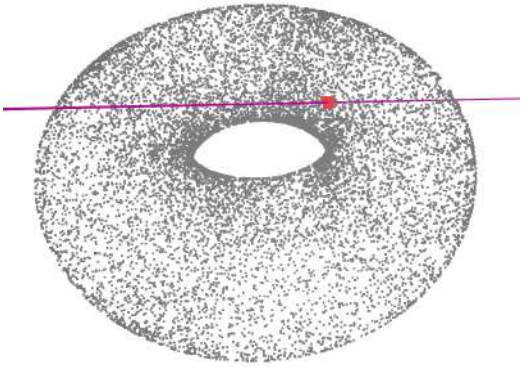
Universität Bremen

Fachbereich 3: Mathematik und Informatik

Bachelorarbeit

Optimierung von Raycastingalgorithmen mit dynamischen Surface Point Clouds auf der GPU



Kevin Ahlering

14. Januar 2019

1. Gutachter: Prof. Dr. Gabriel Zachmann
2. Gutachter: Prof. Michael Beetz PhD
Betreuer: M.Sc. Philipp Dittmann

Kevin Ahlering

Optimierung von Raycastingalgorithmen mit dynamischen Surface Point Clouds auf der GPU

Bachelorarbeit, Fachbereich 3: Mathematik und Informatik

Universität Bremen, Januar 2019

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Bremen, den 14. Januar 2019

_____

Kevin Ahlering

Danksagung

Ich bedanke mich bei AG Computergrafik und Virtuelle Realität dafür, dass mir ein Rechner zum Ausführen meiner Testfälle zur Verfügung gestellt wurde. Außerdem möchte ich mich bei Philipp Dittmann bedanken, dessen Feedback eine große Hilfe bei der Erstellung dieser Arbeit war.

Abstract

Diese Bachelorarbeit befasst sich mit der Berechnung des Schnittpunktes eines Strahls mit einer Oberfläche, die durch eine dynamische 3D Punktwolke ohne Oberflächennormalen repräsentiert wird. Es wird geprüft, ob der Schnittpunkt in Echtzeit berechnet werden kann. Dynamische Punktwolken erschweren die Verwendung von Beschleunigungsdatenstrukturen, da für jede neue Punktwolke die Beschleunigungsdatenstruktur aktualisiert werden muss. Aus diesem Grund wird untersucht, ob mithilfe von CUDA auf das Erstellen von Suchstrukturen verzichtet werden kann. Diese Untersuchung hat ergeben, dass dies der Fall ist, wenn nur wenige Schnittpunkte pro Punktwolke berechnen werden, da der Aufwand zum Erstellen der Suchstrukturen den zusätzlichen Aufwand zur Schnittpunktberechnung überwiegt. Werden hingegen viele Schnittpunkte pro Punktwolke berechnet, ist der Ansatz nicht geeignet, weil der Aufwand pro Schnittpunktberechnung höher ist. Mit dem in dieser Arbeit entwickelten Ansatz lassen sich Schnittpunkte mit Punktwolken von bis zu $2,5 \cdot 10^6$ Punkten zuverlässig in Echtzeit berechnen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel werden die Motivation und Relevanz für diese Arbeit und das Thema vorgestellt. Außerdem wird eine Übersicht über die weiteren Kapitel gegeben.

1.1 Motivation und Relevanz

Punktwolken ermöglichen es Objekte oder ganze Umgebungen in 3D darzustellen und mit Sensoren wie der Microsoft Kinect ist es möglich in Echtzeit eine Punktwolke der Umgebung zu erstellen. Punktwolken werden z.B in der Robotik verwendet, um Informationen über die Umgebung des Roboters zu erhalten. Des Weiteren könnte man Umgebungen in einer Virtual Reality durch Punktwolken darstellen. In beiden Fällen gibt es viele Situationen, in denen es nützlich ist, den Schnittpunkt eines Strahls mit der Punktwolke in Echtzeit bestimmen zu können. Möglicherweise will man Objekte in einer Virtual Reality Umgebung durch zeigen markieren oder prüfen, ob ein Pfad durch ein Hindernis blockiert ist. In einem ähnlichen Anwendungsfall könnte man prüfen, ob eine Sichtverbindung zwischen zwei Punkten besteht. Außerdem könnte man so Punktwolken für Entfernungsbestimmungen benutzen, indem man die Distanz vom Startpunkt des Strahl bis zum Schnittpunkt misst. Dabei ist zu beachten, dass sich die Umgebung und somit auch die Punktwolke jederzeit ändern kann. Aus diesem Grund wird in dieser Arbeit geprüft, ob es möglich ist einen Schnittpunkt zwischen einem Strahl und einer sich ändernden Punktwolke in Echtzeit zu bestimmen.

1.2 Ziel der Arbeit

In dieser Arbeit wird geprüft, ob man den Schnittpunkt eines Strahls mit einer sich ändernden Punktewolke in Echtzeit bestimmen kann. Durch die dynamische Punktwolke wird die Verwendung von Beschleunigungsdatenstrukturen erschwert, weil diese mit jeder neuen Punktwolke aktualisiert werden müssen. Des Weiteren wird davon ausgegangen, dass keine Oberflächennormalen für die Punktwolke zur Verfügung stehen und dass die Punktdichte auf der Oberfläche überall ungefähr gleich groß ist. Außerdem wird die Verwendung von CUDA zur Parallelisierung der Rechenschritte in diesem Zusammenhang getestet und mit einer sequenziellen Implementierung verglichen. Dabei wird untersucht, wie sich die Laufzeit einer CUDA-Implementierung ohne Beschleunigungsdatenstrukturen gegen eine sequenzielle CPU-Implementierung mit Beschleunigungsdatenstrukturen, in verschiedenen Situationen verhalten, um schlussendlich festzustellen, ob durch die Verwendung von CUDA auf das Erstellen von Beschleunigungsdatenstrukturen verzichtet werden kann. Dabei muss abgewägt werden, ob der zusätzliche Aufwand bei der Schnittpunktberechnung die Einsparung durch den Wegfall der Beschleunigungsdatenstruktur wert ist.

1.3 Aufbau der Arbeit

Nach dieser Einleitung folgen in Kapitel 2 die Grundlagen dieser Arbeit. Zu diesen gehören die verwendeten Datenstrukturen und Algorithmen. In Kapitel 3 wird dann der entwickelte Algorithmus zur Berechnung des Schnittpunktes zwischen einem Strahl und einer Punktwolke erklärt, dessen genaue Implementierung Kapitel 4 folgt. Daraufhin werden die Laufzeiten der Implementierung in Kapitel 5 getestet und ein abschließendes Fazit in Kapitel 6 gegeben.

Kapitel 2

Stand der Forschung

Nachdem in dem vorherigen Kapitel das Thema der Arbeit vorgestellt wurde, werden in diesem Kapitel die in der Arbeit angewendeten Konzepte und Technologien vorgestellt. Zu diesen gehören Punktwolken, Ray Tracing, verschiedene Beschleunigungsdatenstrukturen und CUDA. Außerdem wird geklärt, was mit dem Begriff Echtzeit in dieser Arbeit gemeint ist. Im nächsten Kapitel wird der für diese Arbeit entwickelte Ansatz zu Berechnung des Schnittpunktes eines Strahls mit einer Punktwolke erklärt.

2.1 Punktwolken

Laut Martin Weinmann [Wei16] beschreibt der Begriff Punktwolke eine Menge an Punkten in einem bestimmten Raum. Der Raum und die Punkte können dabei beliebig viele Dimensionen haben, allerdings werden in dieser Arbeit nur Punktwolken aus dem dreidimensionalen Raum betrachtet, da diese von Sensoren wie der Microsoft Kinect generiert werden, um die Geometrie der Umgebung darzustellen. In Abbildung 2.1 ist als Beispiel die Punktwolke eines Torus zu sehen. Die Punkte liegen dabei auf der Oberfläche der gescannten Objekte und werden durch die XYZ-Koordinaten gekennzeichnet. Nach Martin Weinmann [Wei16] können die Punkte von Punktwolken noch weitere Attribute wie z.B. Normalen oder RBG-Werte für Farben haben.

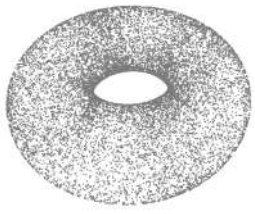


Abbildung 2.1 Punktwolke eines Torus

2.2 Ray Tracing

In der Computergrafik bezeichnet Ray Tracing eine Renderingtechnik [Wel13], bei der für jeden Pixel von einem Sichtpunkt aus ein Strahl verfolgt wird. Wenn ein Strahl auf eine Oberfläche trifft, so wird von dort ein weiterer Strahl zu der Lichtquelle geschossen und reflektierte und gebrochene Strahlen werden rekursiv weiter verfolgt. Da die Strahlen normalerweise als unendlich dünn betrachtet werden und sie beim Reflektieren und Brechen der Strahlen nicht verschwimmen, entstehen auf diese Weise unnatürlich scharfe Bilder. [WW92]

Die Herausforderung beim Ray Tracing ist hauptsächlich Schnittpunkte von Strahlen mit der Szene zu berechnen [Wel13]. Aus diesem Grund ist Ray Tracing dem Problem dieser Arbeit sehr ähnlich, da auch hier der Schnittpunkt eines Strahls mit der Szene berechnet wird. Allerdings geht es in dieser Arbeit nicht darum Bilder zu generieren, sondern nur den ersten Schnittpunkt eines Strahls mit der Szene zu berechnen. Die Szene wird in dieser Arbeit als Punktwolke dargestellt und deshalb werden im Folgenden zwei Ansätze für Ray Tracing auf Punktwolken vorgestellt.

2.2.1 Ray Tracing auf Punktwolken

Das Problem bei Ray Tracing auf Punktwolken ist, dass die Oberfläche nur durch Punkte repräsentiert wird und so der Schnittpunkt nicht direkt berechnet werden kann. Aus diesem Grund haben sich laut Linsen et al. [Lin07] zwei Hauptherangehensweisen für Ray Tracing auf Punktwolken entwickelt, nämlich der surface-splatting Ansatz und der point-set-surface Ansatz. Beim surface-splatting Ansatz werden an jedem Punkt die Oberflächennormalen und eine Scheibe berechnet, die tangential zur Oberfläche ist. Die Scheiben stellen die Oberfläche der Punktwolke dar und sollen sich überschneiden, um eine geschlossene Oberfläche zu bilden, sodass mit deren Hilfe der Schnittpunkt berechnet werden kann. Point-set-surface basierte Ansätze sind nach Linsen et al. [Lin07] in der Regel viel aufwendiger. Da der Aufwand für die Echtzeitbedingung in dieser Arbeit eine wichtige Rolle spielt, werden sich zwei surface-splatting basierte Ansätze angeschaut.

Der erste Ansatz ist von Linsen, Müller und Rosenthal [Lin07], die mit der Generierung der Scheiben beginnen. Es wird mit einer Scheibe an einem beliebigen Punkt p_i angefangen, die, wie in Abbildung 2.2 zu sehen ist, iterativ durch das Einbeziehen von Nachbarn von p_i vergrößert wird, bis der Fehlerwert δ_ε zwischen den Punkten und der Scheibe einen vordefinierten Wert überschreiten. Es wird nicht für jeden Punkt eine Scheibe generiert, weil durch das iterative Vergrößern schon die Nachbarn von p_i mit abgedeckt sind. Die erstellten Scheiben werden daraufhin zu einem Octree hinzugefügt. Der Schnittpunkt wird so berechnet, dass ausgehend vom Startpunkt des Strahls alle Scheiben im ersten vom Strahl getroffenen Blattknoten des Octrees, geprüft werden, ob diese auch vom Strahl getroffen werden. Wenn keine Scheibe getroffen wird,

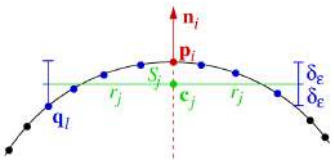


Abbildung 2.2 Das Erstellen der Scheibe S_j mit dem Radius r_j beginnt am Punkt p_i der Punktwolke und vergrößert sie, indem iterativ die Nachbarn q_l von p_i mit einbezogen werden, bis der Fehlerwert δ_ε einen gewählten Wert überschreitet.[Lin07]

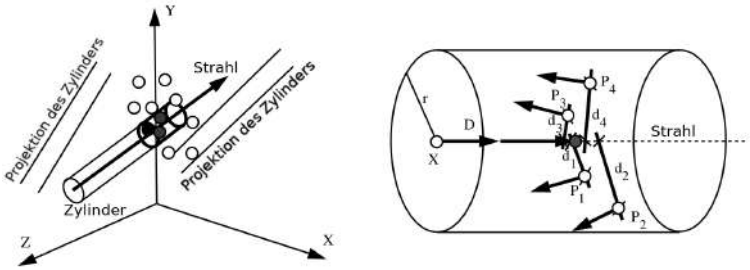


Abbildung 2.3 Links: Der von einem Zylinder umschlossene Strahl. Die erste Scheibe an einem Punkt, die vom Strahl geschnitten wird, löst einen Schnittpunkt aus. Von diesem aus wird ein weiterer Zylinder gestartet und die Position, Normale und andere Eigenschaften des Schnittpunktes werden aus allen Punkten in dem neuen Zylinder interpoliert. Rechts: Berechnung des Schnittpunkts (grau) mit den vier Nachbarpunkten in dem neuen Zylinder mit dem Strahl $X + t \cdot D$. Die Normale und Position eines Punktes P_i wird nach Abstand d_i zum Strahl gewichtet. [SJ00]

wird mit dem nächsten Blattknoten fortgefahren. Wenn ein oder mehrere Scheiben getroffen werden, wird der Schnittpunkt genommen, der am dichtesten am Zentrum seiner Scheibe liegt.

Ein weiterer Ansatz für Ray Tracing auf Punktwolken ist von Schaufler und Jensen [SJ00]. Es gibt um jeden Punkt der Punktwolke eine Scheibe, deren Normale der Oberflächennormale des Punktes entspricht. Es wird davon ausgegangen, dass Oberflächennormalen für jeden Punkt zur Verfügung stehen. Der Radius r der Scheibe ist für jeden Punkt gleich und wird so gewählt, dass keine Löcher in der Oberfläche entstehen. Einen Schnittpunkt existiert, wenn der Strahl eine der Scheiben schneidet (siehe Abbildung 2.3 links). Um das Ganze zu beschleunigen, verwenden Schaufler und Jensen einen Octree. Dabei werden nur die Knoten des Octrees untersucht, die den Zylinder mit dem Radius r schneiden. Sobald eine der Scheiben geschnitten wird, wird ausgehen von dem Schnittpunkt ein weiterer Zylinder gestartet. Alle Punkte, die in diesem Zylinder liegen, werden verwendet im den endgültigen Schnittpunkt zu bestimmen (siehe Abbildung 2.3 rechts).

Nachdem nun zwei Verfahren für Ray Tracing auf Punktwolken vorgestellt wurden, behandelt der nächste Abschnitt die Nearest-Neighbors-Suche (NN-Suche).

