Technische Universität Berlin

Computer Graphics
Fakultät IV
Marchstraße 23
10587 Berlin
http://www.cg.tu-berlin.de

Bachelorarbeit

Efficiently computing extremal points in cylindrical coordinates

Fabian Stroschke

Matrikelnummer: 391575
13.11.2023

Prüfer
Prof. Dr. Marc Alexa
Prof. Dr. Benjamin Blankertz

Betreuer
Dimitris Bogiokas

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 13.11.2023

.....

(Unterschrift)

Zusammenfassung

Das Finden von Extremalpunkten kann benutzt werden, um viele verschiedene Probleme zu lösen. Bekannte Beispiele dafür sind die Berechnung von konvexen Hüllen und die lineare Optimierung. Beide Probleme wurden in der Vergangenheit schon viel untersucht und wurden sehr gut optimiert. Das Finden einzelner Extremalpunkte zu beschleunigen wurde hingegen in weniger ausführlich betrachtet.

In dieser Arbeit wurde untersucht, wie Extremalpunkte effizient berechnet werden können, wenn eine Linie oder Ebene um eine Achse rotiert wird, bis sie einen Extremalpunkt trifft. Da dieses Problem für unsortierte Punkte nicht schneller als in $O(n)$ gelöst werden kann, wurde dafür angenommen, dass die Punkte bereits in einer räumlichen Datenstruktur gegeben sind. Auf diese Weise wurde ein Algorithmus entwickelt, welcher die Eigenschaften dieser Datenstrukturen nutzt, um die Extremalpunkte effizient zu berechnen und mit minimalen Änderungen für viele verschiedene räumliche Datenstrukturen genutzt werden kann. Im schlimmsten Fall hat dieser Algorithmus eine Laufzeit von $O(n)$, es konnte aber empirisch gezeigt werden, dass er im Durchschnitt deutlich schneller ist und sogar Laufzeiten von $O(\sqrt{n})$ erreichen kann.

Da das Bestimmen von Extremalpunkten durch Rotation nahezu identisch mit einem Teil des Gift Wrapping Algorithmus ist, wurde auf dieser Basis auch eine modifizierte Version des Gift Wrapping Algorithmus entwickelt, um zu zeigen, wofür die Berechnung von Extremalpunkten genutzt werden kann. Diese Modifizierung von Gift Wrapping ist deutlich schneller als der normale Gift Wrapping Algorithmus, hat eine bessere empirische Laufzeit als andere neue Modifikationen von Gift Wrapping und ist in manchen Fällen sogar schneller als der Quickhull Algorithmus.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel . . . 2
- 1.3 Ablauf . . . 2

2 Grundlagen **3**
- 2.1 Extremalpunkte . . . 3
  - 2.1.1 Nutzung . . . 3
- 2.2 Konvexe Hüllen . . . 4
  - 2.2.1 Nutzung . . . 5
  - 2.2.2 Algorithmen . . . 5
- 2.3 Räumliche Datenstrukturen . . . 8
  - 2.3.1 Quadtrees und Octrees . . . 9
  - 2.3.2 k-d-Bäume . . . 10
- 2.4 CGAL - *The Computational Geometry Algorithms Library* . . . 10

3 Algorithmus und Implementierung **13**
- 3.1 Benachbarten Extremalpunkt finden . . . 13
  - 3.1.1 Laufzeit . . . 18
  - 3.1.2 Anwendung für Gift-Wrapping Algorithmus . . . 20
- 3.2 Implementierungsdetails . . . 21
  - 3.2.1 Extremalpunkte finden . . . 21
  - 3.2.2 Konvexe Hüllen . . . 22

4 Analyse und Vergleich **23**
- 4.1 Einen Extremalpunkt finden . . . 23
- 4.2 Berechnen der konvexen Hülle . . . 24

5 Fazit **31**

Literatur **33**

1 Einleitung

Extremalpunkte sind ein Konzept, dass in der Mathematik und Informatik weit verbreitet ist. Sie sind eng verwandt mit konvexen Hüllen und das Bestimmen von Extremalpunkten kann genutzt werden, um Optimierungsprobleme mit Hilfe von linearer Optimierung zu lösen oder konvexe Hüllen zu berechnen.

