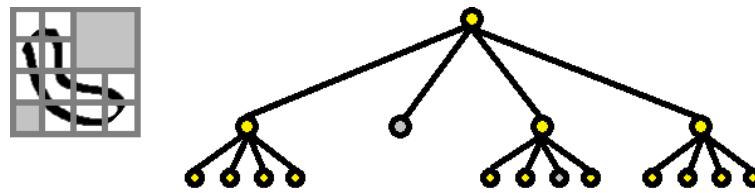


Abbildung 2.7: Die Unterteilung der Quadranten und die dazugehörige Quadtree-Struktur der Tiefe  $n = 2$ . Gelb kennzeichnet, dass im korrespondierenden Quadranten Daten enthalten sind. Sind keine Daten enthalten, wird jener Quadrant nicht weiter unterteilt.

Analog gilt für Octrees, dass ein Volumen (hier: Würfel) in acht Teilvolumen unterteilt wird, die jeweils ein Achtel so groß sind wie ihr Elternknoten.

Ein Satz der im Verlauf der Arbeit noch wichtig sein wird, ist folgender:

**Satz 1.** Sei  $s$  die initiale Größe des Würfels, den es zu unterteilen gilt,  $n = \log_2 \frac{s}{\varepsilon}$  die Tiefe des Baumes und  $\varepsilon \in \{2a: a \in \mathbb{N}\}$  die minimale Größe eines Teilvolumens, dann beträgt die Gesamtzahl der Knoten im Octree  $\frac{1-8^{n+1}}{1-8}$ .

*Beweis.* Die maximale Tiefe des Octree lässt sich algebraisch leicht herleiten, indem man  $\frac{s}{2^n} = \varepsilon$  nach  $n$  umstellt. Die Gesamtzahl der Knoten im Octree beträgt

$$\sum_{i=0}^n 8^i = \frac{1-8^{n+1}}{1-8}.$$

Dies entspricht offensichtlich der endlichen geometrischen Reihe:  $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q}$ .  $\square$

Der Satz gilt für Quadtrees und Binäräbäume für jeweils  $q = 4$  und  $q = 2$  analog.

### 2.3.3 Verwendung beim Raycasting

Octrees werden in dieser Arbeit motiviert, weil für wachsendes  $s$ , die Laufzeit um denjenigen Knoten zu ermitteln, der von einem Strahl geschnitten wird, nur logarithmisch wächst.

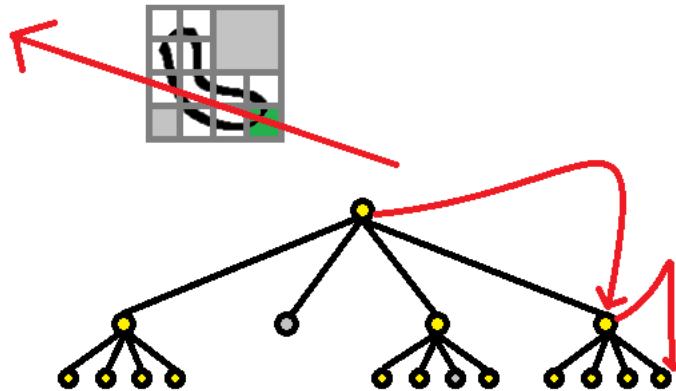


Abbildung 2.8: Der Strahl trifft den unteren, rechten Quadranten: Obschon insgesamt 17 Quadranten existieren, sind lediglich zwei Traversierungen und maximal acht Iterationen notwendig, um diesen zu ermitteln.

**Satz 2.** Sei  $s$  die initiale Größe des Quad-, oder Octree, dann liegt die Laufzeit um denjenigen Knoten zu ermitteln, der als erstes von einem Strahl geschnitten wird, in  $O(\log s)$ .

*Beweis.* Sei  $s$  die initiale Größe des Quadtree und bezeichne  $\varepsilon$  ferner die minimale Größe eines Quadranten, dann beträgt die Tiefe der Quadtree  $n = \log_2 \frac{s}{\varepsilon}$ . In jeder Tiefe  $n_i$  müssen bei einem Quadtree maximal jeweils vier Knoten überprüft werden. Um schließlich den finale Knoten in der letzten Tiefe zu ermitteln, sind also insgesamt  $4 \cdot n$  Iterationen notwendig. Daraus folgt dann  $O(4 \cdot n) = O(4 \cdot \log_2 \frac{s}{\varepsilon}) = O(\log \frac{s}{\varepsilon}) = O(\log s)$ .  $\square$

Der Beweis gilt für Octrees analog.

## 2.4 Hardwarebeschleunigung

Der Begriff **Hardwarebeschleunigung** bezeichnet die Verwendung zusätzlicher Hardware (meist GPU), um die CPU zu entlasten und bestimmte Operationen auf Daten parallel zur CPU auszuführen. Auf diese Weise können Aufgaben logisch (etwa exemplarisch in Interaktion (CPU) und Darstellung (GPU)) getrennt werden.

Verschiedene Programmierschnittstellen (besonders DirectX und OpenGL) erlauben es dem Programmierer die GPU anzusprechen, Daten zwischen GPU und CPU zu transferieren und gewisse Operationen wie den im ersten Kapitel bereits angeschnittenen Draw()-Call anzustoßen.

### 2.4.1 CPU, SIMD und GPU

Um die Rolle von GPUs im Computer besser einordnen zu können, muss erst verstanden werden, welche Typen von Aufgaben besonders geeignet sind, um sie auf die Grafikkarte auszulagern: In diesem Kontext spielt der Begriff „SIMD“ (Single Instruction Multiple Data) eine wichtige Rolle. Soll auf  $n$  unterschiedliche Daten dieselbe Operation ausgeführt werden (wie exemplarisch eine Addition), dann liegt es nahe, statt  $n$  Additionen, jeweils eine Addition auf verschiedenen Daten gleichzeitig (=parallel) auszuführen.

Architekturen, die eine solche Realisierung erlauben, werden unter dem Begriff „Vektorprozessor“ subsumiert. Soll die CPU angewiesen werden, zwei Zahlen zu addieren, so müssen die Daten (hier z.B.: Floats, also 32 Bit) zuerst in ein Register geladen werden, um auf diese anschließend die ADD Instruktion auszuführen. Dies ist bei vielen konsekutiven Additionen ineffizient, weshalb moderne CPUs über weitaus größere Register (unter anderem 128-Bit, 256-Bit, oder gar 512-Bit) verfügen. Bei CPUs mit 256-Bit Registern können theoretisch also acht Floats in jeweils zwei 256-Bit Register geladen und auf diese beiden Register **gleichzeitig** die ADD Instruktion angewandt werden.

#### Beispiel

Gesetzt den Fall, es liegt ein Array  $T$  vor, das aus  $n$  Floats besteht, wobei  $n \in 4 \cdot \mathbb{N}$  ein Vielfaches von vier ist. Sei zudem vorausgesetzt, dass die CPU über 128-Bit-Register verfügt. Es können also theoretisch je vier Floats in jeweils ein Register geladen und diese mit jeweils vier anderen Floats eines anderen Registers parallel addiert werden.

In C++ werden dafür sogenannte „Intel Intrinsics“ (*Intel® Intrinsics Guide*, o. J.) zur Verfügung gestellt - ein Header, das neue Funktionsdeklarationen einführt und die üblichen Datentypen um zusätzliche Vektordatentypen wie `_m64`, oder `_m128` ergänzt.

Soll also die Additionslogik im Beispiel parallelisiert werden, können in jeder Iteration nun jeweils vier Elemente aus dem Array in jeweils ein Register geladen und die Daten beider Registers gleichzeitig addiert werden. Bei  $n$  Daten sind das also nur noch  $\frac{n}{4}$  Instruktionen.

## Von SIMD zu GPU

GPUs sind im Unterschied zu CPUs, die **latenzorientiert** sind, **durchsatzorientiert**. Konkret heißt das, dass CPUs **unterschiedliche** Operationen auf verschiedene Daten möglichst **schnell** und wenig zeitverzögert, während GPUs auf möglichst **viele** Daten **gleiche** Operationen ausführen möchten.

Für sich einzeln genommen, sind die Kerne einer GPU im Vergleich zu den Kernen einer CPU sehr langsam. Die hohe Geschwindigkeit für gleichartige Operationen auf viele Daten wird vielmehr dadurch erreicht, dass viele solcher relativ langsamem Kerne zusammengeschaltet werden, wobei jeder Kern mehrere Daten aufnehmen und auf diesen anschließend Operationen parallel ausführen kann. Hier wird demnach die hohe Latenz, respektive die relativ langsameren Verarbeitungsgeschwindigkeiten, durch einen noch höheren Durchsatz kompensiert.

Programmiert man Einheiten die auf der GPU ausgeführt werden sollen, ist es deshalb besonders wichtig sie so zu entwerfen, dass diese einen möglichst uniformen Ablauf haben, sogenannte **Branches** also vermieden werden. Im Code führen **Branches** (=konditionale Programmflüsse) zu sogenannten **Divergenzen**, die die Effizienz der Ausführung von Shader-Einheiten drosseln.

Um den negativen Einfluss von Divergenzen in der Ausführung besser erfassen zu können, muss die grundlegende Architektur einer GPU verstanden werden. Hierfür wird die sogenannte Fermi-Architektur von Nvidia (*NVIDIA's Next Generation CUDA(TM) Compute Architecture: Fermi*, o. J.) betrachtet:

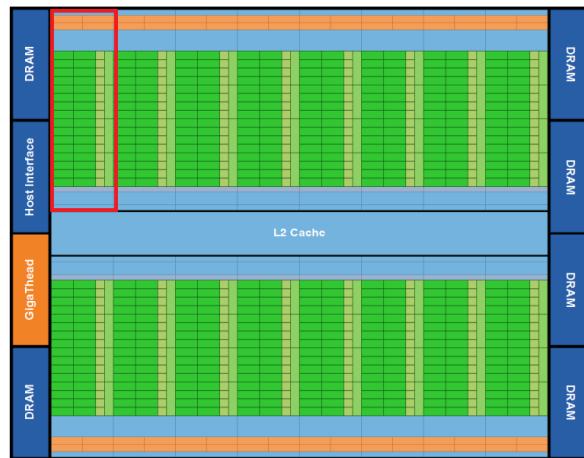


Abbildung 2.9: Die Fermi-Architektur. In rot gekennzeichnet: Ein einzelner Streaming-Multiprocessor (*NVIDIA's Next Generation CUDA(TM) Compute Architecture: Fermi*, o. J.)

Es existieren 16 SMs (Streaming Multiprocessors), wobei jeder Streaming-Multiprocessor aus jeweils 32 Kernen besteht. Die SM führen Threads in Gruppen von jeweils 32 Threads aus, die Warp genannt werden (*NVIDIA's Next Generation CUDA(TM) Compute Architecture: Fermi, o. J.*).

Im Additionsbeispiel vorhin, konnte die CPU mit 128-Bit Registern gerade einmal vier 32-Bit-Zahlen gleichzeitig verarbeiten. Die Fermi-Architektur erlaubt es, 16 unterschiedliche Operationen (=Codes) auf jeweils 32 Daten (die gar wesentlich größer als 32 Bit sind) parallel auszuführen. Innerhalb eines Warps (quasi die „Vektorbreite“, die hier 32 beträgt) werden also dieselben Operationen (=derselbe Code) auf unterschiedliche Daten angewandt. Divergiert der auszuführende Code eines Warps, divergiert der gesamte Warp. Konkret heißt das, dass die unterschiedlichen Code-Pfade eines bedingten Abschnitts bei unterschiedlichen Evaluationen nicht parallel, sondern sequentiell ausgeführt werden, sodass die Ausführungszeit des Codes innerhalb eines Warps sich mindestens verdoppeln kann. Im schlimmsten Fall also, liegen mehrere Kerne eines Warps brach (*Introduction to GPGPU and CUDA Programming: Thread Divergence, o. J.*), bis die Ausführung eines Zweiges abgeschlossen ist. Um die Ressourcen voll auszunutzen liegt es deshalb nahe, möglichst „branchless“ Code zu entwickeln - Code also, der wenige, bis idealerweise gar keine Verzweigungen hat.

Im Kapitel **Implementierung** wird ein solcher, überwiegend divergenz-freier Raycasting-Algorithmus vorgestellt.

### 2.4.2 DirectX und Monogame

DirectX ist eine Sammlung von Programmierschnittstellen (APIs), die es dem Entwickler ermöglichen, bestimmte Klassen von Aufgaben (insbesondere Zeichenaufgaben) auf die GPU auszulagern. Neben der sogenannten Direct2D, oder Direct3D-Grafik-API, stellt DirectX sogar APIs zur Verfügung, um Tastatur -und Mauseingaben zu erkennen oder Sound zu generieren. Die Arbeit wird besonders von der Direct3D-API Gebrauch machen, die es erlaubt, bestimmte Softwaremodule (=Shader) zu kompilieren, diese auf der Grafikkarte auszuführen und sogar Ressourcen wie zwei -oder dreidimensionale Texturen zu allozieren und zu verwenden.

Im Kapitel **Implementierung** wird für die praktische Umsetzung ein „Custom“-Monogame-Fork von Markus Hötzinger (cpt-max) verwendet, eine plattformunabhängige Framework, die auf SharpDX, ein Wrapper (=Adapter), das DirectX-API-Zugriffe für .NET ermöglicht, zurückgreift.

### 2.4.3 Shader

Grundsätzlich werden mit Shader insbesondere programmierbare Teile der Grafikpipeline bezeichnet. In diesem Zusammenhang spielen, wie im ersten Kapitel bereits vorweggenommen, der Vertex-Shader sowie der Pixel-Shader eine wichtige Rolle. Ersterer bereitet Ecken (=Vertizes) von Primitiven (etwa Dreiecken) auf, letzterer wird für jeden zu zeichnenden Pixel des Primitiven aufgerufen.

Diese Arbeit wird für das Rendering außerhalb der konventionellen Grafikpipeline operieren und auf den sogenannten **Compute-Shader** zurückgreifen.

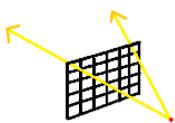
Im Unterunterabschnitt 2.4.1 wurde erklärt, dass ein einzelner Streaming-Multiprocessor Threads in Gruppen von jeweils 32 Threads ausführt. Im Compute-Shader muss über das sogenannte **numthread**-Attribut, die Anzahl der in einer einzelnen Thread-Gruppe auszuführenden Threads spezifiziert werden. Werden Compute-Shader-Instanzen ausgeführt (also ein sogenannter **DispatchCompute()** im Grafikkontext angestoßen), werden dann je Thread-Gruppe, die Anzahl der im Compute-Shader spezifizierten Threads ausgeführt.

Weil ein SM jeweils über 32 Kerne verfügt, liegt es nahe, die Gesamtzahl der auszuführenden Threads in einer Thread-Gruppe auf ein Vielfaches von 32 festzulegen. Auf diese Weise werden die Ressourcen voll ausgenutzt.

## 2.5 Raycasting-Grundlagen

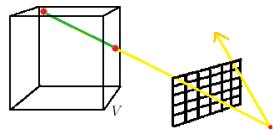
Mit Raycasting bezeichnet man im Allgemeinen ein Verfahren, um Volumendaten zu visualisieren. Das Verfahren lässt sich standardmäßig in vier Schritte unterteilen (vergleiche (Wikipedia contributors, 2022)), und zwar

1.



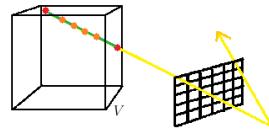
Strahl erzeugen: Vom Betrachter, zum momentan betrachteten Pixels des Frames wird ein Strahl gezogen, wobei der Richtungsvektor unter Umständen normiert wird.

2.



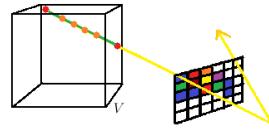
Schnittpunkt ermitteln: Der Gültigkeitsbereich des Volumens wird zumeist durch eine Bounding-Volume festgelegt. Die beiden Schnittpunkte des Bounding-Volume mit dem Strahl werden ermittelt, der Abstand der beiden Schnittpunkte zueinander ist dann die maximale Weglänge, die der Strahl durch das Volumen zurücklegen muss.

3.



Samplen: Auf der maximalen Weglänge entlang wird an äquidistanten Stellen das Volumen abgetastet. Hierbei ist zu beachten, dass die Abtastpunkte auch floatwertig sein können (und damit der Definition eines Voxels nicht genügen). Der Wert muss also unter Umständen interpoliert werden.

4.



Auf Grundlage der Samples wird die finale Farbe des Pixels festgelegt (diskrete Render-Gleichung).

Der dritte Punkt trifft auf Voxel-Rendering nicht zu, sodass das Voxel-Volume-Raycasting-Verfahren, vom Volume-Raycasting-Verfahren explizit abgegrenzt werden muss.

### 2.5.1 Abgrenzung

Die in Abschnitt 2.2 eingeführte Definition von Voxel erlaubt keine gleitkommazahlige, sondern ausschließlich ganzzahlige Werte.

Dabei ist ein individueller Voxel jeweils der Wert der Funktion  $f: \mathbb{N}^3 \rightarrow U$ , wobei  $U$  eine Menge beliebigen Typs bezeichnet - dies kann die Menge der reellen Zahlen  $\mathbb{R}$ , oder gar  $\mathbb{F}_2$  sein, also ein Restklassenkörper modulo 2, was als der Aktivierungsstatus' eines Voxels an der Stelle  $\vec{p} \in \mathbb{N}^3$  interpretiert werden kann.

Um der Definition von Voxels zu genügen und gleichzeitig den Raycasting-Ansatz auf das Rendering von Voxel zu übertragen, gilt es demnach, dass (gleitkommazahlige) Werte zwischen Integer-Koordinaten jeweils auf die komponentenweise (ganzzahlige) Koordinate, also  $\lfloor \vec{x} \rfloor$  mit  $\vec{x} \in V$ , abgerundet werden.

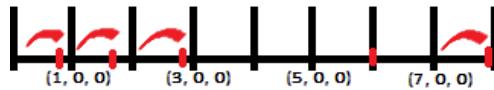


Abbildung 2.10: Gleitkommazahlige Koordinaten werden komponentenweise auf Integer-Koordinate abgerundet. Zu beachten ist, dass die Koordinate  $(6, 0, 0)$  unverändert bleibt.

Sollen alle diejenigen Voxel ermittelt werden, die von einem Strahl geschnitten werden, kann der in der Abbildung 2.10 dargestellte Sachverhalt algorithmisch ausgenutzt werden: Der Strahl wird in solche Segmente zerlegt, die immer jeweils so groß sind, wie die Voxel, die von diesem geschnitten werden (Amanatides et al., 1987).

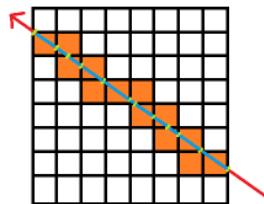


Abbildung 2.11: Der Strahl wird in Segmente zerlegt, die jeweils so lang, wie der Voxel die der Strahl schneidet, groß ist.

Dieser Ansatz wird im Kapitel **Implementierung** weiterverfolgt.

## 2.5.2 Kamera

In diesem Unterabschnitt sollen grundlegende Berechnungen gezeigt werden, um eine Kamera in die dreidimensionale Szene zu setzen. Eine grundlegende physikalische Vereinfachung die dabei gemacht wird, ist, dass (Sicht-)Strahlen vom Betrachter aus in die Szene geworfen werden und nicht etwa, physikalisch plausibler, beliebig viele reflektierte Strahlen den Betrachter treffen. Diese Vereinfachung ermöglicht es immer nur die Strahlen zu betrachten, die auch wirklich im Relevanzfeld des Betrachters stehen.

### Allgemeine Vorgehensweise

Die Kamera-Strahlen werden in der im Unterabschnitt 2.4.3 eingeführten Compute-Shader erzeugt.

Compute-Shader-Instanzen werden über eine sogenannte **.DispatchCompute()**-Methode ausgeführt, die über drei Argumente verfügt, wobei jedes Argument jeweils die Anzahl der Thread-Gruppen in je eine Koordinatenrichtung angibt.