2.3 Nearest-Neighbors-Suche

Bei der NN-Suche geht es darum die Punkte P_c aus einer Menge P an Punkten zu finden, die nach einer Distanzfunktion $d(q, p)$ am dichtesten zu einem Suchpunkt q sind. Die Distanzfunktion muss dabei folgende Eigenschaften haben:[KZN08]

Symmetrie: $d(a, b) = d(b, a)$
Positive Definitheit: $d(a, a) = 0$ und $d(a, b) > 0, a \neq b$
Dreiecksungleichung: $d(a, b) \leq d(a, c) + d(b, c)$

Die NN-Suche spielt für diese Arbeit eine Rolle, weil zur Schnittpunktberechnung die Nachbarn von einigen Punkten gefunden werden müssen und da es sich in dieser Arbeit um eine dynamische Punktwolke handelt, ist vor allem die Erstellungszeit der benötigten Datenstrukturen zur NN-Suche wichtig. Neeraj Kumar, Li Zhang und Shree Nayar haben einige Verfahren zur NN-Suche im Zusammenhang mit der Suche nach ähnlichen Bereichen in Bildern untersucht und sind zu dem Ergebnis gekommen, dass sich K-d Trees und Vantage-point Trees schnell erstellen lassen [KZN08] (siehe Tabelle 2.1). Deshalb wurden diese Datenstrukturen genauer untersucht. Außerdem wurden Octrees betrachtet, da Schaufler und Jensen einen Octree in ihrer Implementierung für Ray Tracing auf Punktwolken verwenden. Im folgenden werden diese Datenstrukturen vorgestellt.

Methode	**Erstellungzeit**	**Radius Nachbarsuche**	**k-NN Suche**
K-d-Tree	exzellent	schlecht	schlecht
PCA Tree	schlecht	mittelmäßig	mittelmäßig
Ball Tree	mittelmäßig	exzellent	exzellent
k-Means	schlecht	gut	gut
Vp-Tree	Exzellent	exzellent	exzellent

Tabelle 2.1 Zusammenfassung der Ergebnisse von Kumar et al.[KZN08] zu der Erstellungs- und Suchzeit verschiedener Beschleunigungsdatenstrukturen.

2.3.1 Octree

Ein Octree ist ein Suchbaum, bei dem jeder interne Knoten acht Kindknoten besitzt. Jeder Knoten in dem Baum entspricht einem Achsen-ausgerichteten 3D Würfel. Die Kinder des Knotens v entsprechen den Würfel der Oktanten aus der Abbildung 2.4.[LZ06]

Octrees können für verschiedene Zwecke angewendet werden. In dem Paper von Meagher [Mea82] werden Octrees für geometrische Modellierung verwendet. Aber auch zur schnellen Suche nach

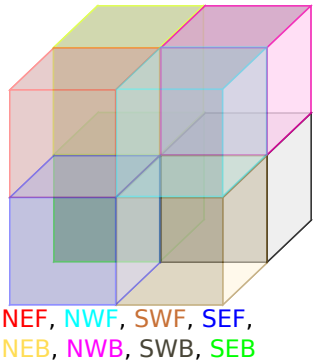


Abbildung 2.4 Diese Abbildung zeigt die acht Oktanten eines Octree-Knotens. NEF steht dabei für North East Front, SWB für South West Back usw.

Objekten in einem bestimmten Volumen lassen sich Octrees aufgrund ihrer Struktur gut verwenden. Als Beispiel lässt sich hier die Suche nach Nachbarpunkten in einer Punktwolke nennen, wie es von Behley et al. [BSC15] gemacht wurde. In dieser Arbeit soll auch ein Octree zur Suche von Nachbarn in Punktwolken eingesetzt werden. Eine rekursive Definition für Quadtrees wurde von Langetepe und Zachmann [LZ06] übernommen und für Octree angepasst. Ein Octree für eine Menge an Punkten P in einem Würfel $W = [x1_w : x2_w] \times [y1_w : y2_w] \times [z1_w : z2_w]$ lässt sich ein Octree folgendermaßen definieren:

- Falls $|P| \leq 1$, dann ist der Octree ein einzelnes Blatt, in dem W und P gespeichert werden.
- Wenn mehr als 1 Punkt vorhanden ist, sind W_{NEF}, W_{NWF}, W_{SWF}, W_{SEF}, W_{NEB}, W_{NWB}, W_{SWB}, W_{SEB} die acht Oktanten (siehe Abbildung 2.4).
  Außerdem gilt $x_{mid} = x1_w + x2_w/2$ und $y_{mid} = y1_w + y2_w/2$ und $z_{mid} = z1_w + z2_w/2$.

$$
\begin{aligned}
P_{NEF} &:= \{p \in P : p_x > x_{mid} \wedge p_y > y_{mid} \wedge p_z > z_{mid}\} \\
P_{NWF} &:= \{p \in P : p_x \leq x_{mid} \wedge p_y > y_{mid} \wedge p_z > z_{mid}\} \\
P_{SWF} &:= \{p \in P : p_x \leq x_{mid} \wedge p_y > y_{mid} \wedge p_z \leq z_{mid}\} \\
P_{SEF} &:= \{p \in P : p_x > x_{mid} \wedge p_y > y_{mid} \wedge p_z \leq z_{mid}\} \\
P_{NEB} &:= \{p \in P : p_x > x_{mid} \wedge p_y \leq y_{mid} \wedge p_z > z_{mid}\} \\
P_{NWB} &:= \{p \in P : p_x \leq x_{mid} \wedge p_y \leq y_{mid} \wedge p_z > z_{mid}\} \\
P_{SWB} &:= \{p \in P : p_x \leq x_{mid} \wedge p_y \leq y_{mid} \wedge p_z \leq z_{mid}\} \\
P_{SEB} &:= \{p \in P : p_x > x_{mid} \wedge p_y \leq y_{mid} \wedge p_z \leq z_{mid}\}
\end{aligned}
$$

Der Octree besteht aus einem Wurzelknoten v mit ach Kindern und der Würfel W ist in v gespeichert. Das X-te Kind ist der Wurzelknoten des Octree der Punktmenge P_X und X ist ein Element aus der Menge $\{NEF, NWF, SWF, SEF, NEB, NWB, SWB, SEB\}$. [LZ06]

2.3.2 K-d-Tree

Ein k-dimensionaler Baum (K-d-Tree) ist eine Generalisierung des eindimensionalen Suchbaums. Sei D eine Menge von n Punkten in $\mathbb{R}^k$. Der Einfachheit halber gilt $k = 2$ und es wird angenommen, dass alle $X-$ und $Y-$Koordinaten unterschiedlich sind. Zuerst wird nach einer Variable s gesucht, mit deren Hilfe die Punkte aus D an der Trennlinie $X = s$ in zwei Teilmengen geteilt werden. [LZ06]

$$D_{<s} = \{(x, y) \in D; x < s\} = D \cap \{X < s\}$$
$$D_{>s} = \{(x, y) \in D; x < s\} = D \cap \{X > s\}$$

Dies wird für die beiden Mengen mit der $Y-$Koordinate und den Trennlinie $Y = t_1$ und $Y = t_2$ fortgesetzt. Die Schritte werden dann rekursiv auf die entstandenen Teilmengen angewendet. Auf diese Weise erhält man einen Binärbaum, nämlich den 2-d-Tree der Punktmenge D, wie in Abbildung 2.5 gezeigt. Jeder interne Knoten des Baums entspricht einer Trennlinie. Für jeden Knoten v des 2-d-Trees wird ein Rechteck $R(v)$ definiert. Für den Wurzelknoten r ist $R(r)$ die Ebene selber, in der die Punkte liegen. Für die Kinder eines Knoten v *rechts* und *links* sind die Rechtecke $R(rechts)$ und $R(links)$ die Rechtecke, die entstehen, wenn man $R(v)$ an der Trennlinie teilt. Die Menge an Rechtecken $\{R(l) : \ l$ ist ein Blatt$\}$ partitioniert die Ebene in Rechtecke. Jedes $R(l)$ beinhaltet genau einen Punkt aus D. [LZ06]

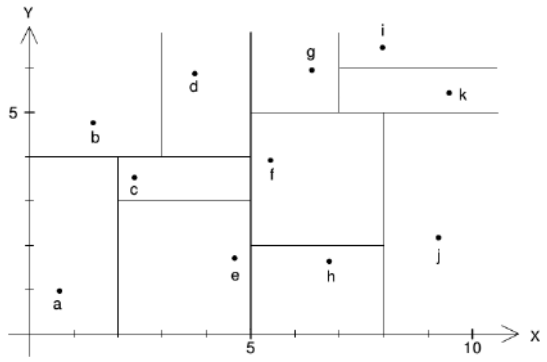


Abbildung 2.5 Die Trennlinien des K-d Trees teilen den Raum zwischen den Punkten $a - k$. (In Anlehnung an [LZ06])

2.3.3 Vantage-Point Tree

Bei einem Vantage-Point Tree (Vp-Tree) besitzt jeder Knoten des Baums einen sogenannten Vantage-Point [Kib07]. Die Punkte der Punktmenge P werden anhand ihrer Distanz zu diesem Vantage-Point in die Mengen P_1 und P_2 partitioniert. Sei dazu vp der gewählte Vantage-Point des Wurzelknotens v, m der Median aller Distanzen der Punkte aus P zu vp und $dist(p, vp)$ die Distanz vom Punkt p zum Vantage-Point vp, dann gilt:

$$P_1 = \{p \in P \mid d(p, vp) < m\}$$
$$P_2 = \{p \in P \mid d(p, vp) \geq m\}$$

Die Partitionierung wird rekursiv für P_1 und P_2 fortgesetzt. Jede Teilmenge wie P_1 und P_2 gehört zu einem Knoten des Baums (Abbildung 2.6 veranschaulicht, wie der Raum bei einem Vp-Tree unterteilt wird).[Fu+]

Bei der Wahl des Vantage-Points kann wie bei Steve Hanov [Han12] zufällig aus der Menge P gewählt werden. In diesem Fall ist die Konstruktionszeit des Vp-Tree $O(n \cdot log(n))$ [Yia70].

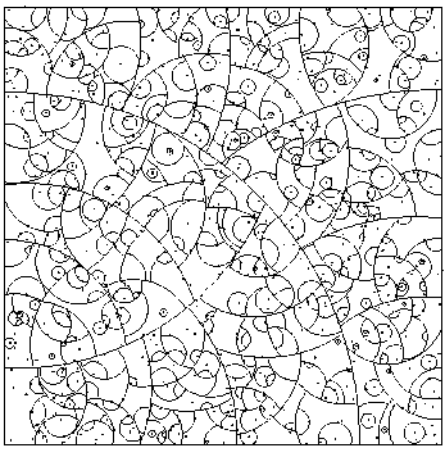


Abbildung 2.6 Dekomposition eines Vp-Tree. Jeder Kreis entspricht dabei einem Knoten im Vp-Tree. ([Yia70])

2.4 CUDA

CUDA ist eine NVIDIA Architektur [NVIc], die für allgemeine Berechnungen auf graphical processing units (GPUs) entwickelt wurde. In GPU-beschleunigten Anwendungen wird der sequen-

tielle Teil der Arbeit auf der CPU ausgeführt, während aufwendige Berechnungen auf tausenden GPU-Kernen (3584 bei einer GTX 1080 TI [NVIb]) parallel laufen. CUDA kann mit Programmiersprachen wie C, C++, Fortran, Python und MATLAB durch Erweiterungen der Sprachen mit einigen Schlüsselwörtern verwendet werden [NVIa]. In dieser Arbeit wird CUDA angewendet, um die Schnittpunktberechnung des Strahls mit der Punktwolke zu parallelisieren.

Eine weitere Schnittstelle für GPU-Programmierung ist die Open Computing Language (OpenCL). OpenCL ist ein offener Standard für parallele Berechnungen auf CPUs, GPUs, Digital Signal Processors (DSPs) und anderen Prozessoren, während CUDA nur auf NVIDIA GPUs läuft. Die API von OpenCL unterscheidet sich dabei von der von CUDA, bietet aber ähnliche Funktionalität. [KDH10]

Karimi et al. [KDH10] haben die Performance von CUDA und OpenCL anhand einer berechnungsintensiven wissenschaftlichen Anwendung verglichen. Das Ergebnis dieses Vergleichs war, dass CUDA sowohl bei der Datenübertragung zur GPU als auch bei der Kernelausführung konsistent besser ist. Aus diesem Grund wird in dieser Arbeit CUDA verwendet.

2.5 Echtzeit

Laut Heinz Wörn [WB05] unterscheidet man je nach Strenge der einzuhaltenden Zeitbedingung zwischen harten, festen und weichen Echtzeitbedingungen. Bei harten Echtzeitbedingungen müssen die Zeitbedingungen auf jeden Fall eingehalten werden, da sonst Schaden droht. Bei festen Echtzeitbedingungen ist das Ergebnis nach Ablauf der Zeit wertlos und kann verworfen werden, es droht aber kein unmittelbarer Schaden wie bei harten Echtzeitbedingungen. Bei weichen Echtzeitbedingungen ist es erlaubt die Zeitbedingungen in einem gewissen Rahmen zu überschreiten. Als Beispiel nennt Heinz Wörn [WB05] hier Multimediasysteme, bei denen es nicht schlimm ist, wenn ein ein einzelnes Bild zu spät übertragen wird, da dies nur zu einem leichten „Ruckeln“ führt. [WB05]

In dieser Arbeit geht es um weiche Echtzeitbedingungen. Das heißt, dass nicht garantiert wird, dass der Schnittpunkt in einer bestimmten Zeit berechnet wird. Die Bedingung für Echtzeit für die Schnittpunktberechnung in dieser Arbeit ist, dass durchschnittlich mindestens 30 Schnittpunkte pro Sekunde berechnet werden können. Diese Angabe stammt daher, dass dies der Frequenz entspricht, mit der eine Microsoft Kinect neue Punktwolken liefert und in dieser Arbeit als Quelle dynamischer Punktwolken verwendet wird.

2.6 Least-Squares

Ein Least-Squares Algorithmus wird in dieser Arbeit verwendet, um die Oberflächennormalen für einen Punkt p mithilfe seiner Nachbarn zu bestimmen. Eine mögliche Lösung für diese Problem bietet Schneider [SE02]. Dieser bestimmt zu den Nachbarn $\{x_i, y_i, z_i\}_{i=1}^m$ die Werte a, b, und c so, dass die Ebene $z = ax + by + c$ in dem Sinne am besten passt, dass die Summe der quadrierten Fehler zwischen z_i und $ax_i + by_i + c$ minimiert wird. Dabei wird nur der Fehler in z-Richtung betrachtet.

Eine Funktion, die den Fehler in Abhängigkeit von a, b und c beschreibt, könnte so aussehen $E(a, b, c) = \sum_{i=1}^m [(ax_i + by_i + c) - z_i]^2$. Der Scheitelpunkt dieser Funktion enthält die Werte für a, b und c, die für die Ebene $z = ax + by + c$ den kleinsten Fehler ergeben. Daraus kann man nun folgende Gleichungssystem schließen, mit denen man die Werte a, b und c am Scheitelpunkt bestimmen kann.[SE02]

$$\begin{bmatrix} \sum_{i=1}^m x_i^2 & \sum_{i=1}^m x_i y_i & \sum_{i=1}^m x_i \\ \sum_{i=1}^m x_i y_i & \sum_{i=1}^m y_i^2 & \sum_{i=1}^m y_i \\ \sum_{i=1}^m x_i & \sum_{i=1}^m y_i & \sum_{i=1}^m 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^m x_i z_i \\ \sum_{i=1}^m y_i z_i \\ \sum_{i=1}^m z_i \end{bmatrix}$$

Der Vektor $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ entspricht dann der Oberflächennormalen am Punkt p.