Ein Punkt p einer konvexen Menge K ist ein Extremalpunkt, wenn p nicht als Konvexkombination von Punkten aus K dargestellt werden kann[2]. Wenn K eine kompakte konvexe Menge ist und S die Menge aller Extremalpunkte von K, dann ist S die konvexe Hülle von K [20].

1.1 Motivation

Es gibt verschiedene Möglichkeiten, um einzelne Extremalpunkte zu finden. Ein simpler Algorithmus dafür wäre, eine Linie oder Ebene durch eine Menge von Punkten zu schieben und zu bestimmen, welche Punkte als Erstes und als Letztes davon getroffen werden. Diese Punkte sind Extremalpunkte der Menge. Eine andere Möglichkeit ist es, eine Linie oder Ebene um eine Achse zu rotieren, bis sie einen Punkt aus der Menge trifft. Dazu können die Punkte in polaren oder zylindrischen Koordinaten um die Achse dargestellt werden, und dann die Punkte mit dem größten oder kleinsten Winkel gesucht werden. Dieser Ansatz wird zum Beispiel bei Gift Wrapping [12, 17] genutzt, um die konvexe Hülle zu berechnen. Mit diesem Ansatz kann ein Extremalpunkt in linearer Zeit berechnet werden. Wenn alle Extremalpunkte einer Menge auf diese Weise bestimmt werden sollen, kann das dazu führen, dass die Laufzeit davon $O(n^2)$ beträgt.

Da die Menge der Extremalpunkte der konvexen Hülle entspricht, kann dieses Problem effizienter gelöst werden, indem die konvexe Hülle berechnet wird. Das Berechnen von konvexen Hüllen ist in der Informatik ein gut untersuchtes Thema und es gibt viele Algorithmen, welche dieses Problem effizient lösen können. Bekannte Vertreter davon sind Gift Wrapping/Jarvis-March[12, 17], Graham-Scan[16], Divide and Conquer[28] und Quickhull[4]. Kirkpatrick und Seidel [19] und Chan et al. [11, 10] haben Algorithmen entwickelt, welche die konvexe Hülle in einer theoretisch optimalen Laufzeit von $O(nlog(h))$ berechnen können. Dabei ist h die Anzahl der Punkte, aus denen die konvexe Hülle besteht.

Wie einzelne Extremalpunkte schneller als in linearer Zeit gefunden werden können, ist ein weniger gut untersuchtes Problem. Da alle Punkte einer Menge untersucht werden müssen, um einen Extremalpunkt zu finden, kann dieses Problem für unsortierte Punkte nicht schneller als in linearer Zeit gelöst werden. Ein Ansatz ist deswegen, Datenstrukturen zu nutzen, um die Suche von Extremalpunkten zu beschleunigen. Edelsbrunner und

Maurer [14] nutzen dafür eine Datenstruktur, mit der Anfragen zu konvexen Hüllen und Extremalpunkten schnell beantwortet werden können. Damit können Extremalpunkte, durch die Rotation einer Linie oder Ebene, in $O(log(n))$ gefunden werden.

Wenn mit vielen Punkten gearbeitet wird, ist es oft sinnvoll, diese in räumlichen Datenstrukturen, wie Quadtrees [15] oder k-d-Bäumen [6], zu speichern. Mit diesen Datenstrukturen sind viele Probleme schneller lösbar. Wenn diese Datenstrukturen genutzt werden können, um Extremalpunkte schneller zu finden, kann darauf verzichtet werden die Datenstruktur von Edelsbrunner und Maurer zu bauen, damit das Problem effizienter zu gelöst werden kann

1.2 Ziel

Das Ziel dieser Arbeit ist es zu zeigen, dass Extremalpunkte mithilfe von Datenstrukturen effizient gefunden werden können und diese Eigenschaft genutzt werden kann, um einen Algorithmus zu entwickeln, welcher besser als Gift Wrapping ist und für bereits existierende Datenstrukturen sogar besser als andere Algorithmen zur Berechnung von konvexen Hüllen sein kann. Dafür werden Algorithmen implementiert, welche räumliche Datenstrukturen nutzen, um Extremalpunkte schneller zu finden. Diese Algorithmen werden dann genutzt, um einen effizienteren Gift Wrapping Algorithmus zu entwickeln und diesen mit anderen Algorithmen zu vergleichen.