Aufgrund der im Unterunterabschnitt 2.4.1 dargelegten Architektur von Grafikkarten, wird das Fenster in 8x8 große Quadrate gleichmäßig aufgeteilt. Die Anzahl der auszuführenden Threads (=**numthreads**) in einer einzelnen Thread-Gruppe beträgt damit insgesamt 64 (das offensichtlich ein Vielfaches von 32 ist). Die Anzahl der auszuführenden Thread-Gruppen in je X -und Y-Richtung ist dann die Anzahl der 8x8 großen Quadrate in je X -und Y-Richtung.

Konkret heißt das also, dass wenn die Fenstergröße (1088, 720) beträgt, die Argumente der **DispatchCompute()**-Methode ( $\frac{1088}{8}, \frac{720}{8}$ ) lauten müssen. Es würden in diesem Szenario also jeweils 136 Thread-Gruppen in X -und 90 in Y-Richtung im Idealfall mit je 64 Threads **parallel** ausgeführt werden.

### Erzeugung des Strahls

Im Compute-Shader lässt sich über die sogenannte **globalID** (der im **DispatchThreadID**-Register liegt) der Index des momentan ausgeführten Threads auslesen.

Weil für jeden Pixel effektiv ein Thread ausgeführt wird, stellt der Index des aktuellen Threads im Grunde auch gleichzeitig die Pixel-Koordinate dar.

Die Komponenten der Pixel-Koordinate (=**globalID**) liegen zunächst zwischen 0 und der Fensterbreite, respektive Fensterhöhe. Die Pixel-Koordinate wird komponentenweise deshalb zunächst durch die Fensterbreite sowie die Fensterhöhe dividiert und damit auf den Bereich zwischen 0 und 1 skaliert.

Anschließend wird die skalierte Pixel-Koordinate um den Nullpunkt zentriert, um nachfolgende Berechnungen zu erleichtern:  $\vec{uv} = \vec{v}_{scaledPixelCoord} \cdot 2 - \vec{1}$ .

Weil es sich beim Raycasting-Verfahren um eine **Projektion** dreidimensionaler Objekte im Raum auf eine Ebene handelt, ist in diesem Zusammenhang deshalb die Öffnungsweite (field-of-view (fov), z. Dt. Sichtfeld), die als Winkel angegeben wird, wichtig.

Konkret handelt es sich bei der Öffnungsweite um den Winkel  $\alpha$ , wie er in der Abbildung dargestellt wird:

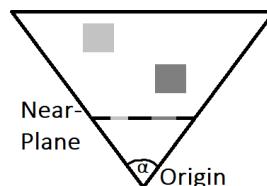


Abbildung 2.12: Charakteristisch für die perspektivische Projektion ist die von der Entfernung abhängige Größe eines projizierten Objekts auf der Projektionsfläche.

Die mit „Near-Plane“ bezeichnete Gerade ist eben auf die zu projizierende Fläche: Ihre Breite entspricht zwei Einheiten in jeweils X - sowie Y-Richtung, ist dabei um den Koordinatenursprung zentriert (deshalb wird auch die Pixel-Koordinate um den Koordinatenursprung zentriert) und liegt damit zwischen  $-1$  und  $1$ . Dem Umstand geschuldet, dass die Projektionsfläche prinzipiell konstanter Größe ist, übt die Öffnungsweite nur Einfluss auf den Abstand des Zentrums (in der Abbildung 2.12 als **Origin** gekennzeichnet) von der Projektionsfläche und entspricht damit der Brennweite.

Die Berechnung des Abstandes erfolgt, trigonometrisch leicht nachvollziehbar, folgendermaßen:

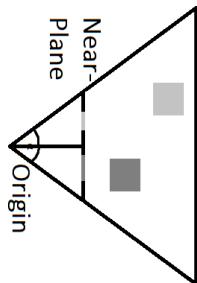


Abbildung 2.13: Die Öffnungsweite sowie die Breite der Projektionsfläche sind bekannt.

Zu ermitteln ist die Brennweite, die in der Abbildung als horizontale Linie vom Ursprung zur Projektionsfläche skizziert ist.

Die Gleichung

$$\tan \frac{\alpha}{2} = \frac{v_{\text{nearPlaneWidth}}/2}{f_{\text{focalLength}}}$$

wird nach  $f_{\text{focalLength}}$  umgestellt.

Dabei entspricht  $v_{\text{nearPlaneWidth}}/2$  der Konstruktion nach eins, woraus folgt

$$f_{\text{focalLength}} = \frac{1}{\tan \frac{\alpha}{2}}.$$

Der Ursprung des Strahls liegt somit in

$$\vec{o}_{\text{origin}} = \begin{pmatrix} 0 \\ 0 \\ -f_{\text{focalLength}} \end{pmatrix},$$

der Richtungsvektor des Strahls ist dann

$$\vec{p}_{direction} = \begin{pmatrix} \vec{uv}_x \\ \vec{uv}_y \\ 0 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ -f_{focalLength} \end{pmatrix}$$

und damit insgesamt:

$$g : \vec{x} = \vec{o}_{origin} + \lambda \cdot \vec{p}_{direction}.$$

### 2.5.3 Ray-AABB-Intersection

Eine wichtige Operation in dieser Arbeit stellt die Berechnung des Schnittpunktes zwischen Strahl und Voxel dar. Dabei können Voxel als sogenannte AABB, also **Axis Aligned Bounding Boxes**, interpretiert werden. Der Name röhrt daher, dass AABBs im Raum achsenorientiert, d.h. die Seiten des Quaders also jeweils parallel zur X-, Y-, und Z-Achse sind.

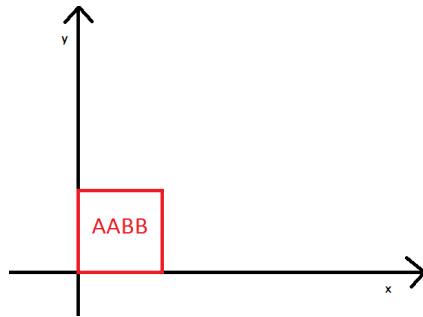


Abbildung 2.14: Die Seiten des AABB (im Bild AABR) verlaufen parallel zu den Achsen.

**Definition 9.** Bezeichne  $w_{left}, w_{right}, h_{bottom}, h_{top}$  die Seiten eines Rechtecks und seien ferner  $M_{left} = \{x \geq w_{left}\}$ ,  $M_{right} = \{x \leq w_{right}\}$ ,  $M_{bottom} = \{y \geq h_{bottom}\}$ ,  $M_{top} = \{y \leq h_{top}\}$  jeweils die Menge aller Werte die von den Seiten des Rechtecks begrenzt werden, dann wird eine AABR (Axis-Aligned-Bounding-Rectangle) als die Schnittmenge der Mengen  $V_{AABR} = M_{left} \cap M_{right} \cap M_{bottom} \cap M_{top}$  definiert. Die Definition gilt für AABBs analog (wobei diese um zwei zusätzliche Seiten,  $t_{back}$  und  $t_{front}$ , ergänzt werden muss).

Konkret heißt das:

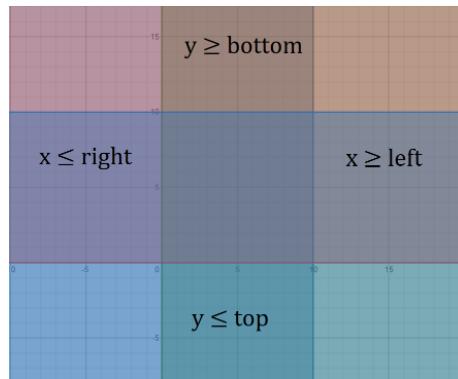


Abbildung 2.15: Die AABR (im Bild AABR) ist der Durchschnitt der in der Definition dargelegten, beschränkten Mengen (Plotter: Desmos).

Diese Definition erleichtert die Berechnung der beiden Schnittpunkte: Es muss jetzt lediglich ermittelt werden, zu welchen beiden Zeitpunkten  $\lambda$  sich der Strahl frühestens im und außerhalb des Durchschnitts aller Mengen befindet.

Die Berechnung erfolgt komponentenweise:

$$\begin{aligned} w_{left} &\leq \vec{o}_{origin_x} + \lambda \cdot \vec{p}_{direction_x} \leq w_{right} \\ h_{bottom} &\leq \vec{o}_{origin_y} + \lambda \cdot \vec{p}_{direction_y} \leq h_{top}. \end{aligned}$$

Es werden beide Gleichungen nach  $\lambda$  aufgelöst:

$$\frac{w_{left} - \vec{o}_{origin_x}}{\vec{p}_{direction_x}} \leq \lambda \leq \frac{w_{right} - \vec{o}_{origin_x}}{\vec{p}_{direction_x}} \quad (2.1)$$

$$\frac{h_{bottom} - \vec{o}_{origin_y}}{\vec{p}_{direction_y}} \leq \lambda \leq \frac{h_{top} - \vec{o}_{origin_y}}{\vec{p}_{direction_y}}. \quad (2.2)$$

Es existieren zwei Lösungen,  $\lambda_1$  und  $\lambda_2$ , wobei ersterer den frühesten Zeitpunkt darstellt, in welchem der Strahl in das ABBR eintritt, und letzterer den frühesten Zeitpunkt darstellt, in welchem der Strahl aus dem AABR austritt.

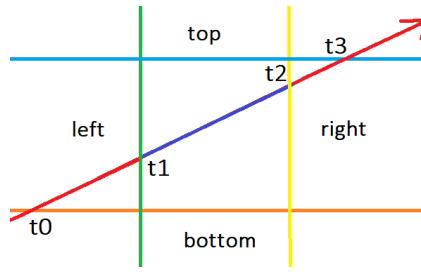


Abbildung 2.16: Das Ziel ist es, die beiden Zeitpunkte zu ermitteln, in welchem der Strahl innerhalb des AABB liegt.

Wenn  $\lambda_1$  den frühesten Eintrittszeitpunkt bezeichnet, dann ist  $t_0$  zu früh,  $t_1$  aber ideal: Der früheste Eintrittszeitpunkt ist demnach das Maximum der beiden Zeitpunkte  $t_0$  und  $t_1$ , also

$$\lambda_1 = \max \frac{w_{left} - \vec{o}_{origin_x}}{\vec{p}_{direction_x}}, \frac{h_{bottom} - \vec{o}_{origin_y}}{\vec{p}_{direction_y}}.$$

Analog ist der früheste Austrittszeitpunkt das Minimum der beiden Zeitpunkte  $t_2$  und  $t_3$ , also

$$\lambda_2 = \min \frac{w_{right} - \vec{o}_{origin_x}}{\vec{p}_{direction_x}}, \frac{h_{top} - \vec{o}_{origin_y}}{\vec{p}_{direction_y}}.$$

Die beiden Schnittpunkte sind dann

$$a_1 = \vec{o}_{origin} + \lambda_1 \cdot \vec{p}_{direction}$$

und

$$a_2 = \vec{o}_{origin} + \lambda_2 \cdot \vec{p}_{direction}$$

wobei  $a_1$  den nahen und  $a_2$  den fernen Schnittpunkt bezeichnet.

# 3 Raycasting: Theoretische Grundlagen

In diesem Kapitel werden theoretische Grundlagen des Raycasting-Ansatzes erarbeitet, vorherige Arbeiten gezeigt und Algorithmen skizziert, die im nachfolgenden Kapitel **Raycasting: Implementierung** realisiert werden.

## 3.1 Sparse-Voxel-Octrees

Wie im Kapitel **Grundlagen** im Unterabschnitt 2.3.2 bereits angeschnitten, ist ein Octree eine Datenstruktur (=Beschleunigungsstruktur), die komplexe Figuren durch einfache geometrische Objekte (in diesem Fall Quader) hierarchisch annähert. Dabei verfügt jeder Eltern-Knoten über acht Kind-Knoten: Soll ein Strahl gegen ein Octree auf einen Schnittpunkt geprüft werden, werden nur solche Knoten betrachtet, die hierarchisch aufeinanderfolgen. Auf diese Weise kann der hierarchisch letzten Knoten, der vom Strahl geschnitten wird, in  $O(\log s)$  ermittelt werden, wobei  $s$  die initiale Größe der Octree bezeichnet (siehe Satz 2 im Unterabschnitt 2.3.2).

Die Sparse-Voxel-Octree-Datenstruktur verfolgt diese Idee und macht sie für Raycasting-Ansätze zugänglich. Das Attribut **sparse** (z. Dt. spärlich) röhrt daher, dass Knoten, die keine relevanten Daten repräsentieren, nicht abgespeichert werden, die Datenstruktur also insgesamt spärlich bleibt.

### 3.1.1 Verwendung beim Raycasting

Vor dem Hintergrund, dass jeder Knoten eines Octree mit einem Quader korrespondiert, macht es Sinn, diese Quader (und damit die Knoten, (Laine & Karras, 2011)) im Grunde als Voxel zu betrachten (siehe Definition 6 im Abschnitt 2.1).

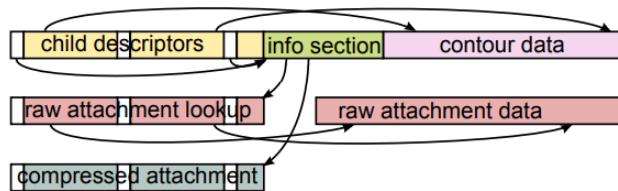
Ein grundlegender Vorteil dabei ist auch, dass der Detaillierungsgrad (engl. Level-of-Detail) des Renderings, aufgrund der hierarchischen Konstruktion dieser Datenstruktur direkt realisiert werden kann, indem die Tiefe der Traversierung etwa durch die Distanz des Betrachters zum Knoten vorgegeben wird.

### 3.1.2 Vorherige Arbeit: Efficient Sparse Voxel Octrees (NVIDIA Research)

Allgemein wird im Paper „Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation“ von (Laine & Karras, 2011) nicht nur eine Datenstruktur vorgestellt, die Voxel-Daten kompakt im Speicher vorhält, sondern auch Ansätze diskutiert, um Voxel-Präsentation von Modellen zu verfeinern sowie ein Algorithmus präsentiert, der die Datenstruktur für das anschließende Raycasting effizient nutzt.

Diese Arbeit wird sich vor allem auf eine Idee der im Paper vorgestellten Voxel-Daten konzentrieren.

Im Paper ist ein einzelner Octree-Datum folgendermaßen definiert:



**Figure 1:** Single block of octree data. The child descriptor and attachment arrays are addressed with the same index. The gaps in the arrays are due to page headers and far pointers. Page headers are placed at every 8 kilobyte boundary, and point to the info section.

Abbildung 3.1: (Laine & Karras, 2011)

Dabei sind besonders das **Child-Descriptor** sowie das **Contour-Data** Feld wichtig: Auf diese wird immer dann zugegriffen, wenn ein Strahl ein Knoten des Octree (also die Oberfläche eines Voxels) trifft. Auf der einen Seite legen die Kontur-Daten die geometrische Form der Voxel fest, auf der anderen Seite gibt der Child-Descriptor die Topologie des Octree vor.

child pointer	far	valid mask	leaf mask
15	1	8	8
contour pointer			contour mask
24			8

Figure 2: 64-bit child descriptor stored for each non-leaf voxel.

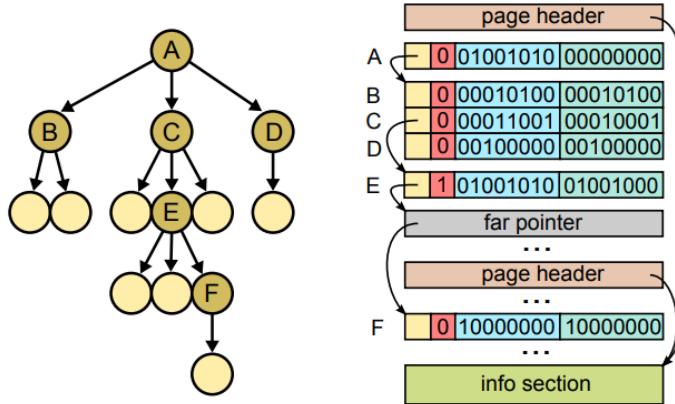


Abbildung 3.2: (Laine & Karras, 2011)

Kontur-Daten erlauben es, dass Problem der Block-Artefakte bei der Voxel-Darstellung von Modellen anzugehen, indem das zu repräsentierende Modell nicht durch ganze Voxel-Würfel, sondern durch Ebenen im Voxel approximiert wird, die sich an die Oberfläche des ursprünglichen Modells anschmiegen, wodurch auch bei genauer Betrachtung der Modell-Silhouetten, würfelartige Artefakte reduziert werden.

Child-Deskriptoren ermöglichen es, die Octree-Datenstruktur wie in einer verketteten Liste (engl. Linked-List) zu traversieren. Im eigenen Ansatz wird besonders diese Idee verfolgt werden.

### 3.1.3 Eigener Ansatz: Sparse-Voxel-Octree

Anders als im Paper, wird im eigenen Ansatz auf Kontur-Daten verzichtet, dafür aber die Idee der verketteten Liste verfolgt. Die Arbeit definiert die Octree-Datenstruktur folgendermaßen:

```
struct OctreeEntry
{
    int childrenstartIndex;
    int data;
};
```

Die Variable **childrenstartIndex** gibt den ersten Index der nächsten acht Kind-Knoten an. Die Kind-Knoten eines Eltern-Knoten liegen im Speicher also nicht verstreut, sondern sequentiell angeordnet vor. Die streng sequentielle Anordnung der Kind-Knoten ist algorithmisches insofern plausibel, dass einerseits kein zusätzlicher Aufwand entsteht, alle Kind-Knoten eines Eltern-Knoten zu ermitteln, andererseits jeder Kind-Knoten einem Oktanten des Eltern-Knotens direkt zugeordnet werden kann.

Es kann exemplarisch definiert werden, dass der relative Index des ersten Kind-Knotens (also 1) den oberen, vorderen und linken Oktanten, der darauffolgende relative Index (also 2) des nächsten Kind-Knotens den oberen, vorderen und rechten Oktanten bezeichnet, et cetera.

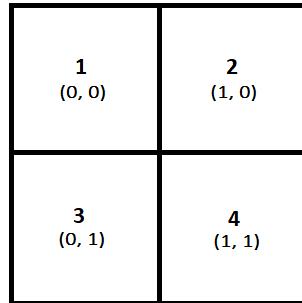


Abbildung 3.3: Jeder Index des Kind-Knotens einer Quadtree korrespondiert mit einem Quadranten. Analog korrespondiert jeder Index des Kind-Knotens einer Octree mit einem Oktanten. Im Klammern stehen die vektoriellen Darstellungen der Quadranten.

Die Zuordnung eines Knoten zu einem Oktanten ist insofern wichtig, weil die Struktur offensichtlich über keine Variable verfügt, die die absolute Position des Knotens explizit angibt. Dies ist auch nicht nötig, weil die absolute Position jedes betrachteten Knotens implizit über die Oktanten bestimmt werden kann. Die genaue Vorgehensweise wird nachfolgend behandelt.

Die Variable **data** ist ein sogenanntes Bitfeld. Der erste Bit des Integers gibt an, ob der Knoten ein Voxel-Datum hält, die übrigen 31 Bits können beliebig definiert werden.

Die Octree-Datenstruktur ist also ein Array von **OctreeEntry**, sodass eigentlich ein wahlfreier Zugriff existiert. Durch die Verwendung der Variablen **childrenstartIndex**

wird eine verkettete Liste lediglich **simuliert**. Insgesamt sieht die Octree-Datenstruktur im Speicher also folgendermaßen aus:

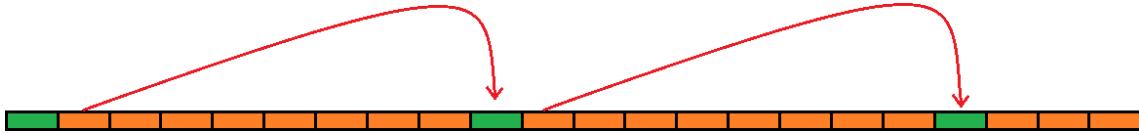


Abbildung 3.4: Die Kind-Knoten (orange) der Eltern-Knoten (grün) liegen im Speicher sequentiell angeordnet vor.