2.7 Distanz zu Strahl berechnen

Um die Punkte nach ihrer Distanz zum Strahl zu filtern, muss diese berechnet werden. Dazu wird der Ansatz von Schneider [SE02] verwendet, der den Punkt Q auf die Linie $L = P + t \cdot \vec{d}$ projiziert und dann die Distanz vom projizierten Punkt Q' zu Q berechnet. Um den Wert t_0 zu bestimmen, für den $Q' = P + t_0 \cdot \vec{d}$ gilt, wird die folgende Gleichung verwendet:

$$t_0 = \frac{\vec{d} \cdot (Q - P)}{\vec{d} \cdot \vec{d}}$$

Für die Distanz a vom Punkt Q zur Linie L ergibt sich somit die Gleichung:

$$a = |Q - (P + \frac{\vec{d} \cdot (Q - P)}{\vec{d} \cdot \vec{d}} \cdot \vec{d})|$$

Wenn man nun die Distanz zum Strahl mit dem Startpunkt P und der Richtung $\vec{d}$ bestimmen will, muss man t_0 auf positive Werte limitieren. In dem Fall gilt für die Distanz a:

$$a = \begin{cases} |Q - P| & t_0 \leq 0 \\ |Q - (P + t_0 \cdot \vec{d}) & t_0| > 0 \end{cases}$$

2.8 Schnittpunkt zwischen Strahl und Scheibe berechnen

Um den Schnittpunkt eines Strahls g und der Scheibe am Punkt P zu berechnen, wird die Oberflächennormale $\vec{n}$ am Punkt P verwendet. Zuerst wird die Normalenform [Erb+11] der Ebene E folgendermaßen gebildet:

$$E : 0 = (\vec{v} - \vec{v_0}) \cdot \vec{n}$$

v_0 ist der Ortsvektor von P und v gehört zu einem beliebigen Punkt auf der Ebene E. Für den Strahl g gilt:

$$g = \vec{s} + t \cdot \vec{d}$$

Den Schnittpunkt zwischen dem Strahl und der Ebene lässt sich berechnen, indem man t_0 bestimmt, so dass der Punkt $Q = \vec{s} + t_0 \cdot \vec{d}$ die Gleichung $0 = (\vec{Q} - \vec{v_0}) \cdot \vec{n}$ erfüllt.

Zum Abschluss muss geprüft werden, ob der Schnittpunkt innerhalb der Scheibe liegt. Dazu wird geprüft, ob der Abstand vom Schnittpunkt zum Punkt kleiner gleich ist, als der Scheibenradius. Ist dies der Fall wurde die Scheibe getroffen, ansonsten wurde sie verfehlt.

Da nun die Grundlagen geklärt wurden, wird im nächsten Kapitel der im Rahmen dieser Arbeit entworfene Algorithmus zur Berechnung des Schnittpunktes zwischen einem Strahl und einer Punktwolke beschrieben.

Kapitel 3

Design des Algorithmus

Im vorherigen Kapitel wurden die Grundlagen der in dieser Arbeit verwendeten Datenstrukturen und Algorithmen geschildert. In diesem Kapitel wird der in dieser Arbeit entwickelten Ansatzes zur Berechnung des Schnittpunktes eines Strahls mit einer Punktwolke vorgestellt. Die Implementierungsdetails werden in Kapitel 4 erläutert.

3.1 Der Algorithmus

Die Schwierigkeit bei Schnittpunktberechnung mit Punktwolken besteht darin, dass die Oberfläche nur durch Punkte repräsentiert wird und es so nicht möglich ist, den Schnittpunkt direkt zu berechnen. Die im Abschnitt 2.2.1 vorgestellten Ansätze für Ray Tracing auf Punktwolken bieten eine Lösung für dieses Problem, indem Scheiben aus der Punktwolke generiert werden, die die Oberfläche repräsentieren. Der vorgestellte Ansatz von Linsen et al. [Lin07] eignet sich in dem Szenario dieser Arbeit allerdings nicht, weil diese zuerst alle Scheiben berechnen und zu einem Octree hinzufügen. Das wäre für diese Arbeit allerdings viel zu aufwendig, da dynamische Punktwolken betrachtet werden und folglich für jede neue Punktwolke alle Scheiben berechnet werden müssten.

Schaufler und Jensen [SJ00] wiederum gehen davon aus, dass Oberflächennormalen zur Verfügung stehen. Mit Hilfe der Oberflächennormalen werden Scheiben mit dem Radius $r_{scheibe}$ um die Punkte gelegt. Die Normalen der Scheiben entsprechen dabei den Oberflächennormalen an den Punkten und $r_{scheibe}$ wird so gewählt, dass keine Löcher in der Oberfläche der Punktwolke entstehen. Dann prüfen sie, ob der Strahl eine der Scheiben schneidet. Da in dieser Arbeit davon ausgegangen wird, dass keine Oberflächennormalen zur Verfügung stehen, lässt sich auch der Ansatz von Schaufler und Jensen nicht direkt verwenden.

Allerdings lässt sich der Ansatz so anpassen, dass die Oberflächennormalen nach Bedarf berechnet

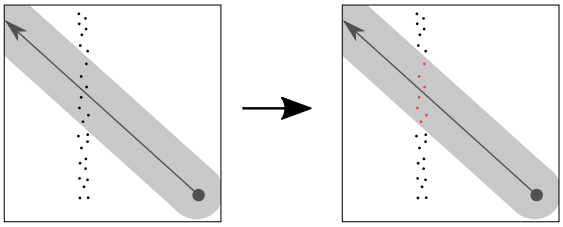


Abbildung 3.1 Die Punkte, die für die Schnittpunktberechnung infrage kommen(rot), besitzen einen Minimalabstand kleiner gleich $r_{scheibe}$ zum Strahl R (grauer Bereich).

werden können. Dazu wurde vor der Schnittpunktberechnung ein zusätzlicher Schritt eingeführt, in dem die Punkte bestimmt werden, die für die Schnittpunktberechnung infrage kommen. Es müssen nur für diese Punkte die Oberflächennormalen berechnet werden. Ob ein Punkt für die Schnittpunktberechnung infrage kommt, wird anhand der Distanz des Punktes zu dem Strahl festgestellt. Wenn die Distanz des Punktes zum Strahl größer ist als $r_{scheibe}$, kann der Strahl die Scheibe nicht schneiden. Somit spielen nur die Punkte für die Schnittpunktberechnung eine Rolle, deren Abstand zum Strahl kleiner als $r_{scheibe}$ ist.

So ergeben sich zum Bestimmen des Schnittpunktes eines Strahls mit einer Punktwolke drei Schritte. Im ersten Schritt werden die Punkte bestimmt, die für die Schnittpunktberechnung infrage kommen. Im zweiten Schritt werden für diese Punkte die Oberflächennormalen berechnet und im letzten Schritt wird der Schnittpunkt mit der Punktwolke bestimmt, indem geprüft wird, wo der Strahl die Scheiben schneidet, die sich durch die berechneten Oberflächennormalen ergeben. Im Folgenden werden die Schritte genauer beschrieben.

Schritt 1: Filtern der Punkte
Um Rechenzeit zu sparen, werden in diesem Schritt alle Punkte aussortiert, die bei der Schnittpunktberechnung keine Rolle spielen. Wie in Abbildung 3.1 zu sehen ist, wird dazu die Menge D der Punkte in der Punktwolke bestimmt, deren kürzeste Distanz $dist(R, p)$ zum Strahl R kleiner gleich $r_{scheibe}$ ist. Es wird die Menge $D_{filtered}$ bestimmt, für die gilt:

$$D_{filtered} = \{p \mid p \in D \wedge dist(R, p) \leq r_{scheibe}\}$$

Je nach Position von Strahl und Punktwolke wird so die Menge der zu untersuchenden Punkte stark reduziert. Wie groß die Reduktion ist wird in Kapitel 5.1 betrachtet.

Schritt 2: Berechnen der Oberflächennormalen
In diesem Schritt werden die Normalen der Punkte aus $D_{filtered}$ berechnet. Abbildung 3.2 zeigt, wie dazu für jeden Punkt $p \in D_{filtered}$ die Nachbarn gesucht werden, deren Entfernung zu p kleiner ist als $r_{scheibe}$. Diese dienen als Eingabe für einen Least-Squares Algorithmus, der eine Ebene zwischen die Nachbarn legt. Die Normale n der Ebene wird als Oberflächennormale am Punkt p verwendet.

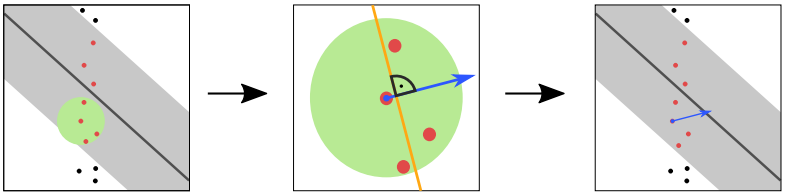


Abbildung 3.2 Um die Oberflächennormale (blau) eines Punktes zu bestimmen, werden zuerst die Punkte in einem bestimmten Radius (grün) um diesen Punkt gesucht. Zwischen diese Punkte wird im mittleren Bild eine Ebene (orange) mit einem Least-Square-Algorithmus gelegt. Das rechte Bild zeigt die daraus resultierende Oberflächennormale zu der ermittelten Ebene. Ob die Oberflächennormale nach vorne oder hinten zeigt spielt keine Rolle, da sie später nur zum Platzieren der Scheiben benutzt wird.

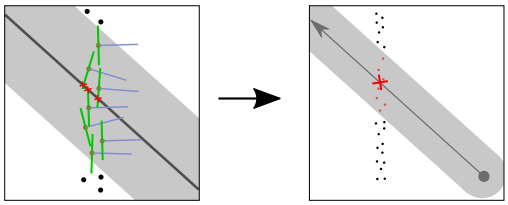


Abbildung 3.3 Das linke Bild zeigt die im vorherigen Schritt bestimmten Oberflächennormalen (blau) und die Scheiben (grün) an jedem Punkt, die sich aus den Normalen ergeben. Die Schnittpunkte des Strahls mit den Scheiben sind mit roten Kreuzen markiert. Aus diesen Schnittpunkten wird der Mittelwert gebildet wie er im rechten Bild dargestellt ist

Schritt 3: Bestimmung des Schnittpunktes

Abbildung 3.3 zeigt, wie die im vorherigen Schritt berechneten Normalen verwendet werden, um einen Schnittpunkt zwischen der Punktwolke und dem Strahl zu bestimmen. Dazu wird ausgehend vom Startpunkt des Strahls geprüft, ob der Strahl eine Scheibe der Punkte $p \in D_{filtered}$ schneidet. Die Normale der Scheibe am Punkt p entspricht der Oberflächennormale, die im vorherigen Schritt für p berechnete wurde. Sobald ein Schnittpunkt festgestellt wurde, werden ausgehend von diesem alle weiteren Schnittpunkte bestimmt, die innerhalb einer bestimmten Entfernung nach dem ersten Schnittpunkt folgen. Aus den gesammelten Schnittpunkten wird dann der Mittelwert gebildet.

3.2 Wahl der Beschleunigungsdatenstruktur

Im vorherigen Abschnitt wurde das Design des Algorithmus vorgestellt. Dabei müssen Punkte nach ihrem Abstand zu dem Strahl gefiltert werden und die Nachbarn von verschiedenen Punkten gefunden werden. Um dies zu beschleunigen, ist die Wahl der richtigen Beschleuni-

gungsdatenstruktur für die CPU-Implementierung wichtig. Dabei ist zu beachten, dass sich die Beschleunigungsdatenstrukturen wegen der dynamischen Punktwolken in dieser Arbeit schnell erstellen lassen. Kumar et al.[KZN08] haben bereits mehrere solcher Beschleunigungsdatenstrukturen anhand ihrer Erstellungs- und Suchzeiten miteinander verglichen und sind zu dem Ergebnis gekommen, dass sich Vp-Trees und Kd-Trees schnell erstellen lassen. Die Ergebnisse dazu sind in Tabelle 2.1 zusammengefasst.

In diesem Abschnitt werden Implementierung dieser Beschleunigungsdatenstrukturen anhand ihrer Konstruktionszeit bei verschiedenen Punktwolken und der benötigten Zeit zur Radius-Nachbar-Suche verglichen. Als Vp-Tree wurde eine Implementierung von Steve Hanov [Han12] verwendet. Da diese keine Funktion zum Suchen von Punkten in einem bestimmten Radius bietet, wurde diese zusätzlich implementiert. Bei der K-d-Tree Implementierung handelt es sich um nanoflann [BR14]. Der Octree wurde selber implementiert und wird im Abschnitt 4.3 genauer beschrieben. Die Tests für die Beschleunigungsdatenstrukturen wurden auf Windows 10 Version 1803(OS Build 17134.81) und einem i5 4670k@4Ghz ausgeführt. Der verwendete Compiler ist Microsoft (R) C/C++ Version 1900242151 mit den Optimierungsoptionen /O2.

3.2.1 Testfälle

Zum Testen wurden die Beschleunigungsdatenstrukturen mit Punktwolken unterschiedlicher Größe erstellt. Die Punktwolken wurden künstlich generiert, indem die Anzahl der gewünschten Punkte zufällig auf der Oberfläche einer Kugel verteilt. Die Testpunktwolke ist also um eine Sphäre. Des Weiteren lässt sich bei dem Octree und bei dem K-d-Tree die bucket size konfigurieren. Diese legt fest, wie viele Punkte ein Blatt des Baumes maximal enthalten darf. Beim Vp-Tree lässt sich dies nicht einstellen und ist immer eins. In einem weiteren Test wurde die Zeit gemessen, die für die Radius-Nachbar-Suche benötigt wird. Dazu wurden für 100 zufällige Punkte der Punktwolke die Nachbarn gesucht. Bei jeder Beschleunigungsdatenstruktur wurden die Gleichen 100 zufälligen Punkte gesucht.