1.3 Ablauf

Diese Arbeit besteht aus fünf Kapiteln. Im Anschluss an die Einleitung werden in Kapitel 2 die grundlegenden Konzepte von Extremalpunkten, konvexen Hüllen und räumlichen Datenstrukturen genauer erklärt. Da in dieser Arbeit CGAL [35] benutzt wird, folgt dann eine kurze Beschreibung dieser Bibliothek und wofür sie genutzt wird. In Kapitel 3 werden dann die theoretischen Aspekte des hier entwickelten Algorithmus erläutert und danach die Details der Implementierung behandelt. Um die Laufzeit der implementierten Algorithmen zu untersuchen, werden sie in Kapitel 4 für verschiedene Datensätze getestet und mit anderen Algorithmen verglichen. Die Ergebnisse der Arbeit werden in Kapitel 5 zusammengefasst und mögliche Verbesserungen vorgeschlagen, mit denen die Algorithmen in zukünftigen Arbeiten verbessert werden könnten.

2 Grundlagen

In diesem Kapitel werden die Grundlagen zu Extremalpunkten und konvexen Hüllen erklärt. Dafür wird darauf eingegangen, wie die beiden Themen zusammenhängen, wofür sie genutzt werden und welche Arbeiten sich mit diesen Themen beschäftigt haben. Anschließend erfolgt eine kurze Einführung der räumlichen Datenstrukturen, welche in dieser Arbeit genutzt werden, sowie ein Überblick über die CGAL-Bibliothek.

2.1 Extremalpunkte

Als Extremalpunkte, im Englischen meistens *extreme points* oder *extremal points* genannt, werden die Punkte einer Menge bezeichnet, welche nicht als Konvexkombination von anderen Punkten aus dieser Menge dargestellt werden können. Eine Konvexkombination ist eine Linearkombination, bei der alle Koeffizienten positiv sind und die Summe aller Koeffizienten 1 ergibt [2].

2.1.1 Nutzung

Extremalpunkte werden oft gesucht, um andere verwandte Probleme zu lösen, welche sich auf eine Suche von Extremalpunkten reduzieren lassen oder bei denen Extremalpunkte die optimale Lösung darstellen. Aus diesem Grund konzentrieren sich viele Arbeiten nicht ausdrücklich auf Extremalpunkte, sondern behandeln sie und deren Suche indirekt, um andere Probleme zu lösen. Zwei der größten Probleme, welche Extremalpunkte und ihre Eigenschaften benutzen, sind die lineare Optimierung, oft auch lineare Programmierung genannt, und die Bestimmung von konvexen Hüllen. Beide Probleme werden schon seit der Entwicklung von Computern und den Anfängen der Informatik untersucht und stammen ursprünglich aus der Mathematik. Dementsprechend existieren für beide Probleme bereits gut optimierte Lösungen und für spezielle Fälle sogar optimale Lösungen. Für Konvexe Hüllen werden diese in 2.2 näher behandelt.

Lineare Optimierung

Lineare Optimierung bildet die Grundlage, um eine Vielzahl von Problemen zu lösen. Es kann genutzt werden, um Produktionsabläufe zu optimieren, Kosten zu sparen oder die Nutzung von Ressourcen zu verbessern. Es können aber auch verschiedene Probleme aus der Mathematik oder Informatik auf eine lineare Optimierung reduziert werden. In [24] wird darauf eingegangen, welche Probleme sich auf lineare Optimierung reduzieren lassen und welche Bereiche durch lineare Optimierung verbessert werden konnten.