### 3.1.4 Überblick: Algorithmus und Pseudocode

Der nachfolgende Pseudo-Code zeigt einen rekursiven Algorithmus. Im Kapitel **Implementierung** wird ein rekursionsfreier Algorithmus gezeigt, weil Rekursionen in HLSL unzulässig sind. Der hier gezeigte Algorithmus setzt voraus, dass die initiale Größe  $s$  der Wurzel bekannt ist, die minimale Größe  $\varepsilon$  eines Voxels spezifiziert und ein Array  $P$  von Oktant-Vektoren existent ist. Die Variable  $P$  bezeichnet dabei ein solches Array, dessen Elemente die Oktanten vektoriell angibt, also exemplarisch dem Index 0 der Oktant  $(0, 0, 0)$ , dem Index 1 der Oktant  $(1, 0, 0)$ , dem Index 2 der Oktant  $(1, 0, 1)$ , et cetera, zuordnet.

Die Funktion verfügt über fünf Parameter: Den Strahl, den Octree, die aktuelle Tiefe, den Index des momentan zu betrachtenden Knotens und die aktuelle Position. Ein initialer Methodenaufruf kann  $(R_{Ray}, O_{Octree}, 0, 0, \vec{0})$  lauten. Am Anfang der Traversierung befindet sich der Algorithmus nämlich auf der nullten Ebene (=Tiefe) des Baumes, der Index des Wurzel-Knoten ist 0 und die momentane Position ist der Null-Vektor.

In jedem neuen Rekursionsaufruf werden die Parameter Tiefe, der Index des momentan zu betrachtenden Knotens sowie die momentane Position neu gesetzt. Ein neuer Rekursionsaufruf findet jeweils immer dann statt, wenn eines der Kind-Knoten des Eltern-Knotens vom Strahl geschnitten wird. Der Index des Oktanten korrespondiert dann mit dem relativen Index des Kind-Knotens. Der neue Wert des Parameters **Position** ist dann die momentane Position addiert mit dem Oktanten-Vektor (der mit dem relativen Index des Kind-Knoten korrespondiert) multipliziert mit der momentanen Größe des betrachteten Knotens (also  $\frac{s}{2^{\text{Tiefe}}}$ ).

---

**Require:**  $s$ : Denoting initial size of root  
**Require:**  $\epsilon$ : Denoting minimum size of voxel  
**Require:**  $P$ : Array of octant-vectors

```

function TRAVERSEOCTREE(Ray, Octree, Depth, CurrentIndex, CurrentPosition)
    CurrentNode ← Octree[CurrentIndex]
    ChildrenStartIndex ← currentNode.childrenStartIndex
    MinDist ←  $\infty$ 
    CurrentChildSize ←  $\frac{s}{2^{\text{Depth} + 1}}$ 
    if CurrentChildSize =  $\epsilon$  then
        return (CurrentPosition, CurrentChildSize, MinDist)
    for  $i \leftarrow 0$  to 7 do
        CurrentChild ← Octree[ChildrenStartIndex +  $i$ ]
        ChildPosition ← CurrentPosition +  $P[i] \cdot \text{CurrentChildSize}$ 
        if CurrentChild.HasData then
            HitAndDist ← CheckHit(Ray, CreateCube( $\vec{\text{ChildPosition}}$ , CurrentChildSize))
            if HitAndDist.Hit = true then
                RsltNode ← TraverseOctree(Ray, Octree, Depth + 1, ChildrenStartIndex +  $i$ , ChildPosition)
                if RsltNode.MinDist < MinDist then
                    CurrentPosition ← RsltNode.CurrentPosition
                    CurrentChildSize ← RsltNode.CurrentChildSize
                    MinDist ← RsltNode.MinDist
    return (CurrentPosition, CurrentChildSize, MinDist)
```

---

### 3.1.5 Bewertung: Sparse-Voxel-Octrees

So effizient die Traversierung einer Octree auch ist, so unflexibel verhält sie sich im Hinblick auf Veränderung der Voxel-Daten. Im schlimmsten Fall muss die gesamte Octree in jedem Frame neu erzeugt werden, was (zurzeit) für große Voxel-Daten in Echtzeit praktisch unmöglich ist. Im Verlauf der Arbeit wird eine simplere Form einer Octree dennoch als Beschleunigungsstruktur eingesetzt, um die Anzahl der sogenannten **false positives** beim Ansatz von (Amanatides et al., 1987) zu reduzieren.

## 3.2 Voxel-Volume-Raycasting

Das Voxel-Volume-Raycasting bedarf, im Unterschied zum Sparse-Voxel-Octree, keiner gesonderten Datenstruktur. Die Voxel sind in einem dreidimensionalen Array wohlge-

ordnet und können demnach beliebig modifiziert werden - Änderungen sind dann direkt sichtbar.

### 3.2.1 Vorherige Arbeit: A Fast Voxel Traversal Algorithm for Ray Tracing

Im Paper „A Fast Voxel Traversal Algorithm for Ray Tracing“ (Amanatides et al., 1987), beschreiben Amanatides und Woo einen Ansatz, um ein Array von Voxel mit möglichst wenig arithmetischen Operationen zu traversieren. Vorausgesetzt wird dabei insbesondere, dass die Voxel äquidistant angeordnet, der Raum also gleichmäßig partitioniert ist.

Eine ideale Datenstruktur, die dieser Forderung genügt, sind die gerade erwähnten Arrays.

Arrays verfügen zunächst weder über eine Größe, noch über eine Position im physikalischen Sinne. Allgemein bezeichnen Arrays eine Sammlung gleichartiger Objekte im Speicher und sind daher logische Konstrukte. Um Arrays im Hinblick auf Raycasting einsetzen zu können, wird eine Abbildungsfunktion  $f$  gesucht, die **Raumkoordinaten** auf **Array-Koordinaten** abbildet.

Mit Array-Koordinaten werden dabei ganzzahlige Werte bezeichnet, die im Gültigkeitsbereich des Arrays liegen und zusätzlich mit der Dimension des Arrays übereinstimmen. Die Array-Koordinaten eines eindimensionalen Arrays sind Indizes, die Array-Koordinaten eines dreidimensionalen Arrays sind ein 3-Tupel von Indizes.

Die Funktion  $f: \mathbb{R}^3 \rightarrow \mathbb{N}^3$  kann dann exemplarisch  $f(\vec{x}) = \left( \left\lfloor \frac{x_1}{s} \right\rfloor, \left\lfloor \frac{x_2}{s} \right\rfloor, \left\lfloor \frac{x_3}{s} \right\rfloor \right)$  lauten, wobei  $s$  die Größe eines Voxels bezeichnet. Dabei liegt es nahe  $s = 1$  zu setzen, um die Division zu eliminieren.

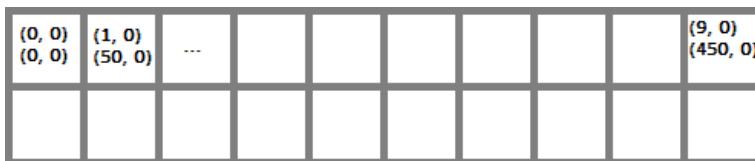
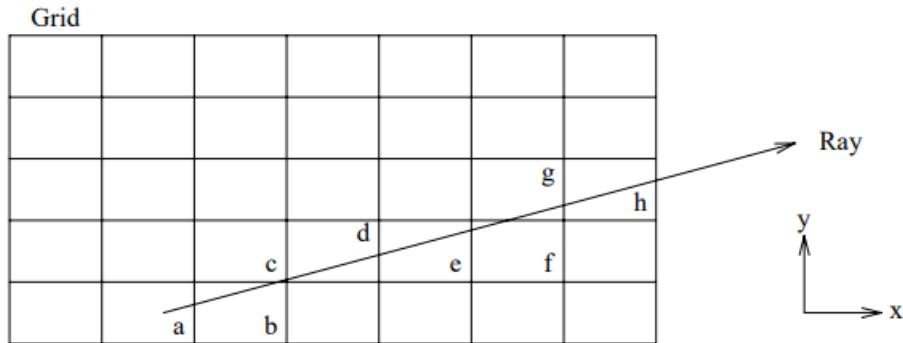


Abbildung 3.5: Der obere 2-Tupel gibt die Array-Koordinate, die untere hingegen die korrespondierende, absolute Koordinate an. In der Abbildung ist demnach  $s = 50$ .

Das Ziel ist jetzt, alle Voxel, die vom Strahl geschnitten werden, in richtiger Reihenfolge zu ermitteln:



**Figure 1**

Abbildung 3.6: Die geschnittenen Voxels sollen in richtiger Reihenfolge ermittelt werden  
(Amanatides et al., 1987)

Die grundlegende Idee des Algorithmus' ist es, die Komponenten des Strahls in jeweils voxel-große Intervalle zu zerlegen und in jeder Iteration immer in Richtung der Achse zu traversieren, die gewährleistet, dass der aktuell zu untersuchenden Voxel nicht verlassen wird (vergleiche (Chris Gyurgyik, 2020)).

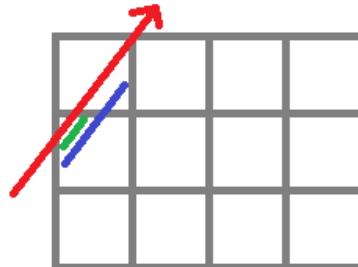


Abbildung 3.7: Es wird in jeder Iteration immer nur so traversiert, dass der momentan zu untersuchende Voxel nicht verlassen wird.

Der Abbildung ist zu entnehmen, dass beim Zeitpunkt  $t_x$  (blau) in welcher der Strahl die Vertikale (also  $x = i$ ) trifft, der momentan zu untersuchende Voxel verlassen wird, beim Zeitpunkt  $t_y$  in welcher der Strahl die Horizontale (also  $y = j$ ) trifft hingegen nicht. In jener Iteration würde also entlang der Y-Achse traversiert werden.

### 3.2.2 Eigener Ansatz: Voxel-Volume-Raycasting

Der eigene Ansatz verfolgt im Prinzip die Idee von (Amanatides et al., 1987), setzt diese aber praktisch anders um. Ähnlich wie bei (Amanatides et al., 1987), wird zunächst der Zeitpunkt  $t_0$  ermittelt, in welcher der Strahl das Gitterfeld frühestens betritt.

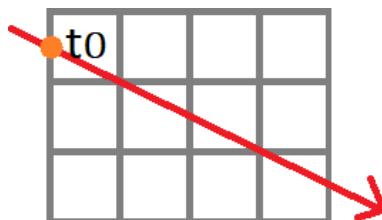


Abbildung 3.8: Der Zeitpunkt  $t_0$  wird ermittelt.

Der Punkt an dem der Strahl das Gitterfeld initial trifft, wird ermittelt als  $\vec{f}_0 = \vec{o} + t_0 \cdot \vec{r}$ . Eine wesentliche Operation bildet hierbei die in den Grundlagen komponentenweise definierte  $\lfloor \cdot \rfloor$  Funktion: Die Komponenten des Vektors  $\vec{f}_0$  werden auf einen Integer abgerundet und so der neue Vektor  $\vec{i}_0 = \lfloor \vec{f}_0 \rfloor$  berechnet. Die Initialen  $\vec{f}$  und  $\vec{i}$  bezeichnen dabei jeweils „float“ und „integer“.

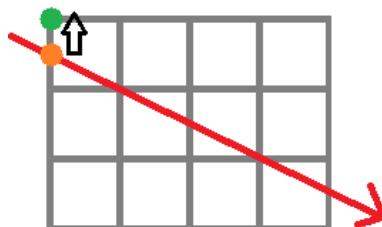


Abbildung 3.9: Um den resultierenden (eventuell nicht-ganzzahligen) Vektor  $\vec{f}_0$  komponentenweise auf eine Ganzzahl abzurunden, wird die  $\lfloor \cdot \rfloor$  Funktion eingesetzt. Der komponentenweise auf einen Integer abgerundete Vektor ist dann der Ursprung (=die Ecke) einer Kachel (=Voxels).

Der Vektor  $\vec{i}_0$  repräsentiert den Ursprung desjenigen Voxels, in der  $\vec{f}_0$  liegt. Ausgehend von diesem Punkt wird eine **AABB** (=Voxel) erzeugt, die nachfolgend als  $v_0$  bezeichnet wird.

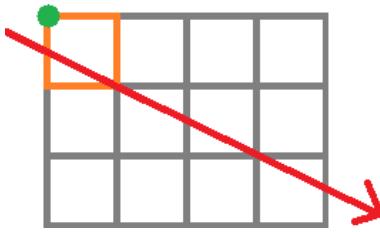


Abbildung 3.10: Ausgehend von  $\lfloor \vec{f}_0 \rfloor = \vec{i}_0$  wird eine AABR konstruiert.

In den Grundlagen im Unterabschnitt 2.5.3 wurde die Schnittpunktberechnung zwischen Strahl und AABB gezeigt. Auf diese Weise wird der Zeitpunkt  $t_{0_{max}} = t_1$  ermittelt, in welcher der Strahl den Voxel  $v_0$  frühestens verlässt.

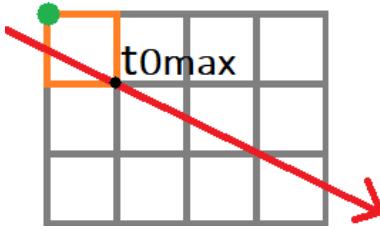


Abbildung 3.11: Der Zeitpunkt  $t_{0_{max}}$  (schwarz) wird ermittelt.

Anschließend wird  $t_1$  wieder in die Gleichung eingesetzt und  $\vec{f}_1 = \vec{o} + t_1 \cdot \vec{r}$  ermittelt. Wieder wird dieser Vektor mit  $\lfloor \cdot \rfloor$  komponentenweise auf einen Integer abgerundet, dadurch der Ursprung des nächsten Voxels  $v_1$  gefunden, wieder eine AABB erzeugt und abermals der Schnittpunkt zwischen Strahl und  $v_1$  berechnet. Dabei ist jeder Ursprung  $\vec{i}_k$  des Voxels  $v_k$  gleichzeitig ein 3-Tupel-Index, über den auf das dreidimensionale Array zugegriffen werden kann.

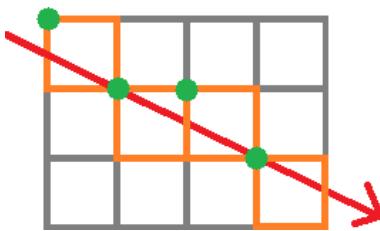


Abbildung 3.12: Das Gitterfeld wird iterativ traversiert, bis entweder ein **nicht-leeres** Voxel ermittelt, oder **das Ende** des Gitterfeldes erreicht wird. In jeder Iteration wird dabei ein neues AABR erzeugt. Die grünen Punkte repräsentieren dabei nicht nur die Ecke des Kachels, sondern auch den Index, über den auf das Array zugegriffen werden kann.

### 3.2.3 Bewertung: Voxel-Volume-Raycasting

Dieser Ansatz bedarf außer eines Arrays keiner weiteren Datenstruktur: Änderungen auf den Daten können dadurch direkt sichtbar gemacht werden, weil sowohl während des Renderings, als auch während der Modifikation, auf dieselben Daten zurückgegriffen wird.

Im Unterschied zu diesem Ansatz, sind Sparse-Voxel-Octrees nicht nur im Hinblick auf die Traversierung prinzipiell effizienter, in Sparse-Voxel-Octrees lassen sich auch Level-of-Detail-Ansätze wesentlich leichter realisieren. Im nachfolgendem Abschnitt wird dieser Ansatz deshalb, besonders wegen des ersten Punktes, um eine Octree-Beschleunigungsstruktur ergänzt. Ein grundlegendes Problem ist nämlich, dass beim Voxel-Volume-Raycasting stets auch solche Voxel untersucht werden, die gar keine Daten tragen, die Wahrscheinlichkeit also, **false positives** zu treffen, groß ist.

### 3.3 Beschleunigtes Voxel-Volume-Rendering

Beim beschleunigten Voxel-Volume-Raycasting, wird die im vorherigen Abschnitt gezeigte Idee um eine zusätzliche Octree-Beschleunigungsstruktur ergänzt. Das Voxel-Volumen wird wieder in gleichgroße Quader unterteilt, wobei die Größe eines Quaders hier zwischen 32x32x32 und 64x64x64 liegen kann.

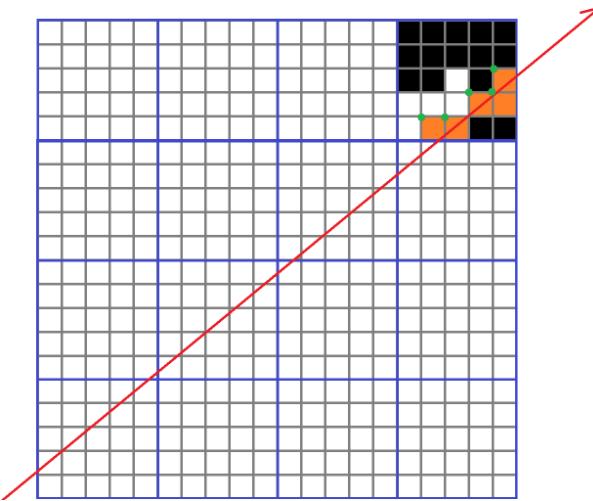


Abbildung 3.13: Die ersten Quadranten werden übersprungen, weil sie keine Daten halten. Erst im oberen, rechten Quadranten wird das Gitterfeld traversiert.

Das Ziel ist es, mögliche große Bereiche die keine Daten halten, gar nicht zu untersuchen. Der Strahl traversiert also den Octree zunächst regulär, bis ein nicht-leerer, 32x32x32 großer Blatt-Knoten getroffen, und erst dann der Volume-Raycast auf diesen Blatt-Knoten ausgeführt wird. Auf diese Weise können wichtige Rechenzyklen gespart werden.

# 4 Implementierung

In diesem Kapitel werden die im vorherigen Kapitel gezeigten Ansätze implementiert.

Wie in den Grundlagen bereits hingewiesen, wird dafür insbesondere HLSL zum Einsatz kommen. Aus Platzgründen wird dieses Kapitel nicht die komplette Implementierung zeigen, sondern lediglich die wesentlichen Teile, die für das Verständnis des Gesamtkonzepts besonders wichtig sind.

Zunächst wird die Schnittpunktberechnung zwischen Strahl und Voxel realisiert, anschließend der Voxel-Volume-Raycasting-Algorithmus implementiert, dann nachfolgend gezeigt, wie ein Octree in der Grafikkarte erzeugt werden kann, worauf schließlich der Voxel-Volume-Raycasting-Ansatz um eine Octree-Beschleunigungsstruktur ergänzt und dabei ein rekursionsfreier Algorithmus gezeigt wird, um diesen effizient zu traversieren.

## 4.1 Implementierung: Ray-AABB-Schnittpunktberechnung

Die Implementierung folgt den in den Grundlagen im Unterabschnitt 2.5.3 eingeführten Berechnungen insbesondere vor dem Hintergrund, dass der Code möglichst bedingungsfrei, also branchless, ist.

### 4.1.1 Vorbemerkung zur Syntax

Vergleichsoperatoren auf Vektoren erfolgen in HLSL immer komponentenweise:

```
float3 a = float3(1, -9, 1)
float3 b = a > 0 // b = (1, 0, 1)
```

Ausdrücke wie diese sind insofern unproblematisch, weil diese einerseits in bedingungsfreien Code kompiliert, andererseits stets dieselbe Operationen auf verschiedene Daten angewandt werden (Unterunterabschnitt 2.4.1).