3.2.2 Ergebnisse der Beschleunigungsdatenstrukturen

In der Tabelle 3.1 sind die Erstellungszeiten der Beschleunigungsdatenstrukturen gelistet. Je größer die bucket size der Bäume, desto schneller lassen sie sich für eine bestimmte Punktwolkengröße erstellen. Der Octree hat die besten Erstellungszeiten und ist durchschnittlich ca 2-3 mal so schnell wie der K-d-Tree. Der Unterschied wird bei größeren Punktwolken noch deutlicher. Des Weiteren ist Vp-Tree in jedem Test am langsamsten. Die Tabelle 3.2 zeigt die benötigte Zeit zur Radius-Nachbar-Suche. Bei der Nachbarsuche ist die Performance der drei Bäume sehr

Größe der Punktwolke	Zeit zum Erstellen								
	bucket size = 1			bucket size = 10		bucket size = 100		bucket size = 1000	
	Vp-Tree	Octree	K-d-Tree	Octree	K-d-Tree	Octree	K-d-Tree	Octree	K-d-Tree
1000	0.0061176 s	0.00152627 s	0.00278385 s	0.00084971 s	0.00179189	0.000337881 s	0.000968023 s	0.000237757 s	0.000574777 s
20000	0.0128617 s	0.00304445 s	0.00569425 s	0.00164469 s	0.00359086	0.000698236 s	0.00216345 s	0.000467736 s	0.00133423 s
40000	0.0275293 s	0.00627415 s	0.0121657 s	0.00342381 s	0.00792398	0.00182825 s	0.00499289 s	0.000905537 s	0.00303687 s
80000	0.0581541 s	0.013376 s	0.0121657 s	0.00727409 s	0.0174103	0.00391991 s	0.0114494 s	0.00237803 s	0.0075161 s
160000	0.122893 s	0.0276136 s	0.0580189 s	0.0159198 s	0.0374585	0.00851874 s	0.0255472 s	0.00618614 s	0.0177668 s
320000	0.257845 s	0.056454 s	0.124104 s	0.0344778 s	0.0840053	0.0182702 s	0.0584841 s	0.0140277 s	0.0420987 s
640000	0.536523 s	0.11483 s	0.307839 s	0.0676026 s	0.222034	0.0390827 s	0.166008 s	0.0302613 s	0.132003 s
1280000	1.12906 s	0.234109 s	0.687243 s	0.143548 s	0.574219	0.0831362 s	0.443662 s	0.0658898 s	0.378824 s
2560000	2.3608 s	0.478482 s	1.5518 s	0.304522 s	1.30052	0.175799 s	1.07544 s	0.140673 s	0.962143 s
5210000	4.9496 s	0.975386 s	3.64075 s	0.644765 s	3.05735	0.370646 s	2.64333 s	0.298286 s	2.35199 s

Tabelle 3.1 Die gemessenen Erstellungszeiten der verschiedenen Beschleunigungsdatenstrukturen bei Punktwolken unterschiedlicher Größe.

Größe der Punktwolke	Zeit suche								
	bucket size = 1			bucket size = 10		bucket size = 100		bucket size = 1000	
	Vp-Tree	Octree	K-d-Tree	Octree	K-d-Tree	Octree	K-d-Tree	Octree	K-d-Tree
1000	0.016563 ms	0.0103846 ms	0.00810262 ms	0.00673765 ms	0.00562523 ms	0.00589582 ms	0.00640091 ms	0.00857465 ms	0.0105289 ms
20000	0.0168095 ms	0.0114549 ms	0.00843635 ms	0.00684588 ms	0.00576353 ms	0.00634981 ms	0.00646405 ms	0.00952471 ms	0.0115752 ms
40000	0.0199063 ms	0.013313 ms	0.00973518 ms	0.0074532 ms	0.00665948 ms	0.00669557 ms	0.0072698 ms	0.0142871 ms	0.0130604 ms
80000	0.0189261 ms	0.0150838 ms	0.0114339 ms	0.0122396 ms	0.0067647 ms	0.00708942 ms	0.00739909 ms	0.010556 ms	0.0135084 ms
160000	0.0209766 ms	0.0169328 ms	0.0118277 ms	0.00978026 ms	0.00825895 ms	0.0100027 ms	0.00744418 ms	0.0145577 ms	0.0127267 ms
320000	0.022086 ms	0.0169689 ms	0.0155077 ms	0.00951268 ms	0.00891739 ms	0.0079102 ms	0.00899254 ms	0.0135354 ms	0.0158835 ms
640000	0.023463 ms	0.0179761 ms	0.0165841 ms	0.0111603 ms	0.0110099e ms	0.00795831 ms	0.0126515 ms	0.0135655 ms	0.0319444 ms
1280000	0.0243499 ms	0.0190254 ms	0.0182407 ms	0.0120652 ms	0.0126154 ms	0.00907374 ms	0.0146268 ms	0.0136467 ms	0.0370074 ms
2560000	0.0235291 ms	0.0204054 ms	0.0183098 ms	0.0129973 ms	0.0129401 ms	0.00949165 ms	0.0156761 ms	0.0148253 ms	0.0390098 ms
5210000	0.0236885 ms	0.020673 ms	0.0176334 ms	0.0130664 ms	0.012871 ms	0.00966903 ms	0.0154386 ms	0.015195 ms	0.0397584 ms

Tabelle 3.2 Die gemessenen Ausführungszeiten der verschiedenen Beschleunigungsdatenstrukturen zur Radius-Nachbar-Suche in Abhängigkeit zur Punktwolkengröße.

ähnlich. Bei kleinen bucket sizes ist der K-d-Tree am schnellsten, während bei großen bucket sizes der Octree etwas schneller ist. Der Vp-Tree ist auch hier am langsamsten.

3.2.3 Auswertung der Beschleunigungsdatenstrukturen

In dieser Arbeit spielt vor allem die Erstellungszeit eine Rolle, da die Beschleunigungsdatenstrukturen aufgrund der dynamischen Punktwolke oft neu erstellt werden müssen. In diesem Aspekt liefert der Octree die beste Performance. Vor allem größere bucket sizes sind im Kontext dieser Arbeit interessant, weil diese die Erstellung deutlich beschleunigen. Bei der Suche sind der K-d Tree und der Octree ca. gleich schnell. Der Vp-Tree ist sowohl bei der Erstellungszeit, als auch bei der Radius-Nachbar-Suche die schlechteste Wahl. Zusammenfassend lässt sich sagen, dass von den hier geprüften Implementierungen der Octree aufgrund der guten Erstellungszeit die beste Wahl für dieser Arbeit ist.

Im folgenden Kapitel wird nun die Implementierung des hier vorgestellten Algorithmus erläutert.

Kapitel 4

Implementierung

Im vorherigen Kapitel wurde der in dieser Arbeit entwickelte theoretische Ansatz zur Berechnung des Schnittpunktes eines Strahls mit einer Punktwolke vorgestellt. In diesem Kapitel wird auf die Implementierung dieses Ansatzes eingegangen. Es gibt sowohl eine CPU-Implementierung als auch und eine CUDA-Implementierung, die in Kapitel 5 evaluiert und verglichen werden.

4.1 CPU-Implementierung

In diesem Abschnitt wird die Implementierung des im vorherigen Kapitel vorgestellten Ansatzes auf der CPU erläutert. Der Ansatz besteht aus den drei Schritten Filtern der Punkte, Berechnen der Oberflächennormalen und Bestimmen des Schnittpunktes. Des Weiteren wurde ein Octree verwendet, um die Nachbarsuche von Punkten in der Punktwolke zu beschleunigen. Es wurde ein Octree gewählt, da die Untersuchung verschiedener Beschleunigungsdatenstrukturen im Kapitel 3.2.2 ergeben haben, dass diese aufgrund der schnellen Konstruktionszeit am besten für das Szenario dieser Arbeit geeignet ist.

4.1.1 Architektur der CPU-Implementierung

Die CPU-Implementierung besteht aus drei Klassen, die im Klassendiagramm 4.1 zu sehen sind. Zum einen gibt es die Klasse IntersectionScene, die als Schnittstelle für die Schnittpunktberechnung dient. Die Klasse bietet eine Funktion, um die interne Punktwolke zu aktualisieren und eine weitere, die den Schnittpunkt zwischen dem übergebenen Strahl und der internen Punktwolke berechnet. Des Weiteren beinhaltet die IntersectionScene-Klasse einen Octree, der dazu dient, die Suche nach bestimmten Punkten aus der Punktwolke bei der Schnittpunktberechnung zu beschleunigen. Der Octree wird aus einer Punktwolke erstellt und bietet Funktionen, um die

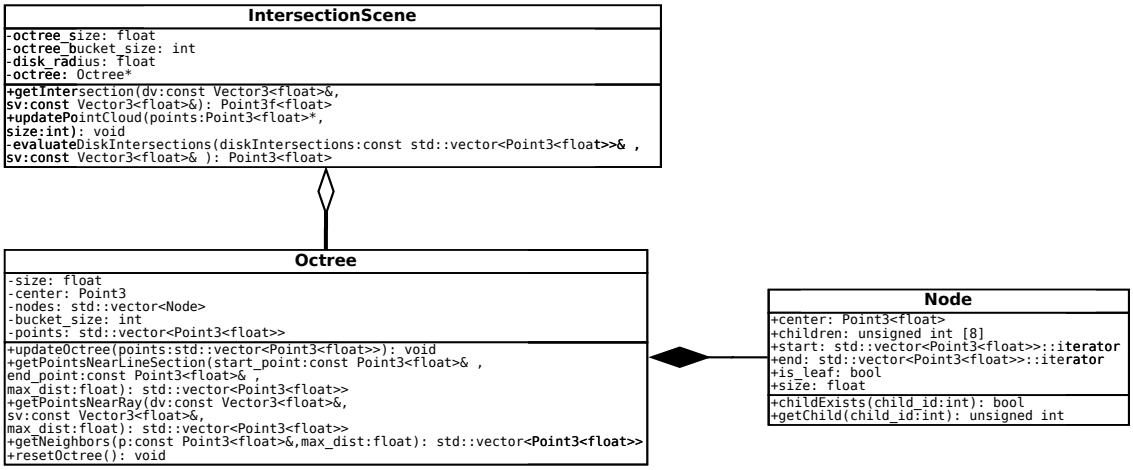


Abbildung 4.1 Klassendiagramm der CPU-Implementierung.

Nachbarn eines Punktes in einem bestimmten Radius zu bestimmen, und eine weitere Funktion, um die Punkte innerhalb eines bestimmten Abstands zu einem Strahl zu finden. Der Octree hat eine Liste von Knoten, die den Würfeln des Octrees entsprechen.

4.1.2 Programmablauf CPU-Implementierung

Dieser Abschnitt geht auf die einzelnen Schritte der Schnittpunktberechnung bei der CPU-Implementierung ein. Abbildung 4.2 zeigt ein Sequenzdiagramm, welches den Ablauf einer Schnittpunktberechnung zeigt. Im weiteren Verlauf werden nun die Schritte genauer beschrieben.

4.1.2.1 Filtern der Punkte

Dieser Schritt entspricht dem Funktionsaufruf *getPointsNearRay(Ray)* aus dem Sequenzdiagramm 4.2. Wie im Ansatz in Kapitel 3 beschrieben, wird in diesem Schritt die Menge der Punkte $D_{filtered}$ bestimmt. D ist hier die Menge der Punkte in der Punktwolke, R der Strahl, $dist(R, p)$ die Distanz vom Punkt p zum Strahl und $r_{scheibe}$ der Radius der Scheiben. Dann gilt für $D_{filtered}$:

$$D_{filtered} = \{p \mid p \in D \wedge dist(R, p) < r_{scheibe}\}$$

Der Octree besitzt eine Funktion, die genau die Menge $D_{filtered}$ zurückliefert. Diese wird im Abschnitt 4.3.2 erklärt. Die Funktion erhält als Eingabe den Strahl R und gibt die Menge $D_{filtered}$ als Ergebnis zurück.

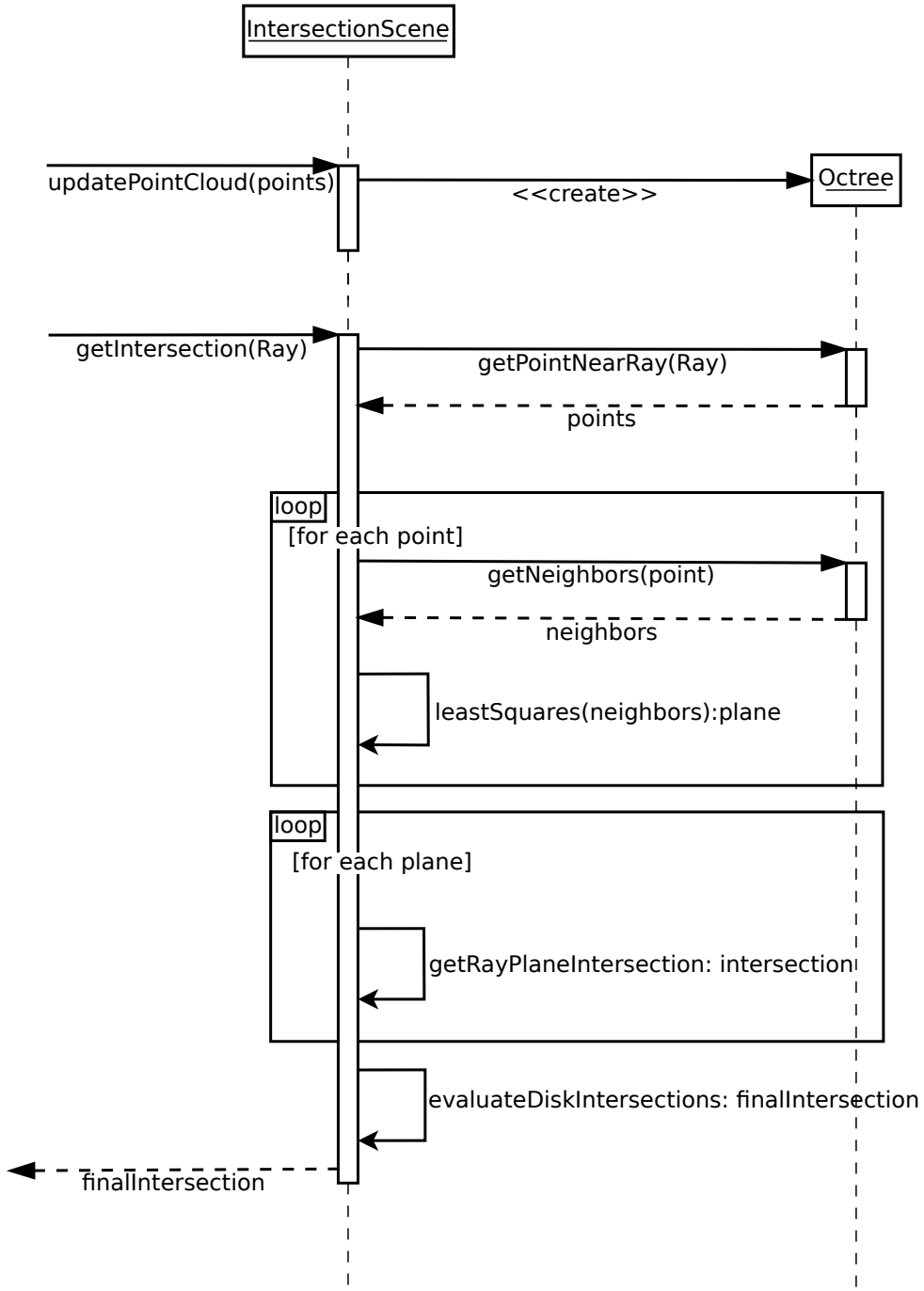


Abbildung 4.2 Sequenzdiagramm, welches den Ablauf einer Schnittpunktberechnung darstellt. Zuerst wird *updatePointCloud()* die Punktwolke aktualisiert und der Octree neu erstellt. Dann wird der Schnittpunkt berechnet, indem die Punkte neben dem Strahl gesucht werden und für diese die Scheiben zur Schnittpunktberechnung ermittelt werden. Abschließend wird der Schnittpunkt mit den Scheiben berechnet.

4.1.2.2 Berechnen der Oberflächennormalen

Nachdem im vorherigen Schritt die Menge $D_{filtered}$ bestimmt wurde, werden nun die Oberflächennormalen der Punkte aus $D_{filtered}$ berechnet. Wie im Ansatz beschrieben, müssen dafür die Nachbarn der Punkte aus $D_{filtered}$ ermittelt werden. Der Octree bietet eine Funktion alle Punkte in einem Radius r_{suche} um einen Suchpunkt q zu finden (siehe Abschnitt 4.3.2). Für jeden Punkt $p \in D_{filtered}$ werden die Nachbarn mit dieser Funktion bestimmt, indem man den Punkt p als Suchpunkt und den Radius $r_{scheibe}$ der Scheiben als Suchradius verwendet.