Bei der linearen Optimierung werden in der Regel eine Reihe von linearen Ungleichungen aufgestellt und dann nach dem Punkt gesucht, welcher eine lineare Funktion maximiert. In [2] wird gezeigt, dass die Lösungsmenge eines Systems von linearen Ungleichungen auch als Polytop interpretiert werden kann. Die Ecken dieses Polytops stellen die Extremalpunkte der Lösungsmenge dar, und einer dieser Extremalpunkte ist die optimale Lösung des Problems. Des weiteren wird dort die Simplex-Methode erklärt, welche diese Eigenschaften nutzt, um das Optimierungsproblem schnell zu lösen. Da die optimale Lösung ein Extremalpunkt ist, müssen nur die Extremalpunkte untersucht werden, um eine Lösung zu finden. Dafür müssen nicht alle Extremalpunkte untersucht werden. Es reicht, wenn von einem ersten Extremalpunkt ausgehend, der benachbarte Extremalpunkt gesucht wird, welcher das Problem am besten optimiert. Der Vorgang wird für diesen Punkt so lange wiederholt, bis kein benachbarter Extremalpunkt eine bessere Lösung ist.

Konvexe Hüllen und verwandte Arbeiten

Bei der Bestimmung von konvexen Hüllen kann das Problem darauf reduziert werden, alle Extremalpunkte einer Menge zu bestimmen, da die Menge der Extremalpunkte der konvexen Hülle entspricht. Da bei der konvexen Hülle alle Extremalpunkte gesucht werden, wird selten die Suche nach einzelnen Extremalpunkten optimiert, sondern versucht, möglichst viele wiederkehrende Schritte zu vermeiden. In 2.2 wird weiter ausgeführt, warum die konvexe Hülle der Menge aller Extremalpunkte entspricht, wie konvexe Hüllen bestimmt werden können und wofür sie genutzt werden.

Die Eigenschaft, dass die konvexe Hülle aus allen Extremalpunkten einer Menge besteht, kann aber auch umgekehrt genutzt werden, um schnell Abfragen zu Extremalpunkten zu lösen. Brodal und Jacob entwickeln in [9] eine dynamische Datenstruktur für Punkte in der Ebene ($\mathbb{R}^2$). Diese Datenstruktur ermöglicht es, die Menge der Extremalpunkte für jeden hinzugefügten oder entfernten Punkt in $O(log(n))$ aktuell zu halten und benachbarte Extremalpunkte in $O(log(n))$ zu ermitteln. In [14] wird von Edelsbrunner und Maurer eine Datenstruktur entwickelt, mit der in $O(log(n))$ bestimmt werden kann, welcher Extremalpunkt als Erstes getroffen wird, wenn eine Ebene um eine Linie rotiert wird. Das ist dasselbe Problem, welches auch mit dieser Arbeit gelöst werden soll. Edelsbrunner und Maurer benutzen dafür eine Datenstruktur, welche hierarchisch Informationen über die konvexe Hülle speichert.

Für die Datenstruktur von Edelsbrunner und Maurer sowie die Datenstruktur von Brodal und Jacob, muss zuerst die konvexe Hülle berechnet werden. Diese Ansätze eignen sich damit nicht dafür, Algorithmen wie Gift Warpping zu beschleunigen.

2.2 Konvexe Hüllen

Eine Menge $C \subseteq \mathbb{R}^n$ ist konvex, wenn alle Konvexkombinationen von Elementen aus C in C enthalten sind, also $\sum_{i=1}^{m} \lambda_i x_i \in C$ für $m = |C|, x_i \in C, \lambda_i \geq 0$ und $\sum_{i=1}^{m} \lambda_i = 1$ gilt [30]. Des Weiteren wird in [30] gezeigt, dass Halbräume konvex sind und bewiesen,

dass die Schnittmenge von konvexen Mengen auch konvex ist. Diese Aussagen werden benutzt, um die konvexe Hülle zu definieren. Die konvexe Hülle $conv(C)$ von C wird als Schnittmenge aller konvexen Mengen, welche C enthalten, definiert. Die konvexe Hülle ist damit die kleinste konvexe Menge, welche C enthält. Krein und Milman haben dazu in [20] gezeigt, dass die Menge S, welche die Extremalpunkte einer kompakten konvexen Menge K enthält, der konvexen Hülle von K entspricht.