### 4.1.2 Code

Folgender Code wird, in leicht abgewandelter Form, in der Implementierung verwendet. Die Variablen **leftSide** und **rightSide** müssen nicht ständig neu berechnet, sondern können vor dem Methodenaufruf zwischengespeichert werden, um Rechenzyklen zu sparen.

```
bool checkHit(in Ray ray, in AABB aabb, out float tMin, out float tMax, out
    float3 sideMin, out float3 sideMax)
{
    float3 isPositive = ray.odir > 0; // ray.odir = 1.0 / ray.dir
    float3 isNegative = 1.0f - isPositive;

    float3 leftSide = aabb.center + isPositive * aabb.minSize + isNegative *
        aabb.maxSize;
    float3 rightSide = aabb.center + isPositive * aabb.maxSize + isNegative *
        aabb.minSize;

    float3 leftSideTimesOneOverDir = (leftSide - ray.origin) * ray.odir;
    float3 rightSideTimesOneOverDir = (rightSide - ray.origin) * ray.odir;

    tMin = max(leftSideTimesOneOverDir.x, max(leftSideTimesOneOverDir.y,
        leftSideTimesOneOverDir.z));
    tMax = min(rightSideTimesOneOverDir.x, min(rightSideTimesOneOverDir.y,
        rightSideTimesOneOverDir.z));

    float3 directionSign = sign(ray.odir);

    sideMin = (leftSideTimesOneOverDir == tMin) * directionSign;
    sideMax = (rightSideTimesOneOverDir == tMax) * directionSign;

    return tMax > tMin;
}
```

### 4.1.3 Erläuterung

Der Rückgabewert der Funktion gibt an, ob die AABB vom Strahl geschnitten wird, wobei die frühesten Zeitpunkte für den Eintritt und Austritt sowie die geschnittenen Seiten über die Parameter gesetzt werden. Die ersten beiden Zeilen werden benötigt, weil im Unterabschnitt 2.5.3 die Gleichungen komponentenweise mit dem Kehrwert des Richtungsvektor multipliziert werden: Ist dieser negativ, kehren sich die Vergleichsoperatoren um.

Insgesamt entsprechen **leftSideTimesOneOverDir** und **rightSideTimesOneOverDir** also komponentenweise jeweils der Gleichung 2.1 und weiter. Die Parameter **tMin** und **tMax** bezeichnen dabei die in der Abbildung 2.16 gezeigten idealen Zeitpunkten.

Die geschnittenen Seiten lassen sich sehr leicht ermitteln: Die Vektoren **leftSideTime-sOneOverDir**, respektive **rightSideTime-sOneOverDir** werden jeweils mit dem idealen Zeitpunkt verglichen, die resultierenden Vektoren **sideMin** sowie **sideMax** geben dann komponentenweise an, welche Komponente der eben genannten Vektoren jeweils mit dem idealen Zeitpunkt korrespondiert.

Um die Seite genauer zu spezifizieren, muss der Vektor vorzeichenbehaftet sein, weshalb der Vektor komponentenweise mit dem Vorzeichen der Komponenten des Richtungsvektors des Strahls multipliziert wird. Schaut der Betrachter von links auf den Voxel (d.h. die X-Komponente des Richtungsvektors ist negativ), dann ist **sideMin** sowie **sideMax**  $(-1, 0, 0)^\top$ : Daraus lässt sich dann ableiten, dass **sideMin** von links aus betrachtet die rechte und **sideMax** von links aus betrachtet die linke Seite des Voxels angibt. Diese wird notwendig sein, um im Voxel-Volume-Raycasting unter anderem **floating-point-Präzisionsfehler** zu kompensieren.

## 4.2 Implementierung: Voxel-Volume-Raycasting

Zunächst wird der Code gezeigt, wichtige Codezeilen wurden beziffert und werden erst nachfolgend erklärt.

### 4.2.1 Code

Etliche Variablen Deklarationen (als [...] gekennzeichnet) sind nicht enthalten, weil diese sich prinzipiell aus dem Kontext ableiten lassen.

```
bool arrayRayHit(in Ray ray, in AABB volume, out float3 hitPoint, out float3
normal, out AABB voxel)
{
    [...]
    bool initialHitStatus = checkHit(ray, volume, tMin, tMax, sideMin,
        sideMax);
    // [1]           v          [2]
    if ((!initialHitStatus) | (tMin < 0) & (tMax < 0))
        return false;

    float maximumDepth = tMax;
```

```

bool isOriginInside = isInsideVolume(ray.origin, arrayCube);

// [3] v [4]
float3 floatPosition3 = ray.origin + tMin * ray.dir * (!isOriginInside);
float3 intPosition3 = floor(floatPosition3); // [5]

float3 offset = saturate(intPosition3 - floor(floatPosition3 + 0.01f));
intPosition3 -= offset; // [6]

float3 pstvDirComps = saturate(sign(ray.dir)); // [9][0]
float3 ngtvDirComps = 1.0 - pstvDirComps; // [9][1]

AABB nextVoxel = createAABB(intPosition3, float3(1, 1, 1));
while (checkHit(ray, nextAABB, tMin, tMax, sideMin, sideMax)
    & (currentVoxelData == 0)
    & (maximumDepth - tMax) >= 1)) // [7]
{
    floatPosition3 = ray.origin + tMax * ray.dir; // [8]
    intPosition3 = floor(floatPosition3); // [8]

    float3 rightRnndComps = (intPosition3 - floor(floatPosition3 +
        0.01f)) + 1; // [9][2]
    float3 wrongRnndComps = 1 - rightRnndComps; // [9][3]

    offset = sideMax * ((pstvDirComps * wrongRnndComps) + (ngtvDirComps *
        rightRnndComps)); // [9][4]
    intPosition3 += offset; // [9][5]

    normal = -sideMax; // [10]

    currentVoxelData = getData(intPosition3);
    nextVoxel.center = intPosition3; // [11]
}

hitPoint = floatPosition3;
voxel = nextVoxel;

return currentVoxelData > 0; // [12]
}

```

## 4.2.2 Erläuterung

[1]: Wie im Unterabschnitt 3.2.2 dargelegt, wird zunächst der früheste Zeitpunkt  $t_0$  ermittelt, in welcher der Strahl das Gitterraum (=Voxel-Volumen) betritt und damit gleichzeitig auch, ob der Strahl das Volumen insgesamt schneidet. Die Variablen `sideMin` und `sideMax` erhalten diejenige Seite, die jeweils im最早est Eintritts-, respektive Austrittszeitpunkt geschnitten wurde.

[2]: Diese Bedingung prüft, ob der Blick des Betrachters vom Volumen abgewandt ist. Auf diese Weise wird verhindert, dass das Volumen-Rendering gespiegelt wird.

[3]: Der früheste Eintrittspunkt in den Gitterraum wird ermittelt.

[4]: Liegt der Ursprung des Strahls im Gitterraum (=Volumen), dann ist der früheste Eintrittspunkt in das Gitterraum der Ursprung des Strahls selbst.

[5]: Dies entspricht  $\vec{i}_0 = \lfloor \vec{f}_0 \rfloor$  im Unterabschnitt 3.2.2 - auf diese Weise wird der Ursprung des ersten Voxels ermittelt. Gleichzeitig bezeichnet `intPosition3` auch den Array-Index.

[6]: Unter Umständen durchstößt der Strahl den Gitterraum auf eine Art und Weise, dass der ermittelte Ursprung des Voxels außerhalb des Gitterraums liegt:

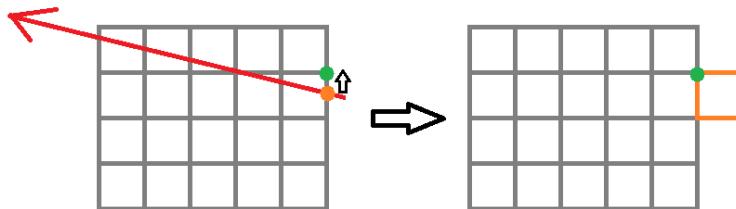


Abbildung 4.1: Der initiale Startpunkt  $\vec{f}_0$  und damit der komponentenweise auf einen Integer abgerundete Vektor  $\vec{i}_0$  liegt außerhalb des Gitterfeldes. Die mit  $\vec{i}_0$  korrespondierende AABR liegt damit ebenso außerhalb des Gitterfeldes.

Damit der Strahl **durch** den Gitterraum traversieren kann, wird der Ursprung des AABB (also  $\vec{i}_0$ ) in den Gitterraum hinein verschoben:

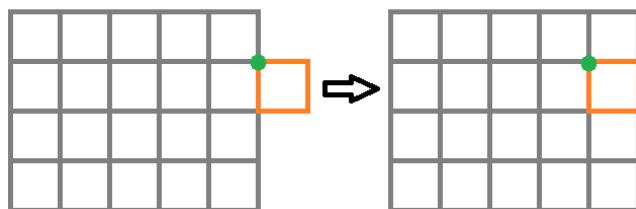


Abbildung 4.2: Der Ursprung der Kachel wird in das Gitterfeld hinein verschoben.

Durch Rundungsfehler ist es möglich, dass eine der Komponenten von **intPosition3** bereits verschoben vorliegt, sodass der **Offset**, um den **intPosition3** verschoben wird, die Differenz zwischen **intPosition3** sowie der komponentenweise aufgerundeten **floatPosition3** ist. Auf diese Weise wird eine Komponente nicht doppelt verschoben. Die Verschiebung des Vektors **floatPosition3** um eine Konstante (hier 0.01) wird nachfolgend erklärt.

[7]: Die Ausführung der Schleife wird solange fortgesetzt, bis a) entweder der nächste Voxel vom Strahl nicht geschnitten, b) ein nicht-leerer Voxel ermittelt wurde, oder c) der Strahl an einen der Volumengrenzen liegt.

[8]: Wie im Unterabschnitt 3.2.2 dargelegt, wird der Schnittpunkt mit dem momentan zu betrachtenden Voxel ermittelt und dieser anschließend wieder abgerundet, um den Ursprung des nächsten Voxel zu erhalten. Im Unterschied zu [3] wird hier **tMax** statt **tMin** eingesetzt. Dies liegt daran, dass der nächste Voxel an den momentan betrachteten Voxel angrenzt und damit mit dem frühesten Austrittszeitpunkt korrespondiert:

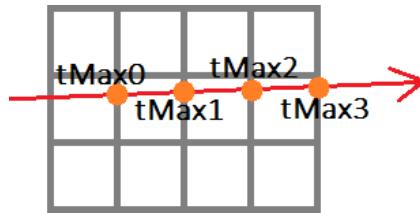


Abbildung 4.3: Der nächste Voxel grenzt an diejenige Seite des momentan betrachteten Voxels, die mit dem frühesten Austrittszeitpunkt des Strahls aus dem aktuellen Voxel korrespondiert.

[9]: Die mit [9] bezifferten Codeabschnitte haben im Grunde zwei Aufgaben: Einerseits um (a) die bei der Rundung aufgrund von Präzisionsfehlern entstehende Fehler zu kompensieren, andererseits um (b) eine ähnliche Problematik wie bei [6] zu behandeln:

(a) Aufgrund der einfachen Genauigkeit bei Float-Datentypen, kann unter Umständen der ermittelte Austrittspunkt exemplarisch  $\vec{p}_{hit} = (1.9999, 2, 2)^\top$  lauten, sodass  $\lfloor \vec{p}_{hit} \rfloor = (1, 2, 2)^\top$  ist. Der Ausdruck in [9][2] würde in diesem Beispiel also zu  $\vec{t}_{correctComponents} = (1, 2, 2)^\top - (\lfloor 1.999 + 0.01 \rfloor, \lfloor 2 + 0.01 \rfloor, 2)^\top = (-1, 0, 0)^\top$  und damit insgesamt zu  $(0, 1, 1)^\top$  ausgewertet werden.

(b) Zudem kann der Strahl (analog zu [6]) den Voxel auf eine solche Art und Weise schneiden, dass der auf einen Integer abgerundete Austrittspunkt nicht den Ursprung des nächsten zu betrachtenden, sondern des aktuellen Voxels darstellt. Dies geschieht immer dann, wenn der früheste Austrittszeitpunkt auf einer der minimalen Seiten des Voxels liegt:

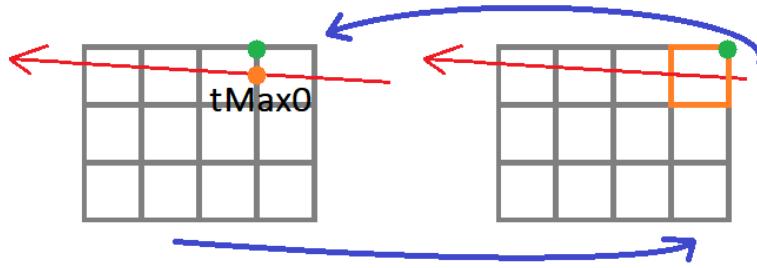


Abbildung 4.4: Der ermittelte Zeitpunkt  $t_{max_0}$  liefert nicht den Ursprung des nächsten, sondern des aktuellen Voxels.

In der Berechnung des **Offsets** in [9][4] muss schließlich verzweigungsfrei unterschieden werden, ob (a), oder (b) vorliegt:

Der Vektor **pstvDirComps** gibt an, welche Komponenten des Richtungsvektors positiv, die Umkehrung (**ngtvDirComps**) gibt dann analog an, welche Komponenten des Richtungsvektors negativ sind. Der Vektor **rightRnndComps** gibt an, welche Komponenten von **floatPosition3** richtig abgerundet, die Umkehrung (**wrongRnndComps**) gibt wieder analog an, welche Komponenten falsch abgerundet wurden.

Der Ausdruck

```
(pstvDirComps * wrongRnndComps)
```

entspricht (a): Die Multiplikation verhält sich hier ähnlich zum booleschen  $\wedge$ -Operator: Die Komponente des resultierenden Vektors ist dann eins, wenn in jener Komponente jeweils die Richtung positiv war und ein Rundungsfehler vorlag. Die korrespondierenden Komponenten des Vektors **intPosition3** müssten dann verschoben werden, um den Rundungsfehler auszugleichen.

Analog entspricht der Ausdruck

```
(ngtvDirComps * rightRnndComps)
```

hierbei (b): Die Komponenten des aus der Multiplikation resultierenden Vektors sind jeweils eins, wenn in jener Komponente die Richtung negativ war und **kein** Rundungsfehler vorlag, insgesamt also dieselbe Problematik wie in [6] vorliegt.

Die beiden Ausdrücke werden anschließend addiert (was dem booleschen  $\vee$ -Operator entspricht), um alle notwendig zu verschiebenden Komponenten zu ermitteln, wobei das Vorzeichen der Verschiebung durch den (mit **tMax** korrespondierenden) Vektor **sideMax** vorgegeben wird.

[10]: Die Variable **sideMax** gibt an, an welche Seite des aktuellen Voxels der nächste Voxel angrenzt. Die maximale Seite des aktuellen Voxel (also auf welcher Seite der Austrittspunkt liegt) wird durch das Vorzeichen spezifiziert: Ist der Wert von **sideMax** etwa  $(1, 0, 0)^\top$  - liegt also der Austrittspunkt auf der rechten Seite des Voxels -, dann liegt der Eintrittspunkt des Strahls in den nächsten Voxel konsequenterweise auf der linken Seite, womit die Normale also  $\vec{n} = (-1, 0, 0)^\top$  lautet.

[12]: Der Ursprung des nächsten Voxels ist eben der aktuell ermittelte 3-Tupel Index

[13]: Die Funktion gibt ein **true** zurück, wenn ein Voxel ermittelt wurde, der entsprechende Eintrag im Array also nicht-null war.

### 4.2.3 Komplexitätsanalyse

Floating-Point-Operationen (FLOP) stellen in aktuellen GPU-Modelle keinen wesentlichen Flaschenhals dar. Dass im Schleifenkörper mehr als sieben Multiplikationen ausgeführt werden, ist zunächst unproblematisch. Die Pascal-Architektur von Nvidia erlaubt exemplarisch über fünf Tera-FLOPS (Floating-Point-Operation per Second), also etwa  $10^{12}$  FLOP pro Sekunde.

Bei hohen Auflösungen aber beträgt die Anzahl der Strahlen um die  $1920 \cdot 1088 = 2088960$  pro Frame, weshalb es durchaus wichtig sein kann, die Anzahl der Schleifen-durchläufe (in einem „Worst-Case“-Szenario) zu analysieren:

Die maximale Weglänge die der Strahl durch das Volumen zu durchlaufen hat, ist die Diagonale des Quaders durch den Quader-Mittelpunkt. Bei einer Volumengröße von  $n^3$  beträgt diese also

$$\|\vec{v}_{diagonal}\| = \sqrt{3 \cdot n^2}$$

und liegt damit in

$$\sqrt{3 \cdot n^2} = O(\sqrt{3 \cdot n^2}) = O(\sqrt{3} \cdot \sqrt{n^2}) = O(\sqrt{n^2}) = O(n)$$

Mit steigendem  $n$  wächst die Anzahl der maximalen Iterationen also linear. Das Worst-Case-Szenario trifft aber immer nur dann ein, wenn der Strahl entlang der Diagonalen ausschließlich leere Voxel ermittelt: Der Algorithmus ist also extrem anfällig für **false positives**.

Um die Anzahl der **false positives** zu minimieren, wird das Volumen in gleichgroße Quader aufgeteilt (also im Grunde eine Octree erzeugt, siehe Abschnitt 3.3) und anschließend nur solche Quader untersucht, die auch nicht-leere Voxel umfassen.

#### 4.2.4 Divergenzanalyse

Der Code verfügt über zwei bedingte Abschnitte (ein **if**-Bedingung sowie eine **while**-Schleife). Der initiale bedingte Abschnitt ist überwiegend unproblematisch, weil es nahe liegt, dass sich der Betrachter häufiger im Volumen, als außerhalb des Volumens befindet und die Threads damit denselben Zweig ausführen. Befindet sich der Betrachter außerhalb des Volumens, ist eine Divergenz auch dann relativ unwahrscheinlich. Denn eine Thread-Gruppe führt Threads in einer Anordnung von 8x8-Quadranten aus (siehe Unterabschnitt 2.5.2), sodass innerhalb eines Warps ähnliche Strahlen in die Szene geworfen werden. Das Argument lässt sich auch auf die Schleife übertragen: Innerhalb einer Thread-Gruppe (8x8 Threads) unterscheiden sich die Distanzen der Strahlen zu einem Voxel nicht wesentlich, sodass im Idealfall alle Kerne des Warps ähnlich oft die Schleife durchlaufen und die Kerne damit ähnlich oft arbeiten.

### 4.3 Erzeugung einer Octree in der GPU

Im Unterschied zum Sparse-Voxel-Octree, wird das Volumen für die nachfolgende Datenstruktur in jeweils 32x32x32 gleichgroße Quader partitioniert. Teile des Volumen werden also auch dann weiter unterteilt, wenn diese keine Daten umfassen. Dies erleichtert nämlich die Erzeugung der Octree in einem **Compute-Shader**.

Es wird dieselbe Octree-Struktur wie in Unterabschnitt 3.1.3 benutzt, wieder wird ein Array eingesetzt und eine verkettete Liste simuliert. Die Anzahl der Array-Elemente unterscheidet sich aber, wie einleitend angedeutet, drastisch von der Sparse-Voxel-Octree-Variante: Weil das Volumen in gleichgroße Quader unterteilt wird, beträgt die Gesamtzahl der Elemente des Arrays

$$\sum_{i=0}^n 8^i = \frac{1 - 8^{n+1}}{1 - 8}$$

Beträgt die Größe des Volumens also  $512^3$  und ist die minimale Größe eines Knotens  $\varepsilon = 32$ , dann verfügt das Array über 4681 Elemente. Ein Element ist 64 Bit (nämlich zwei Integer), die Datenstruktur damit also insgesamt circa 37 Kilobyte groß.