Die Nachbarn werden als Eingabe für den Least-Squares Algorithmus aus dem Abschnitt 2.6 benutzt, um die Oberflächennormale am Punkt p zu berechnen. So erhält man für jeden Punkt aus $D_{filtered}$ die Oberflächennormale. Dieser Schritt wird im Sequenzdiagramm 4.2 durch die erste Schleife dargestellt.

4.1.2.3 Bestimmen des Schnittpunktes

Das Bestimmen des Schnittpunktes entspricht dem Teil ab der zweiten Schleife im Sequenzdiagramm 4.2. Dafür muss noch geprüft werden, ob der Strahl R eine der Scheiben schneidet. Im Sequenzdiagramm wird dies durch den Teil ab der zweiten Schleife dargestellt. Wenn $\vec{n}$ die Oberflächennormale am Punkt $p \in D_{filtered}$ ist, kann geprüft werden, ob die Scheibe von R geschnitten wurde, indem der Schnittpunkt von R mit der durch $\vec{n}$ und dem Punkt p aufgespannten Ebene berechnet wird. Ist der Abstand des Schnittpunkts zu p kleiner gleich $r_{scheibe}$, wird die Scheibe geschnitten. In Abschnitt 2.8 wird das genaue Vorgehen beschrieben, mit dem geprüft wird, ob eine Scheibe von einem Strahl geschnitten wurde. Ausgehend vom Startpunkt des Strahls R wird dies für jeden Punkt gemacht. Sobald eine Scheibe geschnitten wurde, werden alle weiteren Schnittpunkte in einem bestimmten Abstand hinter dem ersten Schnittpunkt gesammelt und der Mittelwert der gesammelten Schnittpunkte berechnet. Der Mittelwert ist der endgültige Schnittpunkt.

4.2 CUDA-Implementierung

In diesem Abschnitt wird die Umsetzung des Ansatzes zur Berechnung des Schnittpunktes eines Strahls mit einer Punktwolke in CUDA behandelt. Da es sich um eine dynamische Punktwolke handelt und das Erstellen von Beschleunigungsdatenstrukturen wie Octrees oder K-d-Trees für jede neue Punktwolke sehr aufwendig ist, wird in der CUDA-Implementierung auf solche verzichtet. Der Gedanke dabei ist, dass mit Hilfe der großen Anzahl an CUDA-Kernen das Filtern und die Nachbarsuche für jeden Punkt parallel ausgeführt werden kann. Wie im Abschnitt

3 beschrieben, werden alle Punkte, die nicht für die Schnittpunktberechnung infrage kommen, im ersten Schritt aussortiert. Bei der CUDA-Implementierung ist die Menge der verbleibenden Punkte folgendermaßen definiert:

$$D_{filtered} = \{p \mid p \in D \wedge dist(R, p) < r_{scheibe} + r_{nn}\}$$

D ist die Menge der Punkte in der Punktwolke, R der Strahl, $r_{scheibe}$ der Radius der Scheiben und r_{nn} der Radius, in dem die Nachbarn gesucht werden. $dist(R, p)$ ist die Distanz vom Strahl R zum Punkt p . Da es keine Suchstrukturen gibt, werden die Nachbarn der Punkte aus $D_{filtered}$ gesucht. Dazu wird die Distanz zu jedem anderen Punkt aus $D_{filtered}$ berechnet. Die Laufzeit eines sequenziellen Algorithmus ist in diesem Fall $O(n^2)$. Mit CUDA ist es allerdings möglich die Laufzeit auf $O(n)$ zu begrenzen, wenn $|D_{filtered}|$ kleiner gleich ist, als die Zahl der verfügbaren CUDA-Kerne.

4.2.1 Architektur der CUDA-Implementierung

Die CUDA-Implementierung besteht aus einer Klasse, die die Punktwolke und die Schnittpunktberechnungsanfragen verwaltet. Die Funktion *updatePointCloud()* aktualisiert die Punktwolke, indem die alte Punktwolke gelöscht und die neue auf die Grafikkarte kopiert wird. Mit *getIntersection()* ist es dann möglich den Schnittpunkt zwischen einem Strahl und der Punktwolke zu berechnen, indem die *IntersectionScene*-Klasse intern einen CUDA-Kernel aufruft, in dem die Schnittpunktberechnung stattfindet.

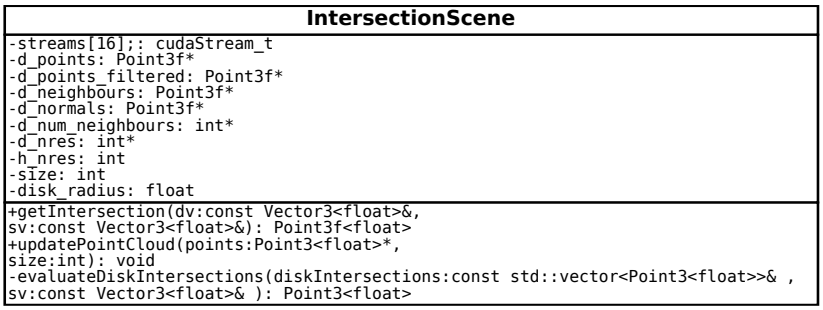


Abbildung 4.3 Klassendiagramm der CUDA-Implementierung.

4.2.2 Programmablauf der CUDA-Implementierung

In diesem Abschnitt wird auf die einzelnen Schritte der Schnittpunktberechnung bei der CUDA-Implementierung eingegangen. Abbildung 4.4 zeigt ein Sequenzdiagramm, welches den Ablauf der Schnittpunktberechnung zeigt. Im weiteren Verlauf werden nun die Schritte genauer beschrieben.

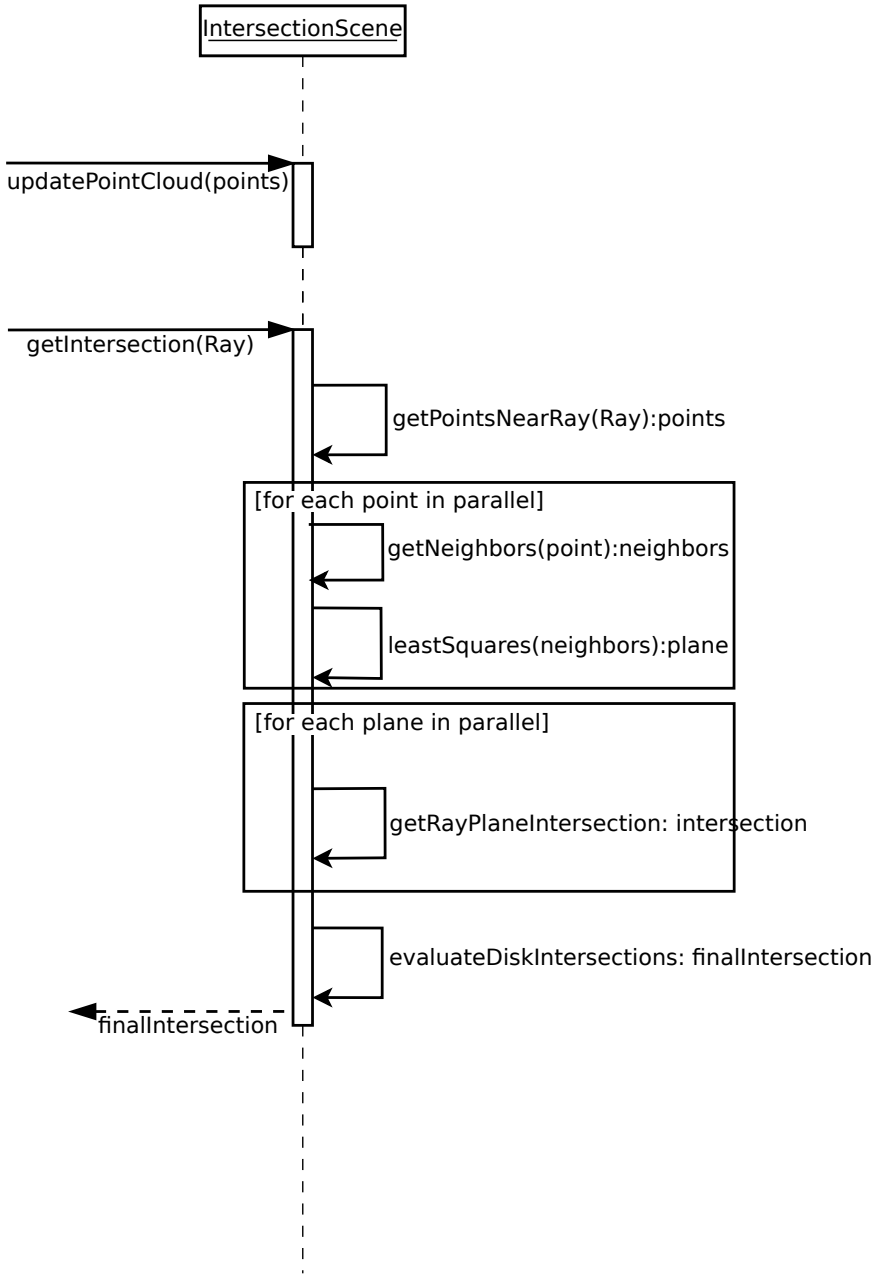


Abbildung 4.4 Sequenzdiagramm, welches den Ablauf einer Schnittpunktberechnung in CUDA darstellt. Erst werden die Punkte neben dem Strahl gesucht und für diese die Scheiben zur Schnittpunktberechnung ermittelt. Abschließend wird der Schnittpunkt mit den Scheiben berechnet.

4.2.2.1 Filtern der Punkte nach Distanz zu Strahl

Das Filtern der Punkte lässt sich gut mit CUDA parallelisieren, da die Entfernung von jedem Punkt zu dem Strahl in einem eigenen Thread berechnet werden kann. Im Sequenzdiagramm 4.4 wird dies durch den Aufruf der Funktion *getPointNearRay()* repräsentiert. Die Entfernungsberechnung vom Strahl erfolgt wie im Abschnitt 2.7 beschrieben. Die Schwierigkeit dabei ist, dass sich die Anzahl der Punkte verringert und somit nicht jedem Punkt derselbe Platz im Ergebnis-Array zugewiesen werden kann, den er schon im Ausgangsarray hatte. Ein naiver Lösungsansatz hierfür wäre einen globalen Zähler zu benutzen, der jedes mal mit einer atomaren Addition erhöht wird, sobald ein Thread festgestellt hat, dass sein Punkt die Bedingung erfüllt. Das Problem dabei ist, dass es so viele Kollisionen bei der atomaren Addition gibt, wodurch die Geschwindigkeit reduziert wird.

Deshalb wurde in dieser Arbeit der Ansatz von Andy Adinets [Adi14] mit „Warp-Aggregated Atomics“ zur Lösung des Problems mit den Kollisionen verwendet. Es werden mehrere atomare Operationen in einem Warp in 5 Schritten zu einer einzigen atomaren Operation zusammengefasst.

1. Die Threads in einem Warp wählen einen „Leader Thread“.
2. Die Threads berechnen das komplette Inkrement für den Warp.
3. Der „Leader Thread“ macht eine atomare Addition, um den Offset für den Warp zu berechnen.
4. Der „Leader Thread“ übermittelt den Offset an alle anderen Threads in dem Warp.
5. Jeder Thread addiert sein Index in dem Warp zu dem Offset, um seine Position im Ergebnis-Array zu erhalten.

4.2.2.2 Bestimmen der Oberflächennormalen

Um die Oberflächennormalen der Punkte aus $D_{filtered}$ zu bestimmen, müssen zuerst deren Nachbarn gefunden werden. Die Nachbarn für jeden Punkt $p \in D_{filtered}$ werden parallel gesucht, indem die Distanz von p zu jedem anderen Punkt aus $D_{filtered}$ berechnet wird. Durch die Verwendung von Streams [Ren11] wurde die Suche noch weiter beschleunigt werden. Nachdem die Nachbarn gefunden wurden, werden diese als Eingabe für den Least-Squares Algorithmus aus dem Abschnitt 2.6 verwendet, um die Oberflächennormale n für jeden Punkt p aus $D_{filtered}$ parallel zu berechnen.

4.2.2.3 Bestimmen des Schnittpunktes

Mit der Normalen $\vec{n}$ an den Punkten $p \in D_{filtered}$ wird nun an jedem Punkt die Ebene $E : \vec{n} \cdot (\vec{x} - \vec{p}) = 0$ aufgespannt. Daraufhin werden alle Schnittpunkte des Strahls mit den Ebenen parallel nach dem im Abschnitt 2.8 vorgestellten Prinzip berechnet. Die Auswertung der Schnittpunkte geschieht auf der CPU, indem ausgehend vom Anfang des Strahls der erste Schnittpunkt mit einer Scheibe gesucht wird und von dort alle weiteren Schnittpunkte, die in einer bestimmen Distanz hinter dem ersten Schnittpunkt liegen, gesammelt werden. Aus diesen wird der Mittelwert gebildet.

4.3 Octree

Der Octree findet in der CPU-Implementierung für verschiedene Zwecke Anwendung. In diesem Abschnitt werden die Anwendungsfälle des Octrees in der CPU-Implementierung und die daraus resultierende Implementierung beschrieben.

Die eigene Implementierung des Octrees wurde nach der im Abschnitt 2.3.1 vorgestellten Definition angelegt, weicht jedoch insofern von dieser ab, dass jedes Blatt des Octree maximal *bsize* Punkte enthalten kann anstatt ein Punkt. Wenn ein Knoten des Octrees *bsize* oder weniger Punkte enthält, muss dieser Knoten nicht weiter unterteilt werden. Auf diese Weise wird die Erstellungszeit des Octrees reduziert. Natürlich sollte *bsize* dabei nicht zu groß gewählt werden, da ansonsten bei Suchanfragen trotzdem sehr viele Punkte geprüft werden müssen. Der Octree wird zum Filtern der Punkte nach Distanz zu Strahl und zum Suchen der Nachbarn eines Punktes zur Normalenbestimmung verwendet. Dementsprechend bietet er Funktionen, die diese Funktionalität zur Verfügung stellen. Zunächst muss der Octree aber mit einer Punktwolke erstellt werden.

4.3.1 Erstellen des Octrees

Der Algorithmus zum Erstellen des Octrees folgt direkt aus der im Abschnitt 2.3.1 vorgestellten Definition. Jeder Knoten des Baums ist ein Achsen-ausgerichteter Würfel und speichert die Menge der Punkte, die in diesem Würfel liegen. Der Würfel des Wurzelknotens schließt die gesamte Punktwolke ein. Ausgehend vom Wurzelknoten wird die Punktmenge so partitioniert, dass jede Teilmenge den Punkten aus einem Oktanten des Wurzelknotens entspricht. In den Kindern des Wurzelknotens werden die entsprechenden Würfel und Teilmengen gespeichert. Sollte ein Kind mehr Punkte als *bsize* enthalten, wird es rekursiv weiter unterteilt. Im folgenden wird *bsize* als

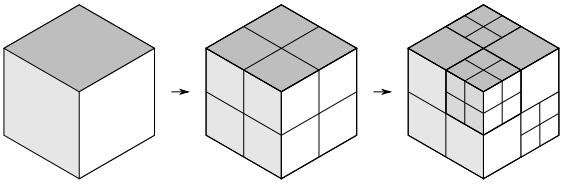


Abbildung 4.5 Bei der Erstellung des Octrees werden in jedem Schritt die Knoten weiter unterteilt, die mehr Punkte als die bucket size enthalten.

bucket size bezeichnet. Abbildung 4.5 zeigt, wie die Knoten in jedem Schritt weiter unterteilt werden.