2.2.1 Nutzung

Da konvexe Hüllen und Extremalpunkte viele nützliche Eigenschaften haben, gibt es für sie viele Anwendungsbereiche. Konvexe Hüllen können unter anderem für Bildverarbeitung [18], Kollisionserkennung in der Robotik [21] und Videospielen [23], Clustering [22], Machine learning [26], Voronoi-Diagramme und Delaunay-Triangulationen [3] genutzt werden. Viele dieser Bereiche können wiederum für eine Vielzahl anderer Probleme genutzt werden. Die Anwendungsmöglichkeiten von Voronoi-Diagrammen und Delaunay-Triangulationen werden zum Beispiel in [3] untersucht.

2.2.2 Algorithmen

Aufgrund des großen Anwendungsfeldes von konvexen Hüllen besteht eine große Nachfrage an effizienten Algorithmen zur Berechnung von konvexen Hüllen. Algorithmen mit optimaler asymptotische Laufzeit existieren bereits seit über 50 Jahren für spezielle Fälle (konvexe Hüllen in $\mathbb{R}^2$) [16] und mit dem Algorithmus von Chazelle [13] existiert ein Algorithmus mit optimaler asymptotische Laufzeit für beliebige Dimensionen. Diese Algorithmen sind optimal, da bewiesen wurde, dass die untere Grenze zum Bestimmen einer konvexen Hülle bei $O(nlog(n))$ liegt [36]. Für höhere Dimensionen ist die Laufzeit abhängig von der Anzahl der Facetten auf der konvexen Hülle. Die Anzahl der Facetten F eines konvexen Polytops für Dimensionen $d \geq 4$ kann auf $F = n^{\lfloor \frac{d}{2} \rfloor}$ ansteigen [32]. Wenn jede dieser Facetten in $O(1)$ konstruiert werden kann, ist es leicht zu sehen, dass ein Algorithmus zur Bestimmung von konvexen Hüllen im schlimmsten Fall eine Laufzeit von mindestens $O(F) = O(n^{\lfloor \frac{d}{2} \rfloor})$ haben muss. Algorithmen mit einer Laufzeit von $O(nlog(n) + n^{\lfloor \frac{d}{2} \rfloor})$ sind damit optimal [13]. Obwohl es diese optimalen Algorithmen gibt, werden weiterhin neue Ansätze entwickelt oder verbessert, da sie leichter zu implementieren sind, eine bessere Laufzeit bei realen Datenmengen aufweisen oder durch Techniken wie Parallelisierung die Laufzeit verbessert werden kann [33, 1].

Gift-Wrapping-Algorithmus und Jarvis-March

Der Gift-Wrapping-Algorithmus von Chand und Kapur [12] gehört zu den ersten Algorithmen, die zur Berechnung von konvexen Hüllen entwickelt wurden. Für planare Punkte wird dieser Algorithmus häufig Jarvis-March genannt, da Jarvis [17] einen Algorithmus entwickelt hat, welcher sehr ähnlich zu dem von Chand und Kapur ist. Bei Jarvis-March wird eine Linie um einen Startpunkt rotiert, bis diese auf einen Punkt trifft. Dieser Punkt wird der konvexen Hülle hinzugefügt und der Vorgang für diesen Punkt wiederholt, bis

der Startpunkt wieder gefunden wird. Um einen Startpunkt zu wählen, kann dieser Vorgang für einen Punkt außerhalb der konvexen Hülle ausgeführt und der so gefundene Punkt als Startpunkt genutzt werden, oder es kann ein Punkt gewählt werden, welcher den größten/kleinsten Wert entlang einer der Koordinatenachsen hat. Um den Algorithmus zu beschleunigen, können die Punkte ausgeschlossen werden, welche von der bereits gefundenen konvexen Hülle und einer Linie, die vom Startpunkt zum aktuellen Punkt verläuft, eingeschlossen werden.

Der Gift-Wrapping-Algorithmus kann auch für konvexe Hüllen im $\mathbb{R}^n$ angewandt werden [12, 17, 1]. Die Erweiterung auf für 3 Dimensionen oder mehr ist aber nicht zwangsläufig trivial und kann zu unvollständigen oder fehlerhaften konvexen Hüllen führen, wenn zum Beispiel mehr als 3 komplanare Punkte auf der konvexen Hülle existieren. Ein möglicher Lösungsansatz für dieses Problem wird in [34] behandelt. Mit einer Laufzeit von $O(hn)$, wobei h der Anzahl der Punkte auf der konvexen Hülle entspricht, ist Gift-Wrapping ein effizienter Algorithmus für $h \ll n$, wird aber schnell von $O(nlog(n))$ Algorithmen verdrängt, da die Worst-Case-Laufzeit $O(n^2)$ entspricht. Trotz dieser schlechten Worst-Case-Laufzeit gibt es immer noch aktuelle Entwicklung, um den Algorithmus zu verbessern, da er auch als Grundlage für andere Algorithmen genutzt werden kann [1].