#### 4.3.1 Code

Folgender Sourcecode wird in einem Compute-Shader für jeden Voxel des Volumens ausgeführt. Bei einer Volumengröße von  $512^3$ , entspräche dies also  $512^3$  auszuführende Threads.

```

uint calculateRelativeIndex(uint v, uint d)
{
    return (uint) (v / (size >> d));
}
uint calculateOctant(uint3 currentVoxel, uint d)
{
    return calculateRelativeIndex(pos.x, d) + calculateRelativeIndex(pos.z,
        d) * 2 + calculateRelativeIndex(pos.y, d) * 4;
}
uint calculateRelativeOctant(uint3 currentVoxel, uint d)
{
    return calculateOctant(pos % (uint) (size >> (d - 1)), d);
}
void updateOctree(uint3 currentVoxel) // [3]
{
    int relativeIndex = calculateRelativeOctant(currentVoxel, 1); // [3] [0]
    outputOctree[0].hasData |= 1;

    int nextIndex = relativeIndex + 1; // [3] [1]
    for (int depth = 1; depth <= maxDepth; depth++) // [3] [2]
    {
        OctreeEntry current = outputOctree[nextIndex];
        current.data |= 1; // [3] [3]
        outputOctree[nextIndex] = current;

        relativeIndex = calculateRelativeOctant(currentVoxel, depth + 1);
        // [3] [4]
        nextIndex = current.childrenstartIndex + relativeIndex; // [3] [5]
    }
}
[numthreads(4, 4, 4)] // [0]
void CS(uint3 currentVoxel : SV_DispatchThreadID)
{
    if (getData(currentVoxel) < 1) // [1]
        return;
    updateOctree(currentVoxel); // [2]
}

```

### 4.3.2 Erläuterung

[0]: Die Gesamtzahl der auszuführenden Threads beträgt  $4 \cdot 4 \cdot 4 = 64$ . Das Voxel-Volumen wird in jede Koordinatenrichtung also in 128 ( $= \frac{512}{4}$ ) 4x4x4 große Würfel (=Thread-Gruppe) zerlegt, wobei in jeder Thread-Gruppe jeweils 64 Threads enthalten sind. Das Argument **currentVoxel** enthält dabei die Koordinate des momentan zu betrachtenden Voxels.

[1]: Ist der Voxel an der durch **currentVoxel** angegebenen Position leer, dann wird die Ausführung des Shaders gestoppt, weil die Octree an dieser Stelle nicht weiter aktualisiert werden muss.

[2]: Liegt ein nicht-leerer Voxel vor, wird die Octree aktualisiert.

[3]: Der Octree wird hierarchisch von der Wurzel bis zum letzten Knoten, der den Voxel umfasst, aktualisiert. Wie in Unterabschnitt 3.1.3 erklärt, korrespondiert ein Oktanten-Index mit dem relativen Index des Kind-Knotens des momentan betrachteten Eltern-Knotens.

Das Ziel ist es jetzt, den ersten Bit der Variable **data** des aktuellen Eltern-Knotens zu setzen (also die Existenz von Voxel-Daten in diesem Eltern-Knoten zu markieren), dann den nächsten Kind-Knoten des aktuellen Eltern-Knoten zu ermitteln, der den momentan zu betrachtenden Voxel umfasst (und damit eine Ebene tiefer im Octree zu traversieren), diesen als nächsten Eltern-Knoten zu betrachten, wieder den ersten Bit der Variable **data** zu setzen, abermals den nächsten Kind-Knoten zu ermitteln, bis schließlich ein Blatt des Baumes (der 32x32x32 groß ist) erreicht wurde.

Der Index des Oktanten ist im Prinzip die eindimensionale Repräsentation des 3-Tupel-Index. Um in diesem Zusammenhang die Aufgabe der Funktionen **calculateRelativeOctant**, **calculateOctant** und **calculateRelativeIndex** besser einordnen zu können, wird zunächst die zweidimensionale Variante bemüht.

Gesetzt den Fall, man möchte ein zweidimensionales Array auf einen eindimensionalen Array abbilden. Dann lässt sich die Abbildungsvorschrift leicht herleiten, indem die Elemente des zweidimensionalen Arrays sequentiell anordnet werden:

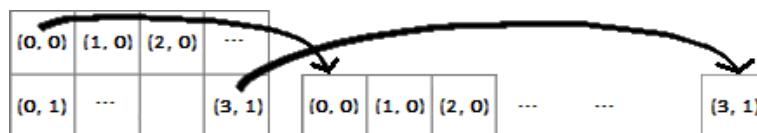


Abbildung 4.5: Die drei oberen Elemente des zweidimensionalen Arrays belegen die ersten drei Plätze, die drei unteren hingegen jeweils den sechsten, siebten und den achten Platz des eindimensionalen Arrays.

Bezeichnet  $w$  die Breite,  $i$  die  $i$ -te Spalte und  $j$  die  $j$ -te Zeile, dann beginnt die  $j$ -te Zeile also ab dem Index  $j \cdot w$ , sodass die Abbildungsvorschrift insgesamt

$$f: \mathbb{N}^2 \rightarrow \mathbb{N}$$

$$f_w(\vec{\mathbf{p}}) = p_i + p_j \cdot w$$

lautet.

Dasselbe Prinzip verfolgt die Funktion **calculateOctant**: Die Kind-Knoten eines Eltern-Knotens sind (logisch) in einem  $2 \times 2 \times 2$  dreidimensionalen Array angeordnet. Ziel ist es nun, 3-Tupel-Indizes auf eindimensionale Indizes abzubilden. Analog lautet die Abbildungsvorschrift hier

$$f: \mathbb{N}^3 \rightarrow \mathbb{N}$$

$$f_{2,2}(\vec{\mathbf{p}}) = p_i + p_j \cdot 2 + p_k \cdot 2 \cdot 2$$

Die  $k$ -te Ebene des dreidimensionalen Arrays würde also ab dem Index  $k \cdot 2 \cdot 2$  aufwärts beginnen, weil eine Ebene des  $2 \times 2 \times 2$  Arrays vier ( $2 \times 2$ ) Elemente umfasst.

In der Funktion **calculateOctant** werden die Komponenten des 3-Tupel-Indexes jeweils der Funktion **calculateRelativeIndex** übergeben, wo sie durch die Größe des aktuellen Knotens dividiert werden. Das Vorgehen wurde bereits im Abschnitt 3.2 vorweggenommen: Dort wurde eine Funktion gesucht, die Raumkoordinaten auf sogenannte Array-Koordinaten abbildet.

Bei einer Volumengröße von  $512^3$  liegen die Raumkoordinaten komponentenweise also zwischen 0 und 512.

Jetzt gilt es, diese Raumkoordinaten auf Vektoren der Oktanten (also Vektoren für die gilt, dass sie  $\vec{\mathbf{v}}_{octant} \in \{0, 1\}^3$  sind) und dann anschließend ein  $2 \times 2 \times 2$  großes, dreidimensionales Array auf ein 8-Element großen, eindimensionalen Array abzubilden.

Ist die aktuelle Tiefe exemplarisch  $n = 2$ , dann ist für

$$g: \mathbb{R}^3 \rightarrow \mathbb{N}^3$$

$$g(\vec{\mathbf{x}}) = \left( \left\lfloor \frac{x_1}{s} \right\rfloor, \left\lfloor \frac{x_2}{s} \right\rfloor, \left\lfloor \frac{x_3}{s} \right\rfloor \right)$$

der Wert  $s = \frac{512}{2^2} = 128$ .

Sei nun  $\vec{\mathbf{p}}_{currentVoxel} = (390, 0, 370)^\top$  die Position des aktuell zu betrachtenden Voxels (was der gerade genannten Raumkoordinate entspräche), dann ist  $g(\vec{\mathbf{p}}_{currentVoxel}) = (3, 0, 2)^\top$ .

Offensichtlich ist  $\vec{\mathbf{p}}_{currentVoxel} \notin \{0, 1\}^3$ . Um die Ursache dieses falschen Ergebnisses besser nachzuvollziehen, betrachte man folgende Skizze:

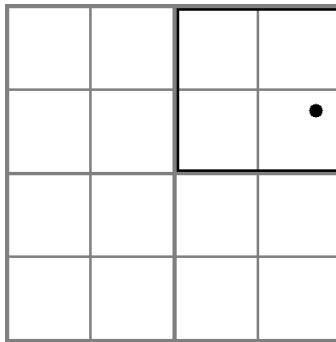


Abbildung 4.6: Die Skizze stellt einen Ausschnitt des 2x2x2 großen Würfels dar. In der Abbildung wird die 0-te Ebene dargestellt.

Der Abbildung 4.7 ist zu entnehmen, dass der Voxel sich einerseits (a) im hinteren, oberen, rechten Oktanten des Eltern-Knotens und andererseits (b) im vorderen, unteren und rechten Oktanten des Kind-Knotens befindet.

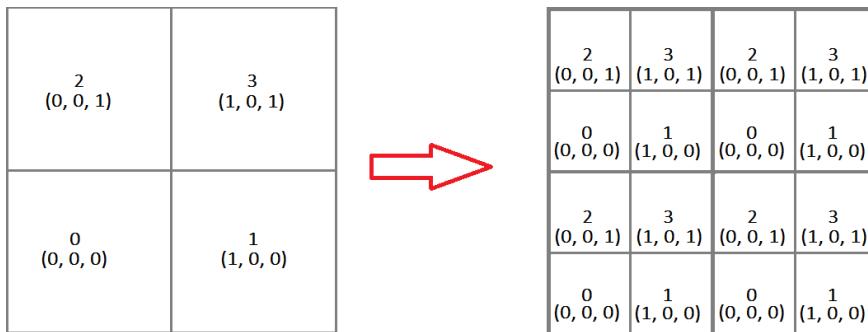


Abbildung 4.7: Die Index-Verteilung auf der 0-ten Ebene kann exemplarisch so aussehen.

Der Vektor  $\vec{p}_{currentVoxel} = (390, 0, 370)^\top$  liegt also im **dritten** Oktanten des Eltern-Knotens und im **ersten** Oktanten des Kind-Knotens.

Die Funktion **calculateRelativeOctant** wendet komponentenweise deshalb den Modulo-Operator, mit der Größe des Eltern-Knotens (also hier 256) als Modul, an. Auf diese Weise liegt der Vektor  $\vec{p}_{currentVoxel}$  komponentenweise dann zwischen [0, 255], sodass die Abbildungsfunktion  $g(\vec{x})$  nun die zum Eltern-Knoten relative, statt absolute Koordinate des Oktanten berechnet:

$$g(\vec{x}) = \left( \left\lfloor \frac{\lfloor x_1 \rfloor \bmod (2 \cdot s)}{s} \right\rfloor, \left\lfloor \frac{\lfloor x_2 \rfloor \bmod (2 \cdot s)}{s} \right\rfloor, \left\lfloor \frac{\lfloor x_3 \rfloor \bmod (2 \cdot s)}{s} \right\rfloor \right)$$

Das Modul wird mit zwei multipliziert, weil die Größe des Eltern-Knotens ermittelt werden muss, um die zum Eltern-Knoten relative Oktant-Koordinate zu berechnen. Die resultierende Array-Koordinate ist jetzt  $g(\vec{\mathbf{p}}_{currentVoxel}) = (1, 0, 0)^\top$ .

Insgesamt lautet also der Index desjenigen Oktanten des Eltern-Knotens (und damit der relative Index des Kind-Knotens), der diesen Voxel umfasst also

$$f_{2,2}((1, 0, 0)^\top) = 1 + 0 \cdot 2 + 0 \cdot 2 \cdot 2 = 1$$

Gelte nach wie vor  $\vec{\mathbf{p}}_{currentVoxel} = (390, 0, 370)^\top$ , dann beträgt der in der Zeile [3][0] ermittelte Index des Oktanten 3 (siehe Abbildung 4.7).

Der nächste zu betrachtende Index wird der in Zeile [3][1] initial inkrementiert, weil im nullten Index der Wurzel-Knoten liegt.

Die Schleife in [3][2] iteriert von der ersten bis zur letzten Tiefe des Baumes: Bei einer minimalen Blatt-Größe von 32x32x32 wäre die maximale Tiefe also 4.

In jeder Iteration wird, wie oben angedeutet, der erste Bit der **data**-Variable des aktuellen zu betrachtenden Knotens gesetzt ([3][3]). Anschließend wird in [3][4] der nächste Kind-Knoten ermittelt (und damit gleichzeitig der Baum eine Ebene tiefer traversiert). Der errechnete Index des Oktanten korrespondiert mit dem Kind-Knoten des aktuellen betrachteten Eltern-Knotens (siehe oben sowie Unterabschnitt 3.1.3).

Der nächste zu betrachtende Index ([3][5]) ergibt sich schließlich also aus der Summe des ersten Indexes des Kind-Knotens (also **childrenStartIndex**) mit dem Index des errechneten Oktanten ([3][4]).

### 4.3.3 Komplexitätsanalyse

Die Komplexität des Algorithmus' liegt offensichtlich in  $O(n^3)$  und ist damit ineffizient. Auch wenn fast alle Divisionen durch Zweierpotenzen, durch eine Bitverschiebung nach rechts substituiert wurden, werden nach wie vor Divisionen mit Rest ausgeführt.

Es liegt also nahe, diesen Algorithmus immer nur initial auszuführen, den Octree also immer nur zu Beginn vollständig zu konstruieren, während der Laufzeit dann nur Ausschnitte davon zu bearbeiten. Werden nämlich lediglich Teile des Voxel-Volumen modifiziert, muss konsequenterweise auch nur ein Teil der Octree-Datenstruktur aktualisiert werden.

Im Kapitel **Ergebnisse und Diskussion** wird ein solcher Ansatz angeschnitten werden, wobei die Octree-Struktur insofern angepasst wird, dass jeder Knoten der Octree die Anzahl der Voxels angibt, statt nur einen Bit, der die Existenz von Voxels in diesem Knoten andeutet, abzuspeichern.

## 4.4 Rekursionsfreie Traversierung eines Octree

Im Allgemeinen sind Rekursionen in HLSL nicht möglich, sodass der im Unterabschnitt 3.1.4 gezeigte Algorithmus in dieser Form nicht funktionieren würde.

Nachfolgend soll also ein iterativer, rekursionsfreier Ansatz erarbeitet werden. Dafür ist es wichtig, das grundlegende Prinzip rekursiver Aufrufe nachzuvollziehen. Man skizziere einen exemplarischen Ausschnitt des Aufruf-Baumes, der innerhalb einer Ausführungsinstanz des Algorithmus' etwa so aussehen könnte:

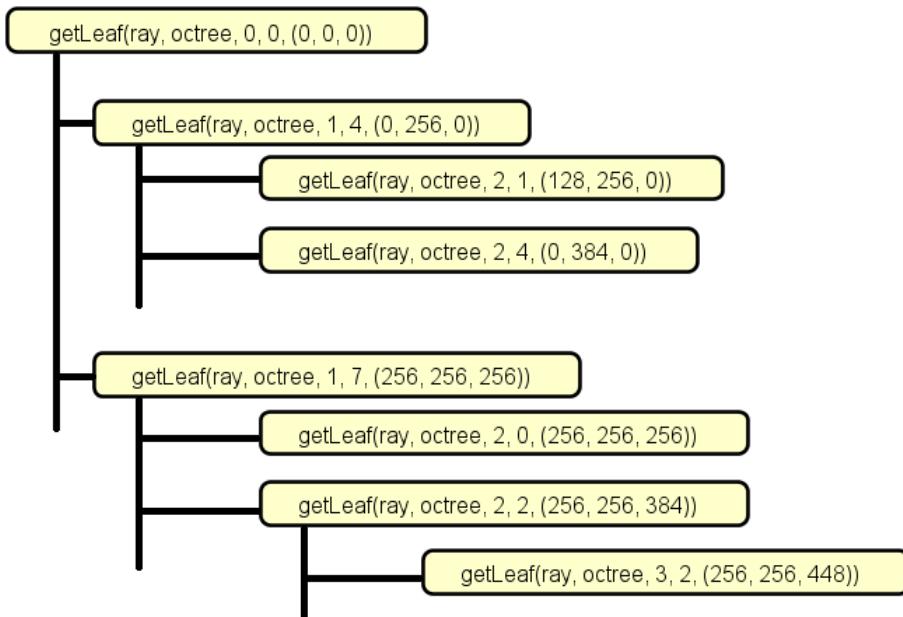


Abbildung 4.8: Die Abbildung zeigt einen möglichen Ausschnitt der rekursiven Aufrufe (Grafik: Astah UML).

Wird im Allgemeinen eine Methode aufgerufen, so werden regulär zunächst die Werte der Aufrufparameter der Methode in einen sogenannten Aufrufstapel geladen, die aktuelle Speicheradresse, auf das der Befehlszähler momentan verweist, gespeichert, der Befehlszähler auf die Speicheradresse der Methode gesetzt, die Methode ausgeführt, Ergebnisse kopiert und abschließend der letzte Stand des Befehlszählers wiederhergestellt. Im Falle einer Rekursion werden die Schritte wiederholt, sodass der Aufrufstapel der Methode wächst.

Diese Konzepte sind nicht ohne Weiteres auf HLSL zu übertragen, schon allein deshalb nicht, weil HLSL über keine Methoden/Funktionen im allgemeinen Sinne verfügt.

Alle deklarierten Methoden in HLSL sind nämlich standardmäßig „inline“ - der Methodenrumpf der verwendeten Methoden werden demnach also nicht aufgerufen, sondern lediglich in den „aufrufenden“ Code kopiert (Microsoft Docs HLSL (Stevewhims), o. J.).

Der nachfolgende Algorithmus verwendet einen Greedy-Ansatz (ähnlich wie in Unterabschnitt 3.1.4), ersetzt Rekursionen durch eine umfassende Iteration und nutzt Arrays fester Größe als Aufrufstapel.

#### 4.4.1 Code

Im nachfolgenden Quellcode werden wieder nur die relevanten Codeabschnitte gezeigt, Deklarationen werden nicht explizit angegeben (befinden sich aber innerhalb [...]).

```
AcceleratorResult acceleratedVolumeRayTest(Ray ray, int maxIterations) // [0]
{
    int skipListPerDepth[4]; int parentListPerDepth[4]; // [1]
    float3 positionListPerDepth[4]; // [1]
    [...]
    if (checkHit(ray, rootVolume, tMin, [...])) // [2]
    {
        [...]
        for (int iterations = 0; iterations <= maxIterations; iterations++)
            // [3]
        {
            int childStartIndex = parent.entry.childStartIndex; // [4]
            [...]
            for (int i = 0; i < 8; i++) // [4]
            {
                if (((skipListPerDepth[currentDepth] >> i) & 1)) // [5]
                    continue;
                Container child = getContainer(childStartIndex + i);
                if (!child.hasData)
                    continue;
                child.currentSize = parent.currentSize * .5f; // [6]
                child.currentPosition = parent.currentPosition +
                    octantVectors[i] * child.currentSize; // [6]
                [...]
                tMin = 0;
                bool result = checkHit(ray, childVoxel, tMin, tMax); // [7]
                if ((result) & (tMin < minDistToNode)) // [8]
                {
                    minDistToNode = tMin; // [8]
                    if (child.currentSize <= 32) // [9]
```

```

{
    [...]
    bool result = arrayRayHit(ray, childVoxel, tMinVoxel,
        [...]); // [10]
    if (result)
    {
        voxelFound = true; // [11]
        if (tMinVoxel < minDistToVoxel) // [12]
        {
            minDistToVoxel = tMinVoxel; // [12]
            [...]
        }
    }
    else
    {
        nextParent = child;
        nextParent.isNull = false;
        usedIndex = i; // [14]
        nextIndex = childStartIndex + i; // [15]
    }
}
if (voxelFound)
    break;
if (nextParent.isNull) // [16]
{
    skipListPerDepth[currentDepth] = 0; // [16] [0]
    currentDepth--; // [16] [1]
    if (currentDepth < 0) // [16] [2]
        break;
    int parentIndex = parentListPerDepth[currentDepth]; // [16] [3]
    nextIndex = parentIndex;
    nextParent = getContainer(parentIndex);
    nextParent.currentSize = initialSize >> currentDepth;
    nextParent.currentPosition =
        positionListPerDepth[currentDepth]; // [16] [4]
}
else // [17]
{
    skipListPerDepth[currentDepth] |= (1 << usedIndex); // [17] [0]
    positionListPerDepth[currentDepth] = parent.currentPosition;
}

```

```
// [17] [1]
parentListPerDepth[currentDepth] = currentIndex; // [17] [2]
currentDepth++; // [17] [3]
}
parent = nextParent;
currentIndex = nextIndex;
}
}
[...]
```

#### 4.4.2 Erläuterung

[0]: In der Methodensignatur ist besonders der Parameter **maxIterations** wichtig: Dieser gibt an, wie oft die Suche iteriert werden soll, um den idealen, vom Strahl geschnittenen, Blatt-Knoten zu ermitteln. Die maximale Anzahl der Versuche (=Iterationen) variable zu halten erlaubt es, unterschiedlichen Operationen, unterschiedliche Rechenzeit zuzuweisen. Auf diese Weise kann exemplarisch die Qualität der Schattenberechnung angepasst werden.