4.3.2 Funktionen des Octrees

Der Octree stellt verschiedene Funktionen zur Verfügung, die benutzt werden, um die Berechnung der Oberfläche der Punktwolke zu beschleunigen. Dazu gehört unter anderen das Filtern von Punkten nach Distanz zu einem Strahl und das Finden von Punkten in einem bestimmten Radius.

4.3.2.1 Filtern von Punkten nach Distanz zu einem Strahl

Beim Filtern der Punkte nach Distanz zu einem Strahl werden die Punkte gesucht, deren Entfernung zum Strahl kleiner ist als ein Wert *maxDist*. Die Distanzberechnung erfolgt wie im Abschnitt 2.7 dargestellt. Abbildung 4.6 zeigt, wie dazu ein Zylinder mit dem Radius $r_{zylinder} = maxDist$ um den Strahl gelegt wird. Anschließend wird absteigend vom Wurzelknoten geprüft, welche Kinder den Zylinder schneiden, damit die Prüfung rekursiv für die geschnittenen Kindern fortgeführt werden kann, bis kein Kind mehr geschnitten, oder ein Blattknoten erreicht wird. Auf diese Weise werde alle Blattknoten gefunden, die vom Zylinder geschnitten werden. Als letzter Schritt muss geprüft werden, ob die Distanz der Punkte in den geschnittenen Blattknoten kleiner als *maxDist* ist.

4.3.2.2 Finden von Punkten in einem bestimmten Radius

Beim Finden von Punkten in einem bestimmten Radius werden alle Punkte gesucht, deren Entfernung zu einem Suchpunkt q kleiner ist, als ein Wert *maxDist*. Dazu wird, wie in Abbildung 4.7 veranschaulicht, eine Kugel um q mit dem Radius $r_{suche} = maxDist$ gelegt. Daraufhin wird absteigend vom Wurzelknoten geprüft, welche Kinder in der Kugel liegen oder von dieser geschnitten werden. Die Prüfung wird rekursiv für die entsprechenden Kinder fortgesetzt bis keine

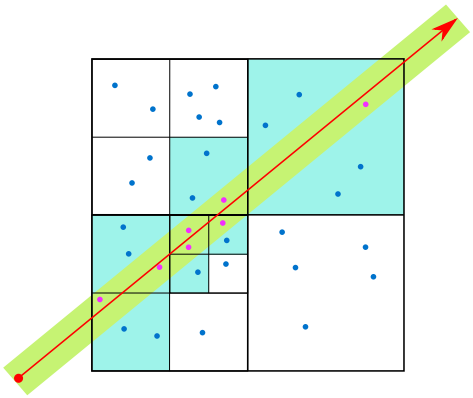


Abbildung 4.6 Um die Punkte in der nähe eines Strahls zu finden, müssen nur die Zellen (blau) geprüft werden, die vom Suchzylinder (grün) um den Strahl geschnitten werden. Aus diesen Zellen werden alle Punkte geprüft, ob diese innerhalb des Suchzylinders liegen

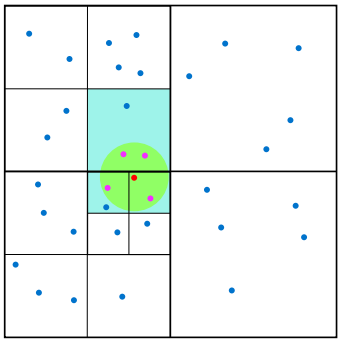


Abbildung 4.7 Bei der Suche nach Punkten in einem bestimmten Radius (grün) müssen nur die Zellen (blau) untersucht werden, die vom Kreis des Suchradius geschnitten werden. Dann werden alle Punkte in diesen Zellen geprüft, ob diese innerhalb des Suchradius liegen.

Kinder mehr die Kriterien erfüllen oder ein Blattknoten erreicht wurde. Dadurch werden alle Blattknoten gefunden, die innerhalb des Radius r_{suche} vom Punkt q liegen. Schlussendlich wird geprüft, welche der Punkte in diesen Blattknoten innerhalb des Radius r_{suche} liegen.

Nach der Implementierung der Schnittpunktberechnung werden im nächsten Kapitel die Experimente beschrieben, mit denen die Implementierung getestet wurde.

Kapitel 5

Experiment Schnittpunktberechnung

Nachdem im letzten Kapitel die Ansätze auf der CPU und GPU zur Schnittpunktberechnung und die Implementierung beschrieben wurden, werden in diesem Kapitel die zum Testen verwendeten Testszenarien vorgestellt und die gemessenen Ergebnisse präsentiert. Des Weiteren wird in diesem Kapitel die notwendige Zeit zum Kopieren der Punktwolke auf die Grafikkarte getestet. Die Tests für die Schnittpunktberechnung wurden auf Windows 10 Version 1803(OS Build 17134.472) mit einem i7 7800x@3.5ghz und einer RTX 2080TI ausgeführt. Der verwendete Compiler war Microsoft (R) C/C++ Version 1915 mit den Optimierungsoptionen /O2 und es wurde CUDA Toolkit 10.0 benutzt.

5.1 Testfälle

Es gibt insgesamt drei Testfälle, um den besten Fall, den schlechtesten Fall und einen durchschnittlichen Fall zu testen. Die Testpunktwolken wurden künstlich generiert, da es so einfach ist die gleiche Punktwolke in unterschiedlichen Größen zum Testen zu erstellen. Der Radius der Scheiben um jeden Punkt, die die Oberfläche an diesem Punkt repräsentiert, wurde bei jedem Test so angepasst, dass der Durchmesser etwas größer ist, als das größte Loch in der Punktwolke. Die Testfälle beinhalten nur die Schnittpunktberechnungszeit. Die Zeit zum Kopieren der Punktwolke auf die Grafikkarte (siehe Abschnitt 5.2.4) und das Erstellen des Octrees für die CPU (siehe Abschnitt 3.2) wird separat betrachtet, da diese Schritte z.B. bei statischen Punktwolken nur einmal am Anfang ausgeführt werden müssten und deshalb nicht groß ins Gewicht fallen würden.

5.1.1 Bester Fall

Im besten Fall trifft der Strahl im rechten Winkel auf nur eine gerade Oberfläche, wie in Abbildung 5.1 illustriert. In diesem Fall kommen sehr wenige Punkte für die Schnittpunktberechnung infrage, weil die Distanz vom Punkt zum Strahl größer ist als der Scheibenradius. Dementsprechend müssen nur für wenige Punkte die Oberflächennormalen bestimmt werden. Zum Testen wurde ein Strahl aus 100 verschiedenen Startpositionen im rechten Winkel auf die Oberfläche geschossen und der Schnittpunkt berechnet.

Im besten Fall liegt der erwartete Zeitaufwand bei der CUDA-Implementierung in $O(n)$. Zuerst müssen alle Punkte nach Distanz zu dem Strahl gefiltert werden, was $O(n)$ Zeit benötigt. Danach müssen für jeden verbleibenden Punkt die Nachbarn gesucht, die Oberflächennormale und der Schnittpunkt der Scheibe an dem Punkt mit dem Strahl bestimmt werden. Weil im besten Fall die Zahl der verbleibenden Punkte k kleiner ist, als die Zahl der verfügbaren CUDA-Kerne, lässt sich dies in $O(k)$ Zeit erledigen. Alle Schritte lassen sich in linearer Zeit zur Punktwolkengröße erledigen und somit ist die erwartete Gesamtlaufzeit im besten Fall $O(n)$.

Bei der CPU-Implementierung ist der Zeitaufwand im besten Fall $O(log(n))$. Dies ergibt sich daraus, dass das Filtern der Punkte durch den Octree normalerweise in logarithmischer Zeit möglich ist. Die Zahl k der verbleibenden Punkte ist im besten Fall konstant und ändert sich nicht mit der Größe der Punktwolke. Um für jeden verbleibenden Punkt die Oberflächennormale zu bestimmen, müssen mit dem Octree für k Punkte die Nachbarn gesucht werden. Der Aufwand einer Nachbarsuche mit dem Octree ist normalerweise $O(log(n))$. Der Aufwand um für k Punkte die Oberflächennormalen zu bestimmen, beträgt also $O(k \cdot log(n))$. Da k im besten Fall abgesehen von kleinen Schwankungen konstant ist, fällt er als Faktor in der Aufwandsbetrachtung weg und der gesamte Aufwand ist $O(log(n))$.

5.1.2 Durchschnittlicher Fall

Für den durchschnittlichen Fall wurde eine Sphäre als Punktwolke gewählt. Diese wurde erstellt, indem die Anzahl der Punkte in der Punktwolke zufällig auf der Oberfläche einer Kugel verteilt wurden. Dann wurden 100 Strahlen generiert, die die Sphäre an einer zufälligen Stelle schneiden. So wird die Oberfläche immer zwei mal geschnitten. In Abbildung 5.2 wird ein möglicher Verlauf der Strahlen dargestellt. Es ist zu sehen, dass der Strahl außerhalb der Sphäre startet und sie an zwei Stellen schneidet. Auf diese Weise treffen die Strahlen aus verschiedenen Winkeln auf die Oberfläche der Punktwolke.

Für den erwarteten Zeitaufwand gilt im durchschnittlichen Fall sowohl für die CPU-Implementierung, als auch für die CUDA-Implementierung, das Gleiche wie im besten Fall. Der einzige Unterschied

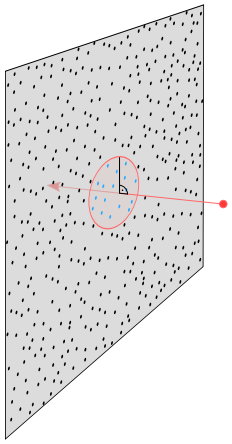


Abbildung 5.1 In dieser Abbildung wird der beste Fall veranschaulicht. Der rote Strahl trifft im rechten Winkel auf die Oberfläche der Punktwolke. Der rote Kreis um den Schnittpunkt entspricht dem Suchradius um den Strahl.

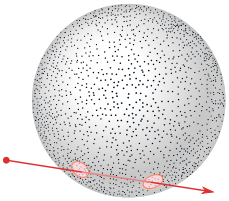


Abbildung 5.2 Ein möglicher Verlauf eines Strahls im durchschnittlichen Testfall. Der Strahl startet außerhalb der Sphäre und schneidet sie an zwei zufälligen Punkten (In diesem Fall links unten und rechts unten).

ist, dass die Anzahl k der verbleibenden Punkte nach dem Filtern größer ist.

5.1.3 Schlechtester Fall

Im schlechtesten Fall verläuft der Strahl sehr dicht und parallel zu der Oberfläche der Punktwolke, wie es in Abbildung 5.3 veranschaulicht ist. Dadurch, dass der Strahl parallel verläuft, gibt es viele Punkte, deren Abstand zum Strahl kleiner ist als der Scheibenradius.

Der erwartete Zeitaufwand der CUDA-Implementierung in diesem Fall ist $O(n)$. Das Filtern der Punkte lässt sich wieder in $O(n)$ Zeit bewerkstelligen, aber für k gilt in diesem Fall $k = \sqrt{n}$. Wenn k nun größer wird als die Anzahl der verfügbaren CUDA-Kerne, liegt die benötigte Zeit zur Nachbarsuche in $O(k^2)$. Die gesamte Laufzeit liegt dann in $O(\sqrt{n}^2) = O(n)$.

Bei der CPU-Implementierung ist der erwartete Aufwand $O(\sqrt{n} \cdot log(n))$, was sich damit begründen lässt, dass das Filtern wieder in $O(\cdot log(n))$ möglich ist. Die Zahl k der verbleibenden Punkte gilt $k = \sqrt{n}$ und der Aufwand zur Oberflächennormalenberechnung eines Punktes ist $O(log(n))$.

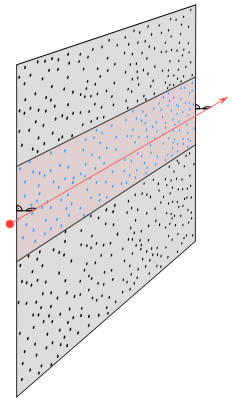


Abbildung 5.3 Der rote Strahl verläuft sehr nahe und parallel zu der Oberfläche der Punktwolke. Auf diese Weise gibt es sehr viele Punkte (grün), für die Oberflächennormalen bestimmt werden müssen.

Somit ist der erwartete Zeitaufwand im schlechtesten Fall $O(\sqrt{n} \cdot log(n))$, da für $\sqrt{n}$ viele Punkte die Oberflächennormale berechnet werden muss. Im Folgenden werden nun die praktischen Ergebnisse vorgestellt.

5.2 Ergebnisse

In diesem Abschnitt werden die gemessenen Ausführungszeiten der drei Testfälle vorgestellt.

5.2.1 Ergebnisse bester Fall

Die Ergebnisse der Schnittpunktberechnung für den besten Fall lassen sich an den Graphen 5.4 für die CPU-Implementierung und 5.5 für die CUDA-Implementierung ablesen. Diese haben auf der X-Achse die Größe der Punktwolke in Punkten und auf der Y-Achse die Ausführungszeit in Sekunden. Für die CPU-Implementierung gibt es drei Graphen, die jeweils die Ausführungszeiten für Octrees mit den bucket sizes 10, 100 und 1000 zeigen. Der Anstieg der Schnittpunktberechnungszeit ist bei den bucket sizes 10 und 100 logarithmisch zur Größe der Punktwolke. Der Graph für die bucket size 1000 schwankt sehr stark, aber wenn man die einzelnen Hochpunkte oder die Tiefpunkte verbindet, ergibt sich auch in logarithmischer Verlauf. Außerdem ist zu erkennen, dass die Ausführungszeit bei größeren bucket sizes zunimmt. Bei dem Graph der CUDA-Implementierung ist zu sehen, dass die Ausführungszeit am Anfang konstant ist und bei größeren Punktwolken linear zu der Größe wächst.

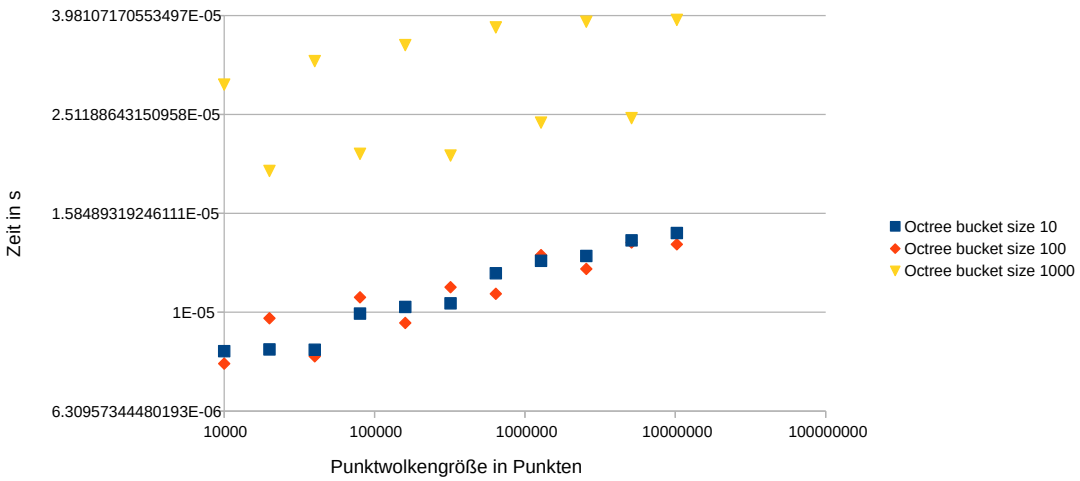


Abbildung 5.4 Die Ergebnisse der CPU-Implementierung im besten Fall.