[1]: Die Arrays fester Länge simulieren zwar einerseits den oben genannten Aufrufstapel, andererseits funktioniert die Implementierung grundsätzlich anders als der im Unterabschnitt 3.1.4 gezeigte Algorithmus: Wie angedeutet, basiert die Implementierung auf einem reinen Greedy-Ansatz: Zu jedem Zeitpunkt wird zunächst immer nur der Kind-Knoten betrachtet, der die kürzeste Distanz zum Betrachter hat. Unter Umständen kann es aber passieren, dass von allen Eltern-Knoten ein Eltern-Knoten zwar die kürzeste Distanz zum Betrachter hat, die Kind-Knoten dieses Eltern-Knotens aber den Strahl nicht schneiden, oder ein im Kind-Knoten anderer Eltern-Knoten befindlicher Voxel, eine noch kürzere Distanz zum Betrachter hat (**false-positive**).

Liegt ein **false-positive** vor, traversiert der Algorithmus rückwärts (also eine Tiefe höher) zum vorherigen Knoten (=Eltern-Knoten des aktuell betrachteten Kind-Knoten) und untersucht dann den nächstbesten Kind-Knoten des Eltern-Knotens:

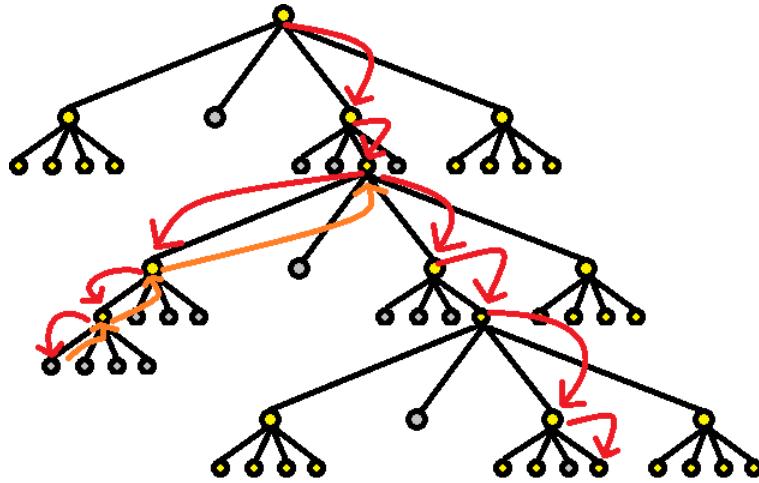


Abbildung 4.9: Der Octree wird nicht nur vorwärts traversiert (rot), sondern auch rückwärts (orange).

Um nicht denselben Kind-Knoten zu untersuchen, wird deshalb das Array **skipListPerDepth** verwendet: Dieser kodiert binär, welche Kind-Knoten in welcher Ebene bereits untersucht wurden. Der Ansatz wird unter [5] weiter präzisiert werden.

Wie der Abbildung 4.9 zu entnehmen, springt der Algorithmus unter Umständen wieder zum Eltern-Knoten. Die Octree-Datenstruktur ist aber nur einfach verkettet (siehe Strukturdefinition in Unterabschnitt 3.1.3). Um dennoch zwischen Kind -und Eltern-Knoten hin und her zu traversieren, wird deshalb das Array **parentListPerDepth** verwendet: Diese speichert den Index des zuletzt betrachteten Eltern-Knotens ab, sodass der Algorithmus jederzeit vom Kind-Knoten zum Eltern-Knoten gelangen kann.

Das Array **positionListPerDepth** speichert die zuletzt berechnete Position: Muss der Algorithmus den Baum rückwärts traversieren, muss die Position des Eltern-Knotens wiederhergestellt werden.

Insgesamt verfügen alle Arrays über vier Elemente, weil  $\frac{512}{2^4} = 32$ : Ein Blatt (und damit alle Knoten der letzten Tiefe der Octree) ist, wie im Abschnitt 4.3 beschrieben, 32x32x32 groß.

[2]: Schneidet der Strahl das Volumen (also die Wurzel des Baumes, der  $512^3$  groß ist) nicht, wird der Algorithmus an dieser Stelle nicht fortgesetzt.

[4]: Der erste Index des Kind-Knotens des aktuell betrachteten Eltern-Knotens wird ausgelesen. Ein Knoten verfügt über acht Kind-Knoten - die Anzahl der Schleifendurchläufe beträgt dementsprechend auch acht.

[5]: Der Ausdruck

```
if (((skipListPerDepth[currentDepth] >> i) & 1)) // [5]
    continue;
```

stellt sicher, dass kein Kind-Knoten mehrfach besucht wird. Dabei ist **skipListPerDepth** ein Array von Integer. Die ersten acht Bits eines Integers einer Tiefe kodieren stellenweise jeweils die in jener Tiefe bereits besuchten Kind-Knoten (acht Bits, weil acht Oktanten und damit acht Kind-Knoten).

Um also zu prüfen, ob der  $i$ -te Kind-Knoten in der  $n$ -ten Tiefe bereits besucht wurde, wird der Integer um  $i$ -Stellen nach rechts verschoben, anschließend wird auf den aus der Verschiebung ermittelte Bit, der bitweise  $\wedge$ -Operator angewandt. Ist das Ergebnis 1, wird der Schleifenkörper an der Stelle nicht weiter fortgesetzt.

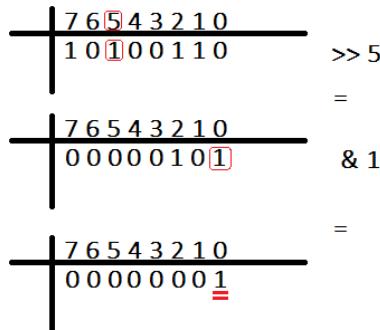


Abbildung 4.10: Exemplarische Vorgehensweise für Index = 5. Der resultierende Bit in der letzten Zeile ist 1, woraus folgt, dass in dieser Tiefe, der dem fünften Index zugewiesene Kind-Knoten bereits untersucht wurde.

[6]: Entspricht dem Vorgehen im Unterabschnitt 3.1.3. Die Position des aktuellen Knotens wird hierarchisch ermittelt und zwischengespeichert.

[7]: Prüft ob der aktuelle Kind-Knoten vom Strahl geschnitten wird.

[8]: Die Ausführung des Codes wird fortgesetzt, wenn die ermittelte Distanz zum Knoten kleiner ist, als die insgesamt ermittelte Distanz (entspricht dem Greedy-Ansatz). Nachfolgend wird dann die aktuell kürzeste Distanz aktualisiert und auf den gerade ermittelt Wert gesetzt.

[9]: Der Konstruktion nach (siehe Abschnitt 4.3) liegt ein Blatt-Knoten vor, wenn dieser  $32 \times 32 \times 32$  groß ist. Falls dem so ist, wird mit [10] der im Abschnitt 4.2 gezeigte Algorithmus auf ein  $32 \times 32 \times 32$  großes Volumen ausgeführt. Im Idealfall werden also etwa  $\frac{\sqrt{512^3}}{\sqrt{32^3}} = 64$ -fach weniger Voxel untersucht.

[10]: Hier wird erstmals der im Abschnitt 4.2 gezeigte Raycasting-Algorithmus ausgeführt.

[11]: Wurde entlang der Richtung des Strahls ein Voxel im Array ermittelt, wird die Variable **voxelFound** auf „true“ gesetzt: Der Algorithmus ist damit prinzipiell abgeschlossen, wird an dieser Stelle aber nicht abgebrochen. Unter Umständen kann es sein, dass Volumina anderer Kind-Knoten derselben Instanz, einen Voxel umfassen, der dem Betrachter näher ist, als der aktuell ermittelte.

[12]: Aufgrund der in [11] dargelegten Problematik, wird die Distanz des aktuell ermittelten Voxels zum Betrachter mit der momentan kürzesten Distanz verglichen. Ist die Distanz des aktuellen Voxels kürzer als die letzte kürzeste Distanz, wird der aktuell ermittelte Voxel als ideal gekennzeichnet.

[13]: Wurde entlang der Richtung des Strahls kein Voxel ermittelt, wird die mit dem aktuell betrachteten Blatt-Knoten korrespondierende minimale Distanz gelöscht, sodass ein anderer Blatt-Knoten mit höherer Distanz zum Betrachter untersucht werden kann.

[14]: Ist die Bedingung in [9] nicht erfüllt, das heißt, ist der aktuell betrachtete (ideale, also zum Betrachter kürzeste Distanz habende) Knoten **kein** Blatt-Knoten, wird der verwendete Index zwischen-gespeichert: Dies ist notwendig, um im Falle eines **false positives**, diesen Knoten bei der Rücktraversierung nicht nochmal zu besuchen (siehe [5]).

[15]: Der nächste Index (also der Index des aktuellen Kind-Knotens) ist die Summe aus dem Index des ersten Kind-Knotens und des relativen Indexes des Kind-Knotens (der mit dem Index des Oktanten korrespondiert).

[16]: Konnte in der aktuellen Tiefe kein Knoten ermittelt werden (ist also der nächste zu betrachtende Eltern-Knoten leer (=**isNull**)), muss der Baum rückwärts traversiert werden (siehe in der Abbildung 4.9 den orangen Pfad). In Zeile [16][0] werden zunächst die besuchten Indizes der aktuellen Tiefe gelöscht, weil diese nicht länger gültig sind. In [16][1] wird die Tiefe dekrementiert (der Baum also rückwärts traversiert). Nachfolgend muss dann der vorherige Zustand wiederhergestellt werden: Der Index des letzten Eltern-Knotens wird durch **parentListPerDepth** ermittelt, der Eltern-Knoten geladen, anschließend die Position und die Größe des Eltern-Knotens wiederhergestellt ([16][4]).

[17]: Wie in [1] bereits vorweggenommen, muss der Index des nächsten zu untersuchenden Knotens als „besucht“ markiert werden, um im Falle eines **false positives**, denselben Knoten bei der rückwärtigen Traversierung nicht doppelt zu untersuchen. Analog zum lesenden Ausdruck in [5], schreibt folgender Ausdruck

```
skipListPerDepth[currentDepth] |= (1 << usedIndex);
```

an die  $n$ -te Stelle eine 1. Dafür wird die Zahl 1, binär um  $n$  Stellen (mit  $n = \text{usedIndex}$ ) nach rechts verschoben und mit dem aktuellen Wert (bitweise mit dem  $\vee$ -Operator) verknüpft. Nachfolgend werden in [17][1] und [17][2] jeweils der Index und die Position des aktuellen Eltern-Knotens zwischengespeichert und schließlich die **currentDepth**-Variable inkrementiert, der Baum also eine Ebene tiefer traversiert.

#### 4.4.3 Komplexitätsanalyse

Um einen Voxel zu ermitteln, liegt die Laufzeit im **Idealfall** jetzt in  $O(\log s)$ , wobei  $s$  die initiale Größe der Octree bezeichnet und mit der Größe der Seite eines Volumens korrespondiert: Die Laufzeit des Raycasting-Algorithmus' aus Abschnitt 4.2 liegt der Konstruktion nach in  $O(1)$ , weil, unabhängig von der variablen Größe  $s$ , immer nur ein 32x32x32 großes Volumen untersucht wird.

Im **schlimmsten** Fall liegt die Laufzeit hingegen in  $O(s \cdot \log s)$ , und zwar dann, wenn entlang der Diagonalen (durch den Volumenmittelpunkt) jeder 32x32x32 große Blatt-Knoten untersucht werden muss:

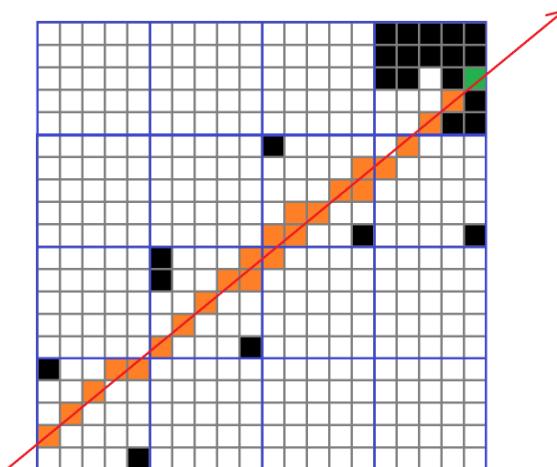


Abbildung 4.11: Jeder Blatt-Knoten entlang der Diagonalen wird untersucht, obschon die meisten keine nennenswerten Daten umfassen: Rauschen in den (Volumen-)Daten senken die Effizienz des Algorithmus' also erheblich.

Die Anzahl der zu untersuchenden Blätter entlang der Diagonalen beträgt:

$$t_{countNodes} = \sqrt{3 \cdot \left(\frac{s}{32}\right)^2} \quad (\text{Satz des Pythagoras})$$

Um jeweils ein Blatt-Knoten zu ermitteln, sind  $\log s$  Schritte notwendig. Zusätzlich muss das 32x32x32 große Volumen untersucht werden (siehe Abschnitt 4.2), sodass insgesamt

$$t_{totalCount} = t_{countNodes} \cdot (\log s + \sqrt{3 \cdot 32^2}) = \sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \log s} + \sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \sqrt{3 \cdot 32^2}}$$

Damit ist

$$\begin{aligned} & \sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \log s} + \sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \sqrt{3 \cdot 32^2}} = \\ & O\left(\sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \log s} + \sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \sqrt{3 \cdot 32^2}}\right) = \\ & O\left(\sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \log s}\right) + O\left(\sqrt{3 \cdot \left(\frac{s}{32}\right)^2 \cdot \sqrt{3 \cdot 32^2}}\right) = \\ & O\left(\sqrt{s^2} \cdot \log s\right) + O\left(\sqrt{s^2} \cdot 1\right) = \\ & O\left(\sqrt{s^2} \cdot \log s\right) = O(s \cdot \log s) \end{aligned}$$

für  $s \rightarrow \infty$ .

Es liegt also nahe, die Blattgröße zu parametrisieren: Je größer die Blattgröße, desto effizienter ist der Algorithmus (insbesondere bei spärlichen Volumendaten) im Idealfall, umso wahrscheinlicher aber sind **false positives** entlang der Richtung des Strahls. Je kleiner die Blattgröße, desto geringer ist die Wahrscheinlichkeit entlang der Richtung des Strahls false positives zu erwischen, umso höher aber ist der Speicherbedarf der Octree.

#### 4.4.4 Divergenzanalyse

Der Code verfügt über mehrere bedingte Abschnitte (zwei verschachtelte **for**-Schleifen, mehrere **if**-Bedingungen). Die obere und die innere Schleife sowie die Bedingungen divergieren aber **fast** nie, denn ähnlich wie in Unterabschnitt 4.2.4 gilt auch hier, dass innerhalb eines Warps, ähnliche Strahlen in die Szene geworfen werden und damit im Idealfall jeder Strahl jeweils ähnliche Ergebnisse produziert. Insgesamt sind hier aber Divergenzen auch deshalb unproblematisch, weil die Anzahl der Iterationen bei rauschfreien Voxel-Daten insgesamt relativ gering ausfällt (siehe Komplexitätsanalyse).

## 4.5 Denoising-Strategie: Medianfilter als Postprocessing-Shader

Aufgrund unvorhersehbarer Präzisionsfehler kann es vorkommen, dass Strahlen keine, oder falsche Ergebnisse ermitteln, sodass die mit dem Strahl korrespondierende Pixelfarbe falsch, oder gar nicht gesetzt wird. Diese können sich im Bild als Rauschen manifestieren, das stark an sogenanntes **Salt and Pepper**-Noise erinnert (siehe fünftes Kapitel, Unterabschnitt 5.1.1).

Um diese zu reduzieren, wird ein sogenannter **Median-Filter** eingesetzt, der auf das durch das Raycasting generierte Bild (also im sogenannten **Postprocessing-Schritt**) angewandt wird.

### 4.5.1 Median-Filter

Beim Median-Filter werden zu jedem Pixel  $p_{i,j}$ , die in einem 3x3-Quadrat umliegenden Pixel ermittelt, in einer Liste gespeichert und diese anschließend sortiert.

Dabei wird im Allgemeinen der Median folgendermaßen definiert:

$$\tilde{x} = \begin{cases} x_{m+1} & \text{für } n = 2 \cdot m + 1 \\ \frac{1}{2} \cdot (x_m + x_{m+1}) & \text{für } n = 2 \cdot m \end{cases}$$

Die beim Median-Filter zu sortierende Anzahl der Pixelfarben beträgt neun, der Median ist damit  $\tilde{x} = x_5$ . Die Farbe des Pixel  $p_{i,j}$  wird dann schließlich auf  $\tilde{x}$  gesetzt.

### 4.5.2 Code

Folgender Code wird im Pixel-Shader nach jedem Raycast ausgeführt.

```
bool compare(float4 a, float4 b)
{
    return dot(GRAY_SCALE_FACTORS, a) > dot(GRAY_SCALE_FACTORS, b); // [6] [0]
}
float4 PostprocessingPixelShader(VertexShaderOutput input) : SV_TARGET
{
    float dx = 1.0f / width; // [0]
    float dy = 1.0f / height; // [0]
    float2 lookups[] = // [1]
    {
        float2(-dx, -dy), float2(0, -dy), float2(dx, -dy),
        float2(-dx, 0), float2(0, 0), float2(dx, 0),
        float2(-dx, dy), float2(0, dy), float2(dx, dy),
    }
}
```

```

};

float4 colorStack[9]; // [2]
int currentIndex = 0;
for (int j = 0; j < 9; j++) // [3]
{
    float4 currentColor = getColor(input.TextureCoordinates, lookups[j]);
    // [4]
    if (currentIndex == 0) // [5]
        colorStack[currentIndex++] = currentColor;
    else
    {
        if (compare(currentColor, colorStack[currentIndex - 1])) // [6]
        {
            float4 temp = colorStack[currentIndex - 1]; // [6] [1]
            colorStack[currentIndex - 1] = currentColor; // [6] [2]
            colorStack[currentIndex] = temp; // [6] [3]
            for (int k = currentIndex - 2; k >= 0; k--) // [6] [4]
            {
                if (compare(currentColor, colorStack[k]))
                {
                    temp = colorStack[k];
                    colorStack[k] = currentColor;
                    colorStack[k + 1] = temp;
                }
            }
            currentIndex++; // [6] [5]
        }
        else
            colorStack[currentIndex++] = currentColor; // [7]
    }
}
return colorStack[4]; // [8]
}
}

```

### 4.5.3 Erläuterung

[0]: Textur-Koordinaten liegen im Pixel-Shader zwischen 0 und 1. Um also auf die umliegende Pixel zuzugreifen, ist der Kehrwert der Bildgröße gleichzeitig jeweils die Schrittweite ( $dx$  und  $dy$ ).

[1]: Um zu jedem Pixel  $p_{i,j}$  auf die, in einem 3x3 großen Quadrat angeordneten, umliegenden Pixel zuzugreifen, wird das Array **lookups** verwendet. Die Elemente des Arrays

sind jeweils Vektoren, die komponentenweise die Schrittweite  $(dx, dy)$  angeben, um auf einen der im  $3 \times 3$  Quadrat liegenden Pixel zuzugreifen.

- [2]: In der Implementierung wird ein Array wie ein (geordneter) Stack verwendet.
- [3]: Die Schleife iteriert durch alle Elemente des **lookups**-Arrays und ermittelt in Zeile [4] anschließend die Farbe des Pixels.
- [4]: Die Textur-Koordinate wird in der Methode **getColor** mit dem  $i$ -ten Element des **lookup**-Arrays addiert und auf diese Weise der umliegende Pixel ermittelt.
- [5]: Ist das Array (= der Stack) leer, wird das Element hinzugefügt, der Stack danach aber nicht sortiert.
- [6]: Ist das Array nicht leer, wird der Stack sortiert, falls der Grauwert der aktuell hinzuzufügende Farbe heller ist, als der des vorherigen Elements. Grauwerte werden hier motiviert, weil, im Unterschied zu Skalaren, im Allgemeinen für 4-Tupel keine Ordnung definiert ist. Die Berechnung des Grauwerts in [6][0] erfolgt auf Grundlage des Skalarprodukts.

In [6][1] bis [6][3] werden zunächst die beiden Elemente vertauscht. Anschließend wird in [6][4] der Stack von der aktuellen bis zur initialen Stack-Tiefe sortiert. Die Schleife iteriert deshalb nicht von 9, sondern von **currentIndex** – 2 bis 0. Die ersten beiden Elemente werden übersprungen, weil diese bereits in [6][1] bis [6][3] sortiert wurden. Innerhalb der Schleife wird dann die Vertauschungsoperation aus [6][1] bis [6][3] wiederholt. Auf diese Weise wird gewährleistet, dass der Stack nach jedem Hinzufügen einer neuen Farbe insgesamt sortiert bleibt. In [6][5] wird schließlich der aktuelle Index inkrementiert: Der Stack ist also um ein Element gewachsen.
- [7]: Ist der Grauwert der aktuell hinzuzufügenden Farbe dunkler als der des vorherigen Elements des Stacks, muss der Stack nicht sortiert werden.
- [8]: Weil der Stack am Ende der Schleife ([3]) bereits sortiert vorliegt, ist der Median und damit die neue Farbe des aktuellen Pixels, anders als oben eingeleitet, das vierte Element von **colorStack**. Dies liegt daran, weil Arrays nullbasiert sind: Das erste Element des Arrays korrespondiert also nicht mit dem ersten, sondern mit dem nullten Index, sodass das fünfte Element nicht mit dem fünften, sondern mit dem vierten Index korrespondiert.

### 4.5.4 Komplexitätsanalyse

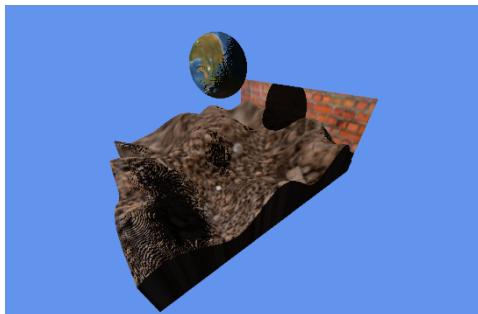
Die Laufzeit des Algorithmus' liegt offensichtlich in  $O(1)$ : Unabhängig von der Größe des zu filternden Bildes, iteriert die Schleife stets durch ein Array der festen Länge neun.

# 5 Ergebnisse und Diskussion

In diesem Kapitel sollen die Ergebnisse der Arbeit gezeigt, die durchschnittliche Bildrate für verschiedene Auflösungen tabellarisch dargestellt und abschließend Problematiken sowie Lösungsansätze diskutiert werden.

## 5.1 Ergebnisse

### 5.1.1 Blockartefakte und Rauschreduktion



- (a) Texture -und Mip-Mapping sowie Schattenwürfe, lassen sich sehr leicht implementieren. Block-Artefakte sind insbesondere vom Nahen sichtbar.

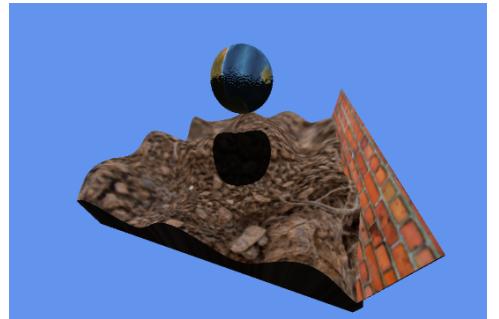
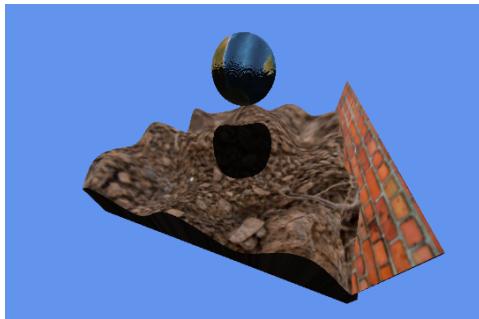


- (b) Links wurde die Szene ohne ein Median-Filter, rechts mit anschließendem Median-Filter gerendert.

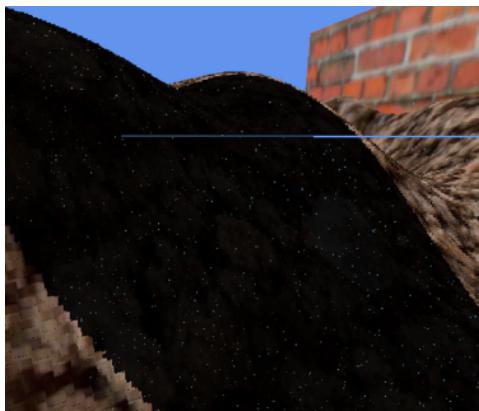
Abbildung 5.1: Mit mittlerer Auflösung (1088x720p) gerenderte Szenen.

### 5.1.2 Beschleunigungsstruktur

Die Verwendung einer Beschleunigungsstruktur ist nicht mit negativen, sondern gar positiven visuellen Veränderungen verbunden, insofern etwa, dass das Rauschen, auch ohne die Verwendung eines Median-Filters, drastisch reduziert wird:



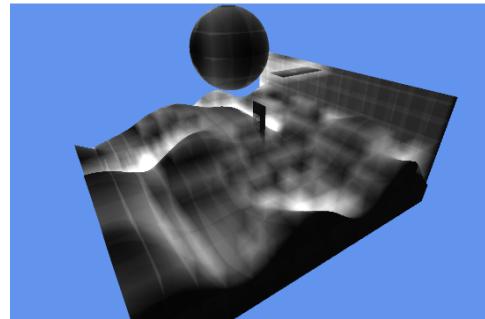
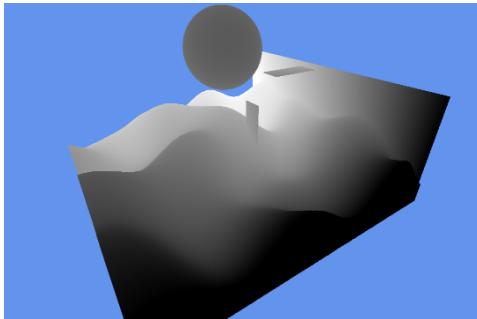
(a) Die Szene links wurde ohne, die Szene rechts mit einer Beschleunigungsstruktur gerendert.



(b) Verwendung einer Beschleunigungsstruktur reduziert Rauschen: Links ohne, rechts mit einer Beschleunigungsstruktur. Man betrachte insbesondere die, aufgrund von Präzisionsfehler entstehende, horizontale Linie im linken Bild, die im rechten Bild praktisch inexistent ist.

Abbildung 5.2: Die Verwendung einer Beschleunigungsstruktur prägt das finale Bild positiv.

Ferner wird die Anzahl der nötigen Iterationen je Strahl bei Verwendung einer Beschleunigungsstruktur deutlich gesenkt, wie folgende Abbildungen zeigen:

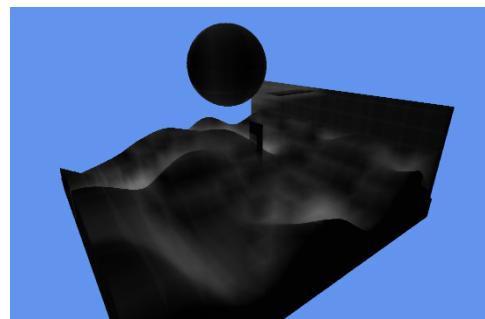
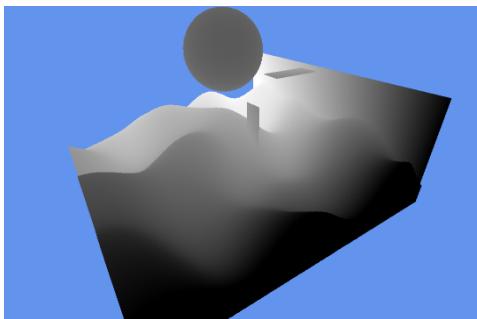


(a) Die Szene links wurde ohne, die Szene rechts mit einer Beschleunigungsstruktur gerendert.

Abbildung 5.3: Die Anzahl der Iterationen wurden farblich kodiert: Je heller der Pixel, desto höher die Anzahl Iterationen, wobei die Skalierungsfaktoren sich unterscheiden.

Die Abbildung rechts macht deutlich, dass bei Verwendung einer Beschleunigungsstruktur insgesamt weniger Iterationen nötig sind. Dabei ist zu beachten, dass die Skalierungsfaktoren sich wesentlich unterscheiden: Während beim Rendern mit Beschleunigungsstruktur die Anzahl der Iterationen im Grunde zwischen 0 und 300 liegt, kann die Anzahl der Iterationen beim Rendern ohne Beschleunigungsstruktur um das 3 bis 4-fache größer ausfallen, also etwa zwischen 0 und 1000 liegen.

Benutzt man denselben Skalierungsfaktor (also  $\frac{1}{1000}$ ), dann werden die Unterschiede besonders deutlich:

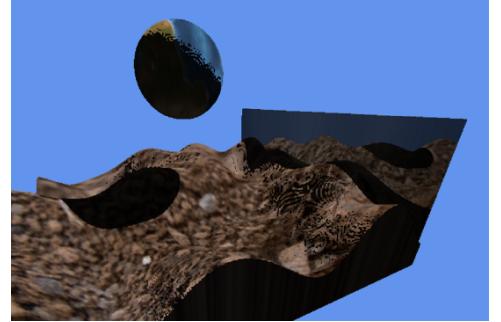
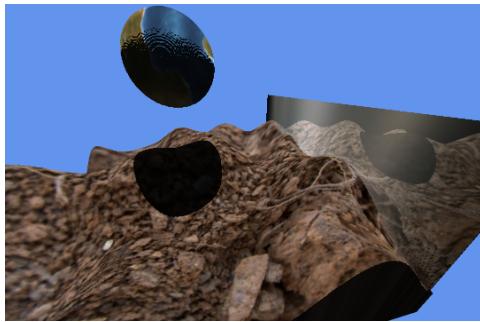


(a) Die Szene links wurde ohne, die Szene rechts mit einer Beschleunigungsstruktur gerendert. Das Bild rechts erscheint dunkler, weil die Anzahl der Iterationen häufiger deutlich unter 1000 liegen.

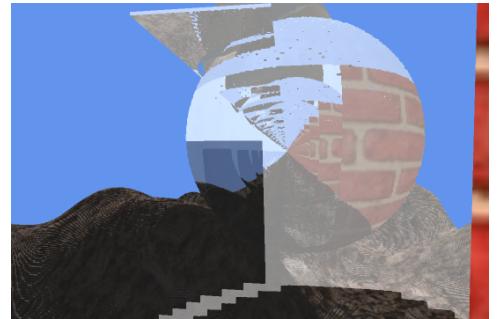
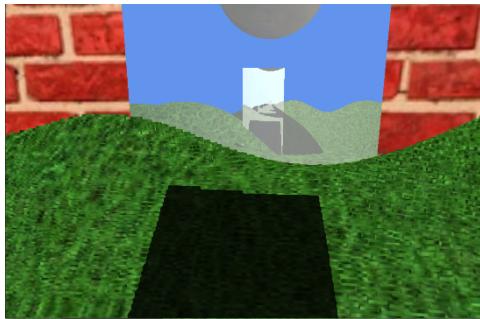
Abbildung 5.4: Die Anzahl der Iterationen wurden mit demselben Skalierungsfaktor ( $\frac{1}{1000}$ ) multipliziert.

### 5.1.3 Raytracing

Der Strahlenlauf kann weiter verfolgt werden ( $\rightarrow$  Raytracing), sodass gar Reflexionen realisiert werden können:



(a) An der silbernen Wand wird die Szene samt Schattenwurf reflektiert.



(b) Wird der Strahlenlauf weiter verfolgt, können gar Reflexionen reflektiert werden.

Abbildung 5.5: Während beim Raycasting lediglich Strahlen in die Szene geworfen werden, wird beim Raytracing-Verfahren die Interaktion der Strahlen mit der Szene weiterverfolgt.

## 5.2 Performance

### 5.2.1 Hardwarespezifikation

Der Raycaster wurde im Fenstermodus auf einem mittlerweile sieben Jahre alten Computer, mit folgenden Spezifikationen ausgeführt:

- Betriebssystem: Windows 10 64 Bit
- CPU: Intel® Xeon® CPU E3-1231 v3 @ 3.4 GHz (4 Kerne, 8 Threads)
- Arbeitsspeicher: 16 GB
- Grafikkarte: NVIDIA GeForce GTX 1060 6GB

Für die Implementierung wurde, wie im zweiten Kapitel unter Unterabschnitt 2.4.2 angemerkt, ein Custom-Monogame-Fork von Markus Hötzinger (Markus Hötzinger (cptmax), o. J.) eingesetzt. Der Compute-Shader wurde ferner mit Shader Model 5.0 kompiliert. Das Testvolumen ist zudem  $512^3$  groß und sieht folgendermaßen aus:

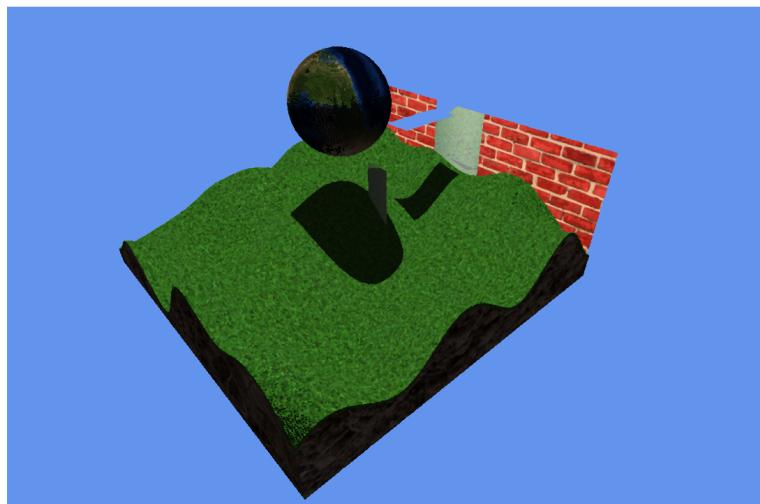


Abbildung 5.6: Die Szene enthält mehrere reflektierende Oberflächen, eine Kugel in der Mitte um den Schattenwurf deutlich sichtbar zu machen, Hügel und eine Wand.

## 5.2.2 Tabellen

Nachfolgend werden die minimale, maximale und durchschnittliche Bildrate pro Sekunde tabellarisch dargestellt.

Tabelle 5.1: Bildraten für Raycasting **ohne** Beschleunigungsstruktur

Bildrate	720x416	1088x720	1920x1088
Minimum <sup>a</sup>	111	69	34
Maximum <sup>a</sup>	224	112	81
Durchschnittlich <sup>a</sup>	153	86	47
Minimum <sup>b</sup>	63	35	19
Maximum <sup>b</sup>	76	49	39
Durchschnittlich <sup>b</sup>	69	41	25
Minimum <sup>c</sup>	59	35	15
Maximum <sup>c</sup>	75	60	39
Durchschnittlich <sup>c</sup>	68	40	24

<sup>a</sup> Keine Reflexionen, keine Schatten

<sup>b</sup> Schatten, keine Reflexionen

<sup>c</sup> Schatten und Reflexionen

Tabelle 5.2: Bildraten für Raycasting **mit** Beschleunigungsstruktur

Bildrate	720x416	1088x720	1920x1088
Minimum <sup>a</sup>	255	125	67
Maximum <sup>a</sup>	371	258	99
Durchschnittlich <sup>a</sup>	297	152	81
Minimum <sup>b</sup>	157	81	38
Maximum <sup>b</sup>	209	111	62
Durchschnittlich <sup>b</sup>	178	91	46
Minimum <sup>c</sup>	78	43	28
Maximum <sup>c</sup>	194	116	52
Durchschnittlich <sup>c</sup>	150	79	35

<sup>a</sup> Keine Reflexionen, keine Schatten

<sup>b</sup> Schatten, keine Reflexionen

<sup>c</sup> Schatten und Reflexionen

## 5.3 Diskussion: Probleme und Lösungsansätze

Die in dieser Arbeit vorgestellte Implementierung ist grundsätzlich nicht nur mit leistungsspezifischen, sondern insbesondere auch mit visuellen Problemen verbunden. Nachfolgend sollen diese gezeigt und mögliche Lösungsansätze diskutiert werden.

### 5.3.1 Raycasting ohne Beschleunigungsstruktur

Wie der Tabelle 5.1 zu entnehmen, ist das reine Raycasting ohne die Verwendung einer entsprechenden Beschleunigungsstruktur, ineffizient, was sich entsprechend auf die Bildrate pro Sekunde deutlich niederschlägt. Die Gründe dafür können einerseits die häufigen Zugriffe auf die Textur-Daten sein, andererseits die relativ hohe Anzahl der Multiplikationen innerhalb der Schleife für mehrere Millionen Strahlen.

Wesentlicher Vorteil dieser Variante ist jedoch, dass Voxel-Modifikationen **direkt** sichtbar werden, ohne das weiterreichende Aktualisierungen, in exemplarisch den Beschleunigungsstrukturen, erfolgen müssen. Liegen mehrere distinkte und vor allem wesentlich kleinere (Voxel-)Entitäten (etwa  $50^3$ , oder  $100^3$  Voxel) mit dichten Voxel-Daten vor, kann sich ein reines Raycasting ohne komplementäre Beschleunigungsstrukturen lohnen.

Bei größeren, oder besonders spärlichen Volumendaten, sollte aber in jedem Fall auf eine Beschleunigungsstruktur zurückgegriffen werden, nicht zuletzt weil, wie in der **Komplexitätsanalyse** im Abschnitt 4.2 dargelegt, die Laufzeit des Raycasting-Algorithmus' in  $O(n)$  liegt.

### 5.3.2 Raycasting mit Beschleunigungsstruktur

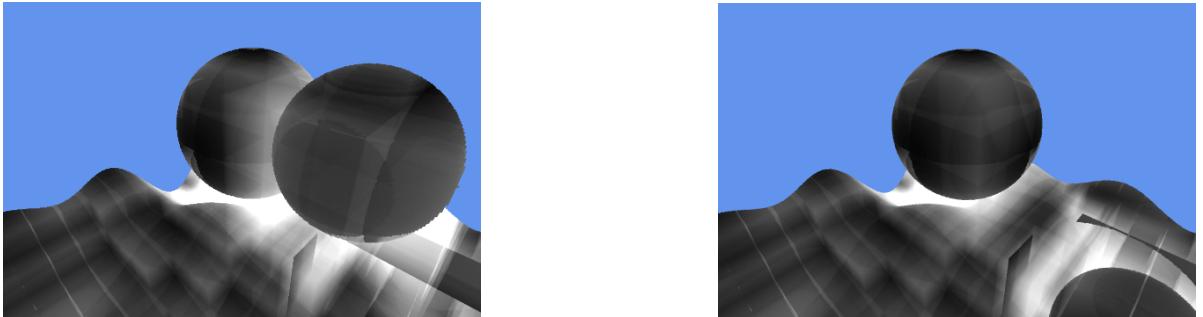
Beim Raycasting mit Beschleunigungsstruktur existieren zwei wesentliche Probleme:

1. Bei Voxel-Modifikationen muss die Beschleunigungsstruktur aktualisiert werden.
2. Data-Racing: Synchronisationsprobleme bei nebenläufigen (schreibenden) Zugriffe auf die Beschleunigungsstruktur (etwa während der Aktualisierung).