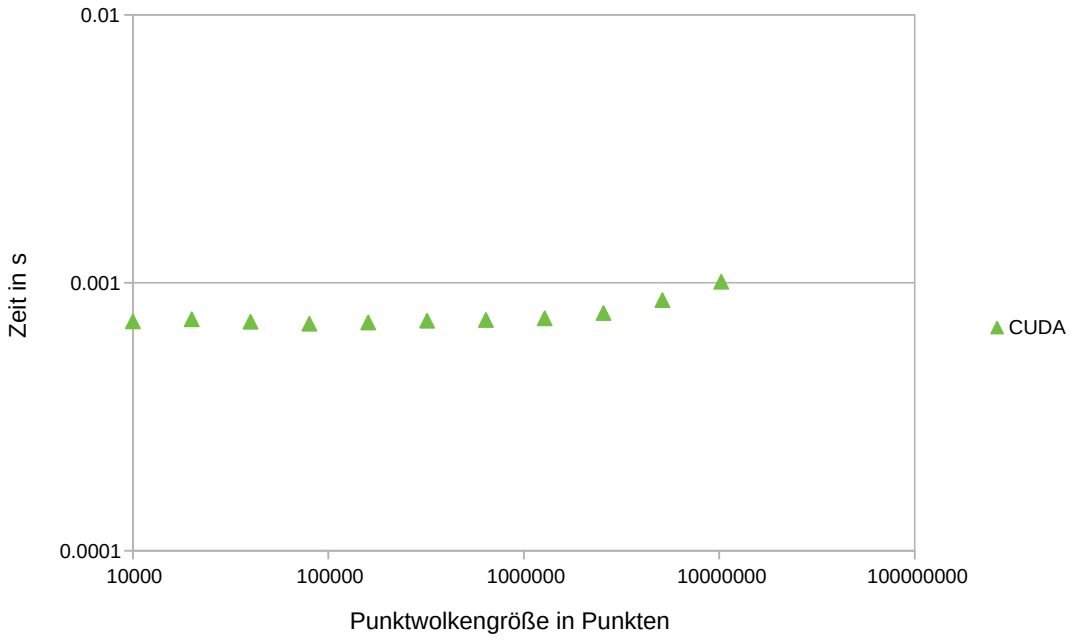


Abbildung 5.5 Die Ergebnisse der CUDA-Implementierung im besten Fall.

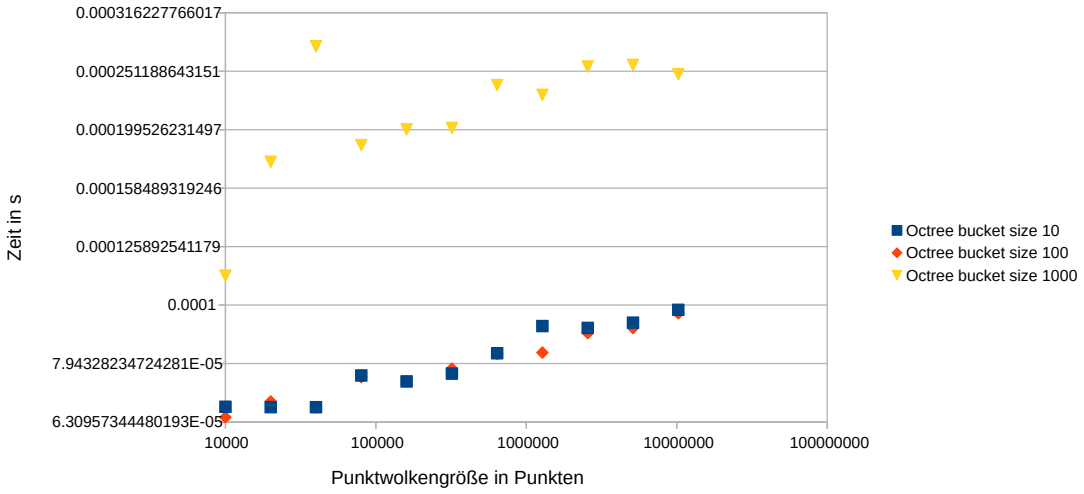


Abbildung 5.6 Die Ergebnisse der CPU-Implementierung im durchschnittlichen Fall.

5.2.2 Ergebnisse durchschnittlicher Fall

Die Ergebnisse des durchschnittlichen Fall sind im Graph 5.6 und 5.7 dargestellt. Diese sind den Graphen des besten Falls sehr ähnlich. Bei der CPU-Implementierung ist wieder ein logarithmischer Anstieg der Ausführungszeit zu der Punktwolkengröße zu sehen und die CUDA-Implementierung ist sie am Anfang konstant und geht bei großen Punktwolken in einen linearen Verlauf über.

5.2.3 Ergebnisse schlechtester Fall

Die Ausführungszeiten des schlechtesten Falls sind in Abbildung 5.8 dargestellt. Dieser zeigt sowohl den Verlauf der CUDA-Implementierung, als auch der CPU-Implementierung mit einem Octree mit der bucket size 10, 100 und 1000. Bei dem Graph der CUDA-Implementierung nimmt die Steigung am Anfang zu und geht dann in einen linearen Anstieg über. Bei der CPU-Implementierung nimmt die Steigung mit der Größe der Punktwolke ab.

5.2.4 Kopieren der Punktwolke auf die Grafikkarte

In diesem Abschnitt wird geprüft wie lange es dauert Punktwolken verschiedener Größe auf die Grafikkarte zu kopieren. Der Grund hierfür ist, dass die entsprechenden Daten auf der Grafikkarte zur Verfügung stehen müssen, damit der Schnittpunkt zwischen einem Strahl und einer

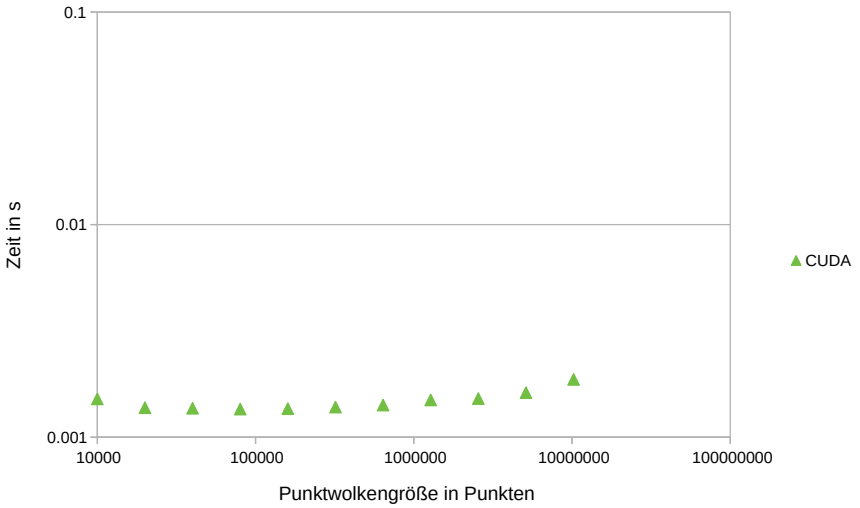


Abbildung 5.7 Die Ergebnisse der CUDA-Implementierung im durchschnittlichen Fall.

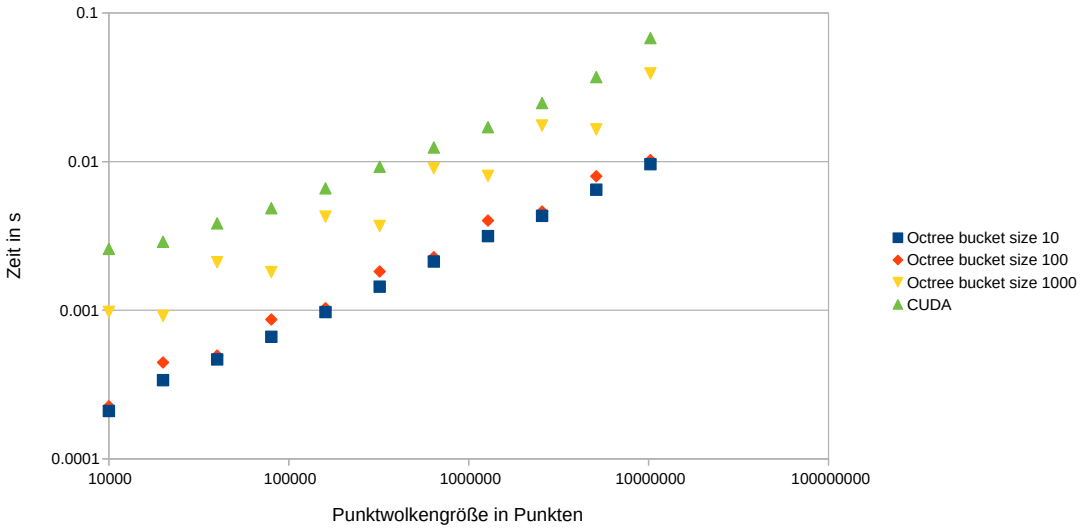


Abbildung 5.8 Die Ergebnisse der Schnittpunktberechnung im schlechtesten Falls.

Punktwolkengröße in Punkten	**Zeit zum Kopieren in s**
1000	9.6447e-05 s
20000	0.000125425 s
40000	0.000160318 s
80000	0.000558015 s
160000	0.00109121 s
320000	0.00164908 s
640000	0.00288507 s
1280000	0.00530292 s
2560000	0.00895198 s
5210000	0.017009 s
10240000	0.0288248 s

Tabelle 5.1 Die gemessenen Zeit, die zum Kopieren von Punktwolken unterschiedlicher Größe auf die Grafikkarte benötigt wird.

Punktwolke auf der Grafikkarte berechnet werden kann. Falls die Punkte schon auf der Grafikkarte vorhanden sind, fällt dieser Schritt bei der Schnittpunktberechnung weg. In Tabelle 5.1 sind die gemessenen Kopierzeiten dargestellt. Es ist zu erkennen, dass die benötigte Zeit linear mit der Anzahl der Punkten steigt.

5.3 Auswertung

In diesem Abschnitt werden die Ergebnisse aus 5.2 ausgewertet und geprüft, ob diese den Erwartungen aus dem Abschnitt 5.1 entsprechen.

5.3.1 Bester Fall

Es wurde bei der CUDA-Implementierung ein linearer Anstieg zur Anzahl der Punkten in der Punktwolke erwartet und der Graph bestätigt diese Erwartung. Lediglich am Anfang ist der Verlauf konstant, weil dort noch das Finden der Nachbarn für die Punkte nach dem Filtern die meiste Zeit kostet. Diese wächst im besten Fall aber nicht mit der Größe der Punktwolke. Ab ca. 1000000 Punkte wird der Aufwand zum Filtern groß genug, dass er eine signifikante Rolle im Vergleich zum Finden der Nachbarn spielt und der Verlauf wird linear.

Die Ergebnisse der CPU-Implementierung entsprechen der erwarteten Laufzeit von $O(log(n))$. Interessant ist allerdings, dass der Graph für den Octree mit der bucket size 1000 stark schwankt. Das liegt daran, dass die Punkte ungefähr gleich verteilt in einer Ebene liegen. Die Punktdichte in

dieser Ebene nimmt mit der Anzahl der Punkten zu, so dass immer zu bestimmten Punktwolkengrößen die Punktdichte so groß ist, dass die meisten Knoten vor dem letzten Unterteilen in ihre Oktanten die bucket size nur knapp überschreiten. Dementsprechend sind dann im Vergleich zur bucket size durchschnittlich sehr wenige Punkte in den Blattknoten sind. Wenn die Punktwolke größer wird, füllen sich die Blattknoten wieder, bis die bucket size überschritten wird. Wenn die Blattknoten viele Punkte enthalten, müssen auch viele Punkte bei der Nachbarsuche überprüft werden.

5.3.2 Durchschnittlicher Fall

Es wurde ein ähnlicher Verlauf wie im besten Fall erwartet, weil die Anzahl der Punkte nach dem Filtern wie im besten Fall nicht mit der Punktwolke wächst. Somit gelten hier die gleichen Erklärungen wie im besten Fall und die Erwartung an den Verlauf des Graphen wurden erfüllt.

5.3.3 Schlechtester Fall

Für die CUDA-Implementierung wurde in diesem Fall ein linearer Verlauf erwartet. Die Ergebnisse entsprechen der Erwartung, lediglich am Anfang gibt es keinen linearen Anstieg. Dies kann damit begründet werden, dass am Anfang noch genügend CUDA-Kerne vorhanden sind, um für jeden verbleibenden Punkt nach dem Filtern die Nachbarn parallel zu suchen. Das Suchen der Nachbarn verbraucht somit am Anfang linear viel Zeit zu der Anzahl k der verbleibenden Punkte und k steigt mit der Wurzel zu der Punktwolkengröße. Sobald nicht mehr genügend CUDA-Kerne vorhanden sind, dauert das Finden der Nachbarn quadratisch viel Zeit, sodass es sich ausgleicht und der Verlauf linear wird.

Bei der CPU-Implementierung ist der erwartete Zeitaufwand $O(\sqrt{n} \cdot log(n))$ mit n als Punktwolkengröße, was sich damit begründen lässt, dass die Anzahl der Punkte, die nahe genug am Strahl liegen, quadratisch mit der Größe der Punktwolke wächst und für jeden dieser Punkte ca. $log(n)$ viel Zeit für die Nachbarsuche benötigt wird. Der Graph entspricht der Erwartung.