Die im Kapitel **Implementierung** gezeigte Beschleunigungsstruktur muss insofern angepasst werden, dass nicht nur die Existenz, sondern nun mehr auch die Anzahl der Voxel innerhalb eines Knotens bekannt sein müssen. Wird ein Voxel innerhalb eines Knotens gelöscht, muss die Anzahl der Voxel dieses Knotens entsprechend dekrementiert werden, um den Knoten, wenn dieser exemplarisch dadurch leer wird, zu deaktivieren

Werden mehrere Voxel desselben Knotens gelöscht, erfolgt während der Aktualisierung des Octree, mehrere nebenläufige (**schreibende**) Zugriffe auf dieselbe Variable (also

Anzahl Voxel), das die Weiterverarbeitung mit inkonsistenten Daten bedingen kann: Nebenläufige Zugriffe auf dieselbe Variable können zu einem bestimmten Grad mit der intrinsischen HLSL-Funktion **InterlockedAdd** synchronisiert werden (die eine atomare Addition gewährleistet). Unter Umständen muss der nebenläufig genutzte Buffer (=StructuredBuffer) als (oder spezifischer: in der Speicherklasse) **globallycoherent** deklariert sein, sodass weiterführende Synchronisationsmechanismen in jedem Fall in allen Warps sichtbar werden (Microsoft Docs HLSL, 2020).



(a) Die Bechleunigungsstruktur wurde erfolgreich aktualisiert: Links die Octree zum Zeitpunkt  $t_0$ , rechts die aktualisierte Octree zum Zeitpunkt  $t_1$ .

Abbildung 5.7: Mit den intrinsischen, atomaren Operationen kann die Octree auch nach Modifikation der Volumendaten in Echtzeit aktualisiert werden.

### 5.3.3 Redundanz

Die im Unterabschnitt 5.1.1 gezeigten Abbildungen stellen eine Voxel-Landschaft dar, die in ein  $512^3$  großes Volumen eingebettet ist. Die maximale Höhe dieser Voxel-Landschaft liegt bei etwa 321 Voxel. Effektiv wird also ein Volumenbereich der Größe  $512 \times 191 \times 512$  nicht benutzt: Dies erhöht die Anzahl der false positives während der Traversierung. Es liegt demnach nahe, die Größe des Volumens auf die Ausmaße der Voxel-Daten anzupassen.

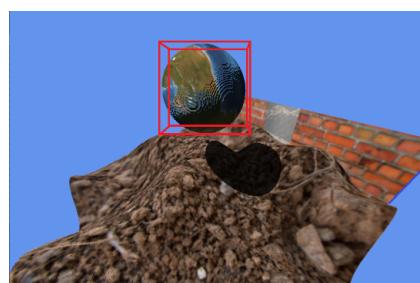


Abbildung 5.8: Die (Voxel-)Kugel liegt hier nicht innerhalb des  $512^3$  großen Volumens, sondern in einem separaten Volumen.

Dieser Ansatz hat einerseits den Vorteil, dass so (a) zwischen Entitäten und der Spielwelt unterschieden werden kann, was (auf Voxel basierende) physikalische Simulationen ermöglicht, andererseits (b) in den Grafikspeicher nicht mehr geladen werden muss, als unbedingt nötig.

Punkt (a) ist insofern zu verstehen, dass Volumina die (dichte) Voxel-Modelle beinhalten, als Bounding-Box eines starren Körpers interpretiert werden können, wobei die Voxel dieses Volumens sowohl die Kontaktpunkte, als auch die Massenteile des starren Körpers darstellen könnten. Punkt (b) erscheint insbesondere wieder vor dem Hintergrund der in den Abbildungen dargestellten Voxel-Landschaft plausibel: In den Grafikspeicher wird eine  $512^3$  große dreidimensionale Textur geladen, obschon die Landschaft aus  $512 \times 321 \times 512$  Voxel besteht.

#### 5.3.4 Beleuchtungsprobleme

Ein wesentlicher Nachteil von AABBs ist, dass die Normalen der Würfelflächen keine visuell ansprechende Beleuchtung erlauben:

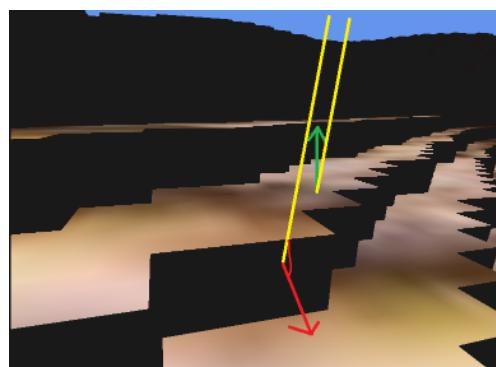


Abbildung 5.9: Die Strahlen der Leuchtkugel schneiden die rote Normale nahezu senkrecht, sodass die Seite des Voxels dunkel, während die grüne Normale in einem sehr kleinen Winkel geschnitten wird und die Seite des Voxels entsprechend stark beleuchtet erscheint.

Die Beleuchtung wirkt „hart“ und physikalisch nicht plausibel: Zu erwarten wäre eher eine graduelle Abnahme der Helligkeit an den Seiten. Ein möglicher Lösungsansatz ist es, nachdem ein Voxel ermittelt wurde, die Normale der geschnittenen Würfelfläche auf Basis der lokalen Nachbarschaft zu bestimmen.

### 5.3.5 Texturemapping

Soll eine zweidimensionale Textur auf ein Voxel-Modell abgebildet werden, liegt ein ähnliches Problem wie im Unterabschnitt 5.3.4 vor: Die Seiten des Voxels zeigen jeweils einen unterschiedlichen Ausschnitt der Textur, was zu harten Kanten führt:

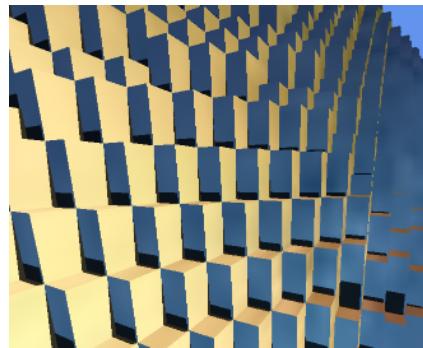


Abbildung 5.10: Derselbe Voxel stellt unterschiedliche Ausschnitte einer Textur dar.

Um bei der Texturierung der Voxel harte Kanten zu verwischen, könnten die Normalen entsprechend des Einheitskreises ausgerichtet und das sogenannte Tri-Planar-Mapping verwendet werden:

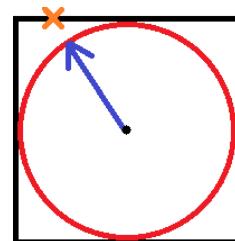


Abbildung 5.11: Vom Schnittpunkt zum Würfelmittelpunkt wird eine Gerade gezogen:  
So entsteht die neue Normale.

Wird ein Voxel ermittelt, ist die Normale dieser dann die vom Voxelmittelpunkt zum Schnittpunkt gezogene Gerade.

### 5.3.6 Schatten

Bei der Darstellung von Schatten können aufgrund der einfachen Präzision von float-Gleitkommazahlen, Artefakte wie diese entstehen:

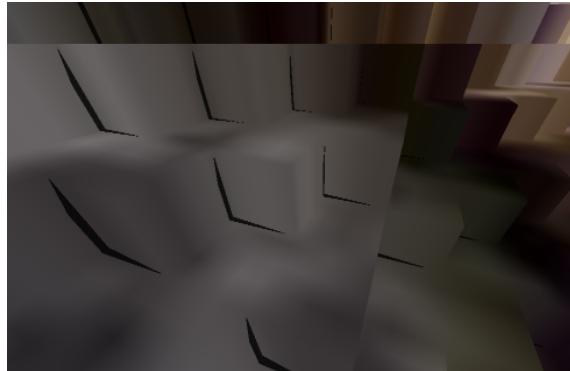


Abbildung 5.12: Die Leuchtquelle scheint genau senkrecht auf die Voxel, dennoch sind an den zur Leuchtquelle gerichteten Seiten Schatten eindeutig sichtbar.

Eventuell würde die Verwendung von Gleitkommazahlen mit doppelter Präzision (also double) bessere Ergebnisse erzielen, die Effizienz der verwendeten Algorithmen würde aber deutlich abnehmen: Nahezu alle Grafikkarten sind für Operationen mit Gleitkommazahlen einfacher Präzision ausgelegt.

### 5.3.7 Flackern bei Modifikation zusammenhängender Voxel-Daten

Soll ein sich auf der Position  $\vec{p}_i$  befindlicher Voxel  $\mathbf{v}_i$  entlang der Richtung  $\vec{h}_i$  verschoben werden, dann muss dieser Voxel an der Position  $\vec{p}_i$  gelöscht und auf die neue Position  $\vec{p}_i + \vec{h}_i$  gesetzt werden. Werden in jedem Frame mehrere zusammenhängende Voxel verschoben, finden unter Umständen nebenläufige - schreibende und löschende - Zugriffe auf dieselbe Position statt, sodass exemplarisch ein verschobener (geschriebener) Voxel  $\mathbf{v}_a$  von einem noch zu verschiebendem Voxel  $\mathbf{v}_b$  gelöscht wird, was sich als Flackern äußern kann:

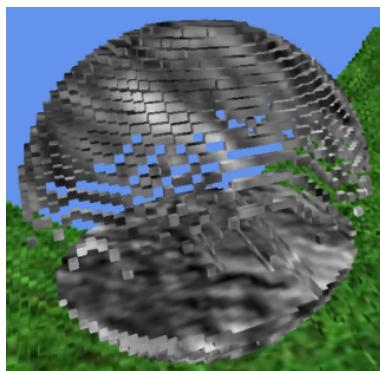


Abbildung 5.13: Mehrere nebenläufige, schreibende und löschende Zugriffe auf dieselben Volumenbereiche, können sich in den Frames als Flackern äußern.

## *5 Ergebnisse und Diskussion*

---

Das Problem lässt sich bis zu einem bestimmten Grad lösen, indem das Volumen initial dupliziert wird ( $\mathbf{Volume}_{old}$  und  $\mathbf{Volume}_{new}$ ) und beim Lesen und Schreiben jeweils auf den beiden Volumendaten gearbeitet wird: Das Volumen  $\mathbf{Volume}_{old}$  wird verwendet, um den Voxel an der Position  $\vec{\mathbf{p}}_i$  auszulesen und zu löschen, das Volumen  $\mathbf{Volume}_{new}$  hingegen verwendet, um den Voxel an die neue Position  $\vec{\mathbf{p}}_{i+1}$  zu schreiben.

In jedem Frame werden dann vor den Modifikationen  $\mathbf{Volume}_{old}$  und  $\mathbf{Volume}_{new}$  getauscht, sodass nebenläufig schreibende und löschrifte Zugriffe auf dieselbe Position praktisch unterbunden werden.

Der Lösungsansatz ist aber kritisch zu betrachten, weil das Volumen in den Grafikspeicher doppelt geladen werden muss und damit zusätzlich Speicherplatz in Anspruch genommen wird.

Alternativ bietet es sich an, komplexe Synchronisationsmechanismen einzuführen, um sichere, nebenläufige Zugriffe auf das Volumen zu gewährleisten.

# 6 Fazit und Ausblick

Die Bachelorarbeit hatte das Ziel, Volumendaten effizient zu visualisieren, ohne diese in ein alternatives Repräsentationsformat zu überführen. Die Arbeit machte deutlich, dass es wesentlich leistungseffizienter ist, wenn sowohl bei der Modifikation, als auch beim Rendering auf dieselben Daten zurückgegriffen wird. Es wurden Schritt für Schritt Ansätze erarbeitet, um dieses Vorhaben zu realisieren, etwa Volumina und Voxel im Kontext dieser Arbeit mathematisch zu präzisieren, Beschleunigungsstrukturen einzuführen, den Begriff SIMD zu erklären, die Grafikkartenarchitektur anzuschneiden, diese in Zusammenhang von Vektorprozessoren einzuordnen und Grundlagen des Raycastings zu thematisieren.

Es wurden verschiedene Algorithmen und Datenstrukturen vorgestellt, ihre theoretische Grundlage erklärt, diese implementiert, wichtige Code-Fragmente gezeigt, die Funktionsweise dieser ausführlich dargelegt und das Laufzeitverhalten der Algorithmen analysiert.

Die Arbeit hat gezeigt, dass Voxel-Volumen der Größe  $512^3$  in Echtzeit mit einem Raycasting-Ansatz visualisiert werden können, indem Volumina als dreidimensionales Array und Voxel als Elemente dieses Arrays betrachtet werden, wobei das Array von je einem Strahl für jeden Pixel iterativ traversiert wird. Dabei wurde, angelehnt an die Idee von Amanatides und Woo, in einer Iteration jeweils immer nur ein Voxel betrachtet. Die Arbeit ergänzte diese Idee zudem um eine Beschleunigungsstruktur, sodass die Laufzeit der Traversierung erkennbar reduziert werden konnte.

Der in dieser Arbeit vorgestellte Ansatz hat zusätzlich den Vorteil, dass das dreidimensionale Array zur Laufzeit beliebig bearbeitet werden kann und Änderungen direkt sichtbar werden, auch dann, wenn eine Beschleunigungsstruktur eingesetzt wird, wie im vorherigen Kapitel im Unterabschnitt **Raycasting mit Beschleunigungsstruktur** gezeigt werden konnte.

Im Verlauf der Arbeit wurden im Kapitel **Ergebnisse und Diskussion** auch Nachteile, insbesondere visueller Art, deutlich. Neben den Blockartefakten, die besonders vom Nahen sichtbar sind, existieren wesentliche Probleme bei der Beleuchtung, oder Texturierung von Voxelflächen, die aber unter Umständen durch zusätzlichen Aufwand, wie die Bereitstellung weiterführender Daten (wie exemplarisch Konturdaten: Unterabschnitt 3.1.2), oder Analyse der unmittelbaren Nachbarschaft der Voxels, zu einem bestimmten Grad gelöst werden könnten.

## *6 Fazit und Ausblick*

---

Die aktuelle Implementierung des Raycasting-Algorithmus' kommt zwar wesentlich ohne Divisionen und Verzweigungen aus, verfügt aber in einer Schleife über mehrere Additionen und Multiplikationen, das die Leistung des Raycasters für mehrere Millionen Strahlen pro Sekunde insgesamt erheblich senken kann.

Die Darstellung der Voxeldaten als dreidimensionale Array ist ferner mit Redundanzen verbunden. Hinzu kommt, dass Arrays der Grafikkarte als dreidimensionale Texturen übergeben werden müssen, sodass die Größe der Arrays, Limitationen der DirectX - API unterliegen. Es sollte deshalb in Erwägung gezogen werden, große Voxel-Volumen in kleinere Volumen (sogenannte Chunks) zu unterteilen und diese ähnlich wie in einer Octree anzugeben. Auf diese Weise könnten dann auch immer nur die Volumen-Chunks im Speicher gehalten werden, die tatsächlich im Sichtfeld des Betrachters liegen, und nur dann weitere Chunks nachgeladen werden, wenn dies unbedingt nötig ist. Unter Umständen könnte der Grafikspeicher so zusätzlich entlastet werden.

Zudem liegt es nahe, die Machbarkeit physikalischer Simulationen, insbesonderer starrer Körper, mit diesem Ansatz zu prüfen, indem Voxel-Volumen als starre Körper und Voxel als Kontakt -und Massenpunkte des Körpers interpretiert werden könnten. Bei der Kollisionserkennung könnten Voxels dann vereinfacht als Kugeln, statt als Quader betrachtet werden, um den Kontaktspunkt zweier Körper schnell und effizient zu ermitteln. Der Massenschwerpunkt des Körpers wäre dann der Durchschnitt der Positionen der Voxel innerhalb des Volumens. Sind der Kontaktspunkt und der Massenschwerpunkt des Körpers bekannt, ließe sich auch der Drehmoment entsprechend leicht berechnen.

Die Arbeit hat auch gezeigt, dass Reflexionen, Reflexionen von Reflexionen und Schatten, mit wenig Aufwand realisiert werden können. Voxel-Volume-Raycasting könnte also ebenso Anwendung in globaler Beleuchtung von auf Polygon basierenden Szenen finden, indem Polygon-Modelle während des Rasterisierungsschrittes in der GPU voxelisiert und die Beleuchtung der Pixels anschließend auf Grundlage der in dieser Arbeit vorgestellten Ansätze berechnet werden.

## Literatur

- 2.1.2.2 *von-neumann-flaschenhals*. (o. J.). Zugriff auf <https://vfhcab.eduloop.de/loop/Von-Neumann-Flaschenhals>
- Amanatides, J., Woo, A. et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Eurographics* (Bd. 87, S. 3–10).
- Bouknight, W. J. (1970). A procedure for generation of three-dimensional half-toned computer graphics presentations. *Communications of the ACM*, 13 (9), 527–536.
- Chris Gyurgyik. (2020, Feb). *An overview of the fast voxel traversal algorithm*. Zugriff auf <https://github.com/cgyurgyik/fast-voxel-traversal-algorithm/blob/master/overview/FastVoxelTraversalOverview.md>
- Game that makes full use of 3d cg – future space war: Interstellar from funai. (1983). Zugriff auf <https://onitama.tv/gamemachine/pdf/19831215p.pdf>
- Graphics processing unit (gpu)*. (o. J.). Zugriff auf <https://web.archive.org/web/20160311065240/http://www.nvidia.com/object/gpu.html>
- Hofmann, K. (2019). Minecraft as AI playground and laboratory. In *Proceedings of the annual symposium on computer-human interaction in play* (S. 1–1).
- Intel® intrinsics guide*. (o. J.). Zugriff auf <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>
- Introduction to gpgpu and cuda programming: Thread divergence*. (o. J.). Zugriff auf [https://cvw.cac.cornell.edu/gpu/thread\\_div](https://cvw.cac.cornell.edu/gpu/thread_div)
- Laine, S. & Karras, T. (2011). Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17 (8), 1048–1059. doi: 10.1109/TVCG.2010.240
- Lorensen, W. E. & Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21 (4), 163–169.
- Markus Hötzinger (cpt-max). (o. J.). *Custom monogame fork*. Zugriff auf <https://github.com/cpt-max/MonoGame>
- Medin, J. & Persson, E. (2009). Seven generations of gaming.
- Microsoft Docs HLSL. (2020, Aug). *Win32/sm5-object-rwstructuredbuffer.md at docs · microsoftdocs/win32*. Zugriff auf <https://github.com/MicrosoftDocs/win32/blob/docs/desktop-src/direct3dhlsl/sm5-object-rwstructuredbuffer.md>
- Microsoft Docs HLSL (Stevewhims). (o. J.). *Function declaration syntax - win32 apps*. Zugriff auf <https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-function-syntax>
- Nvidia's next generation cuda(tm) compute architecture: Fermi*. (o. J.). Zugriff auf [https://www.nvidia.com/content/PDF/fermi\\_white\\_papers/NVIDIA\\_Fermi\\_Compute\\_Architecture\\_Whitepaper.pdf](https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf)
- Pfister, H. (1999, 10/1999). Volume pro - at the frontier of advanced volume graphics. *Nikkei Science*.

## *6 Fazit und Ausblick*

---

- Wikipedia contributors. (2022). *Volume ray casting — Wikipedia, the free encyclopedia.* [https://en.wikipedia.org/w/index.php?title=Volume\\_ray\\_casting&oldid=1075718373](https://en.wikipedia.org/w/index.php?title=Volume_ray_casting&oldid=1075718373). ([Online; accessed 19-March-2022])
- Wloka, M. (2003). Batch, batch, batch: What does it really mean. In *Presentation at game developers conference*.
- Zou, X., Xu, S., Chen, X., Yan, L. & Han, Y. (2021). Breaking the von neumann bottleneck: architecture-level processing-in-memory technology. *Science China Information Sciences*, 64 (6), 1–10.