5.4 Vergleich mit Schaufler und Jensen

Der Vollständigkeit halber ist in Tabelle 5.2 ein Vergleich zwischen den Ergebnissen von Schaufler und Jensen und der Implementierung dieser Arbeit, auch wenn dieser Vergleich wenig aussagekräftig ist, da bei den beiden Ansätzen unterschiedliche Ziele verfolgt wurden. Schaufler und

		Modell	**Punkte**	Strahlen/Sekunde
Schaufler und Jensen		Bunny[a]	34834	34204
		Bunny[b]	34834	32142
		Buddha[b]	543652	12309
		Buddha[c]	543652	15731
Diese Arbeit	CPU[d]	Sphäre	20000	14949
		Sphäre	40000	14953
		Sphäre	640000	12089
	CPU[e]	Sphäre	20000	1553
		Sphäre	40000	846
		Sphäre	640000	33
	CUDA	Sphäre	20000	875
		Sphäre	40000	888
		Sphäre	640000	810

Tabelle 5.2 Ein Vergleich der Ergebnisse von Schaufler und Jensen und dieser Arbeit. Schaufler und Jensen haben ihre Tests auf einem dual PII-400MHz ausgeführt, während in dieser Arbeit ein Intel i7 7800x@3,5Ghz und eine RTX2080TI verwendet wurden.
a: 5 Octree levels
b: 6 Octree levels
c: 7 Octree levels
d: CPU-Implementierung ohne Erstellungszeit des Octree und bucket size 10
e: CPU-Implementierung mit Erstellungszeit des Octree und bucket size 1000

Jensen [SJ00] wollen auf einer statischen Punktwolke so viele Schnittpunkte wie möglich berechnen, während bei dieser Arbeit aufgrund der dynamischen Punktwolke die Laufzeit für einzelne Schnittpunktberechnungen optimiert wurde. Wie zu erwarten lassen sich deshalb mit dem Ansatz von Schaufler und Jensen mehr Strahlen pro Sekunde berechnen, weil diese die für die Schnittpunktberechnung verwendeten Scheiben vorberechnen, währenddessen bei dem Ansatz dieser Arbeit die Scheiben während der Schnittpunktberechnung bestimmt werden, da das vorberechnen der Scheiben aufgrund der dynamischen Punktwolke in dieser Arbeit nicht sinnvoll ist. Des Weiteren gibt es bei der CPU-Implementierung ein Ergebnis für die reine Schnittpunktberechnung und einen weiteres, in dem auch die Erstellungszeit des Octree berücksichtigt wurde. Es ist zu erkennen, dass die CPU-Implementierung für große Punktwolken wesentlich langsamer als die CUDA-Implementierung ist, wenn die Erstellungszeit des Octrees für jeden Schnittpunkt mit eingerechnet wird. Schaufler und Jensen beziehen die Erstellungszeit ihrer Beschleunigungsdatenstruktur aufgrund der statischen Punktwolke in ihrem Szenario nicht mit ein.

5.5 Zusammenfassung

In diesem Abschnitt wird geklärt, ob der Schnittpunkt zwischen einem Strahl und einer Punktwolke mit dem Ansatz aus dieser Arbeit in Echtzeit berechnet werden kann und es wird die CUDA-Implementierung mit der CPU-Implementierung verglichen.

Die Laufzeitkomplexität der CPU-Implementierung ist aufgrund des Octrees besser als die der CUDA-Implementierung und ist deshalb für statische Punktwolken besser geeignet. Allerdings sind statische Punktwolken nicht das Ziel dieser Arbeit und für diese gibt es bereits Ansätze, die besser geeignet sind. Werden nur wenige Schnittpunkte pro Punktwolke berechnet, weil diese sich z.B. ständig ändert, müssen die Erstellungszeit des Octrees und das Kopieren der Daten auf die Grafikkarte berücksichtigt werden. In diesem Fall ist die CUDA-Implementierung deutlich schneller, da das Erstellen des Octrees für jede neue Punktwolke sehr aufwendig ist. So können mit der CUDA-Implementierung Schnittpunkte für dynamische Punktwolken mit bis zu $2,5 \cdot 10^6$ Punkten in Echtzeit berechnet werden. Dies ergibt sich aus der Zeit zur Schnittpunktberechnung, die im schlechtesten Fall ca. 20 ms dauert und das Kopieren dauert weitere 10 ms. Es können also noch 30 Schnittpunkte pro Sekunde berechnet werden. Bei der CPU-Implementierung liegt die Grenze bei 700000 Punkten, da das Erstellen des Octrees in diesem Fall ca. 15 ms dauert (siehe Abschnitt 3.2.2) und die Schnittpunktberechnung im schlechtesten Fall zusätzliche 10 ms. Des Weiteren hängt die Performance stark von dem gewählten Scheibenradius ab. Ist der Scheibenradius zu groß, müssen viele Punkte geprüft werden und die Performance nimmt ab. Wenn der Scheibenradius zu klein ist, entstehen Löcher in der Punktwolke und die Schnittpunktberechnung funktioniert nicht mehr zuverlässig.

Zusammenfassend lässt sich sagen, dass die CUDA-Implementierung den Schnittpunkt eines Strahls mit einer Punktwolke mit bis zu $2,5 \cdot 10^6$ Punkten in Echtzeit berechnen kann während die CPU-Implementierung dies nur bei Punktwolken mit bis zu 700000 Punkten schafft.

Nach den Experimenten und Auswertungen kommt nun im Anschluss das Fazit der Arbeit.

Kapitel 6

Fazit

In dieser Arbeit wurde gezeigt, dass sich der Schnittpunkt zwischen einem Strahl und einer Punktwolke in Echtzeit bestimmen lässt. Für dieses Problem wurde ein Ansatz entwickelt, bei dem Scheiben um die Punkte der Punktwolke gelegt werden, die die Oberfläche der Punktwolke repräsentieren. Die besonderen Rahmenbedingungen dabei waren, dass sich die Punktwolke regelmäßig ändert und dass der Schnittpunkt in Echtzeit bestimmt werden soll. Weitergehend wurde eine CPU- und eine CUDA-Implementierung des Ansatzes entwickelt und getestet. Das Ergebnis dieser Tests war, dass die CUDA-Implementierung besser für sich ändernde Punktwolke geeignet ist, da durch die große Anzahl zur Verfügung stehenden CUDA-Cores auf das Erstellen von Beschleunigungsdatenstrukturen verzichtet werden kann. Dies ist das Problem der CPU-Implementierung, weil das Erstellen von Beschleunigungsdatenstrukturen für jede neue Punktwolke sehr aufwendig ist. So ist es mit der CUDA-Implementierung möglich für Punktwolken mit bis zu $2,6 \cdot 10^6$ Punkten den Schnittpunkt zuverlässig in Echtzeit zu berechnen. Die Schnittpunktberechnung dauert bei Punktwolken bis zu dieser Größe auch im schlechtesten Fall weniger als 33 ms und erfüllt somit noch die Echtzeitbedingung. Bei der CPU-Implementierung ist dies nur bis zu 700000 Punkten möglich. Für statische Punktwolken lohnt es sich am Anfang einen Beschleunigungsdatenstruktur anzulegen, weil so der Aufwand für die eigentlichen Schnittpunktberechnungen deutlich verringert werden kann. In diesem Fall sollte aber ohnehin ein anderer Ansatz als der in dieser Arbeit vorgestellte betrachtet werden, da der hier vorgestellte Ansatz nicht für statische Punktwolken optimiert ist. Eine weitere Einschränkung dieses Ansatzes ist, dass durch das Festlegen des Scheibenradius am Anfang die Punkte der Punktwolke einigermaßen gleich verteilt sein müssen, da ansonsten Löcher in der Oberfläche entstehen und der Schnittpunkt nicht zuverlässig berechnet werden kann.

Kapitel 7

Ausblick

Im Ergebnisteil dieser Arbeit ist deutlich geworden, dass sich die CUDA-Implementierung besser zur Schnittpunktberechnung mit dynamischen Punktwolken eignet, da das Erstellen von Beschleunigungsdatenstrukturen für jede neue Punktwolke in der CPU-Implementierung zu lange dauert. Bei der Schnittpunktberechnung selber ist die CPU-Implementierung jedoch deutlich schneller. In einer weiterführenden Arbeit wäre es interessant zu prüfen, ob das Erstellen von Beschleunigungsdatenstrukturen durch Parallelisierung mit CUDA beschleunigt werden kann und in diesem Zusammenhang auch weitere Beschleunigungsdatenstrukturen zu testen. Außerdem könnte man die Nachbarsuche in der CUDA-Implementierung verbessern. Dazu könnte man sich z.B. die Fast Fixed-Radius Nearest Neighbors [Hoe] anschauen. Des Weiteren muss bei dem in dieser Arbeit vorgestellten Ansatz der Radius der Scheiben, die um jeden Punkt liegen und die Oberfläche repräsentieren, am Anfang gewählt werden. Aus diesem Grund funktioniert der Ansatz nur bei Punktwolken, bei denen die Punkte ungefähr gleich verteilt sind. In einer weiterführenden Arbeit könnte man den Ansatz so anpassen, dass der Scheibenradius dynamisch an die dichte der Punktwolke angepasst wird. Dafür könnte man möglicherweise die Größe der Blattknoten des Octrees zu verwenden.

Kapitel 8

Appendix

8.1 Abbildungsverzeichnis

8.2 Tabellenverzeichnis

8.3 Literatur

[Adi14] Andy Adinets. *CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics*. 2014. URL: *https://devblogs.nvidia.com/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/* (abgerufen am 11. 01. 2019).

[BR14] Jose Luis Blanco und Pranjal Kumar Rai. *nanoflann: a C++ header-only fork of FLANN, a library for Nearest Neighbor (NN) wih KD-trees*. *https://github.com/jlblancoc/nanoflann*. 2014. (Abgerufen am 11. 01. 2019).

[BSC15] Jens Behley, Volker Steinhage und Armin B. Cremers. »Efficient Radius Neighbor Seach in Three-dimensional Point Clouds«. In: *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*. 2015.

[Erb+11] Dr. Rüdiger Erbrecht, Matthias Felsch, Dr. Hubert König, Dr. Wolfang Kricke, Karlheinz Martin, Wolfgang Pfeil, Dr. Rolf Winter und Willi Wörtenfeld. *Das große Tafelwerk*. Berlin, Germany: Cornelsen Verlag, 2011. ISBN: 978-3-464-57143-9.

[Fu+] Ada Wai-chee Fu, Polly Mei-shuen Chan, Yin-Ling Cheung und Yiu Sang Moon. »Dynamic Vp-tree Indexing for N-nearest Neighbor Search Given Pair-wise Distances«. In: ().

[Han12] Steve Hanov. *VP trees: A data structure for finding stuff fast*. 2012. URL: *http://stevehanov.ca/blog/index.php?id=130* (abgerufen am 11. 01. 2019).

[Hoe] Rama C. Hoetzlein. *FAST FIXED-RADIUS NEAREST NEIGHBORS: INTERACTIVE MILLION-PARTICLE FLUIDS*. URL: *http://on-demand.gputechconf.com/gtc/2014/presentations/S4117-fast-fixed-radius-nearest-neighbor-gpu.pdf* (abgerufen am 11. 01. 2019).

[KDH10] Kamran Karimi, Neil G. Dickson und Firas Hamze. »A Performance Comparison of CUDA and OpenCL«. In: *CoRR* abs/1005.2581 (2010). arXiv: *1005.2581*. URL: *http://arxiv.org/abs/1005.2581*.

[Kib07] Ashraf Masood Kibriya. »Fast Algorithms for Nearest Neighbor Search«. Magisterarb. University of Waikato, 2007.

[KZN08] Neeraj Kumar, Li Zhang und Shree K. Nayar. »What is a Good Nearest Neighbors Algorithm for Finding Similar Patches in Images?« In: *The 10th European Conference on Computer Vision (ECCV)*. Okt. 2008.

[Lin07] Rosenthal P. Linsen L. Mueller K. »Splat-based Ray Tracing of Point Clouds«. In: *Journal of WSCG*. Hrsg. von Vaclav Skala. Plzen, Czech Republic: Václav Skala - UNION Agency, 2007, S. 51–58. ISBN: 978-80-86943-00-8.

[LZ06] Elmar Langetepe und Gabriel Zachmann. *Geometric Data Structures for Computer Graphics*. 1. Aufl. AK Peters, 2006. URL: *http://akpeters.com/product.asp?ProdCode=2353*.

[Mea82] Donald Meagher. »Geometric modeling using octree encoding«. In: *Computer Graphics and Image Processing* 19.2 (1982), S. 129–147. ISSN: 0146-664X.

[NVIa] NVIDIA. *CUDA Zone*. URL: *https://developer.nvidia.com/cuda-zone* (abgerufen am 11. 01. 2019).

[NVIb] NVIDIA. *GEFORCE GTX 1080 Ti*. URL: *https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/* (abgerufen am 11. 01. 2019).

[NVIc] NVIDIA. *Parallele Berechnungen mit CUDA*. URL: *http://www.nvidia.de/object/cuda-parallel-computing-de.html* (abgerufen am 11. 01. 2019).

[Ren11] Steve Rennich. *CUDA C/C++ Streams and Concurrency*. 2011. URL: *http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf* (abgerufen am 11. 01. 2019).

[SE02] Philip J. Schneider und David Eberly. *Geometric Tools for Computer Graphics*. New York, NY, USA: Elsevier Science Inc., 2002. ISBN: 1558605940.

[SJ00] Gernot Schaufler und Henrik Wann Jensen. »Ray Tracing Point Sampled Geometry«. In: *Rendering Techniques 2000*. Hrsg. von Bernard Péroche und Holly Rushmeier. Vienna: Springer Vienna, 2000, S. 319–328. ISBN: 978-3-7091-6303-0.

[WB05] Heinz Wörn und Uwe Brinkschulte. *Echtzeitsysteme Grundlagen, Funktionsweisen, Anwendungen*. SSpringer-Verlag Berlin Heidelberg, 2005. ISBN: 978-3-540-20588-3.

[Wei16] Martin Weinmann. »Preliminaries of 3D Point Cloud Processing«. In: *Reconstruction and Analysis of 3D Scenes: From Irregularly Distributed 3D Points to Object Classes*. Cham: Springer International Publishing, 2016, S. 17–38. ISBN: 978-3-319-29246-5.

DOI: *10.1007/978-3-319-29246-5_2*. URL: *https://doi.org/10.1007/978-3-319-29246-5_2*.

[Wel13] Rene Weller. *New Geometric Data Structures for Collision Detection and Haptics*. Springer Series on Touch and Haptic Systems. Springer International Publishing, 2013. ISBN: 9783319010199. URL: *http://www.springer.com/computer/theoretical+computer+science/book/978-3-319-01019-9*.

[WW92] Alan Watt und Mark Watt. *Advanced Animation and Rendering Techniques*. New York, NY, USA: ACM, 1992. ISBN: 0-201-54412-1.

[Yia70] Peter Yianilos. »Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces«. In: (Feb. 1970).

8.4 Glossar

Beschleunigungsdatenstruktur
Eine Datenstruktur, die dazu dient die Suche nach bestimmten Daten zu beschleunigen.
S. 2, 3, 5, 6, 15–17, 19, 22, 38, 41, 43, 46

bucket size
Die bucket size beschreibt die maximale Kapazität der Blattknoten eines Baums.
S. 16, 17, 27, 32, 34, 36–38

Echtzeit
Der Begriff Echtzeit wird in Abschnitt 2.5 definiert.
S. 5, 39, 41

GPU
graphical processing unit
S. 9, 10, 48

K-d-Tree
k-dimensionaler Baum
S. 8

Kernel
Im Zusammenhang mit CUDA und OpenCL bezeichnet ein Kernel den Code, der auf der GPU ausgeführt wird [KDH10].
S. 10, 23

NN-Suche
Nearest-Neighbors-Suche
S. 5, 6

Octree

Octrees werden im Abschnitt 2.3.1 definiert.

S. 4–7, 16, 17, 19–22, 26, 27, 29, 36, 38, 39, 43

OpenCL

Open Computing Language

S. 10, 48

Radius-Nachbar-Suche

Bei der Radius-Nachbar-Suche geht es darum alle Punkte zu finden, die innerhalb eines Radius r um den Suchpunkt p liegen.

S. 16, 17, 46

Strahl

Ein Strahl ist eine gerade Linie mit einem Startpunkt und ohne Endpunkt.

S. 1–5, 11–15, 19–23, 25–28, 30–32, 34, 37–39, 41

Vp-Tree

Vantage-Point Tree

S. 9, 16, 17