Graham Scan

Der Graham-Scan-Algorithmus ist einer der ersten Algorithmen, welche die konvexe Hülle in optimaler Zeit von $O(nlog(n))$ berechnen kann und wurde von Graham[16] entwickelt. Um die konvexe Hülle zu berechnen, wird bei diesem Algorithmus zuerst ein Punkt P gewählt, welcher innerhalb der konvexen Hülle liegt. Dann werden alle Punkte als Polarkoordinaten um P dargestellt und aufsteigend nach ihrem Winkel sortiert. Die Richtung, in welcher der Winkel $\theta = 0$ ist, kann am Anfang beliebig festgelegt werden. Bei den so sortierten Punkten können nun immer drei aufeinander folgende Punkte, P_k, P_{k+1} und P_{k+2}, miteinander verglichen werden. Bei diesem Vergleich kann P_{k+1} entweder als Extremalpunkt ausgeschlossen werden, wird gelöscht und der Vorgang wird für P_{k-1}, P_k und P_{k+1} wiederholt, oder P_{k+1} wird nicht ausgeschlossen und der Vorgang für P_{k+1}, P_{k+2} und P_{k+3} wiederholt. Dieser Vorgang wird so lange wiederholt, bis alle Punkte ausgeschlossen sind, welche keine Extremalpunkte sind. Die übrigen Punkte bilden dann die konvexe Hülle. Dieser Algorithmus kann nur für Punkte im $\mathbb{R}^2$ angewendet werden.

Quickhull

Der Quickhull-Algorithmus besteht zum Großteil aus einer rekursiven Funktion und wird von Preparata und Shamos[29], für Punkte im $\mathbb{R}^2$, folgendermaßen beschrieben.

Die Funktion bekommt als Eingabe eine Menge S_0 und die Punkte p und q, welche Teil von S_0 sind. Die Punkte p und q bilden eine gerichtete Linie L, welche von p nach q verläuft und für S_0 gilt, dass alle Punkte von S_0 links von L oder auf L liegen. Wenn S_0 nur p und q enthält, wird die Kante (p, q) zurückgegeben. Ansonsten wird der Punkt h gesucht, welcher die größte Entfernung zu L hat. Dann werden die gerichteten Linien L_1, von p nach h, und L_2, von h nach q, gebildet. Mit diesen Linien werden zwei neue

Mengen S_1 uns S_2 gebildet. Alle Punkte, welche links von L_1 oder auf L_1 liegen, werden S_1 zugewiesen und S_2 wird auf die gleiche Weise mit der Hilfe von L_2 gebildet. Die Funktion wird dann für (S_1, p, h) und (S_2, h, q) ausgeführt und die Ergebnisse konkateniert. Dabei wird h aus dem zweiten Ergebnis entfernt, damit der Punkt nicht doppelt in der Liste vorkommt. Um den Algorithmus zu starten, kann der Punkt p_0 einer Menge S gesucht werden, welcher den kleinsten/größten Wert entlang der x-Achse hat. Die rekursive Funktion wird dann für (S, p_0, p_0') ausgeführt. Für p_0' gilt $p_0' = p_0 + (0, \epsilon)$ und p_0' wird am Ende aus dem Ergebnis entfernt, weil es kein Punkt von S war. Die so berechnete Menge entspricht dann der konvexen Hülle.

Mit diesem Prozess kann der Quickhull-Algorithmus die konvexe Hülle in der Regel in $O(nlog(n))$ berechnen, hat aber eine Worst-Case-Laufzeit von $O(n^2)$. Obwohl der Quickhull-Algorithmus damit keine optimale Worst-Case-Laufzeit hat, wird er trotzdem, zum Beispiel in der CGAL Bibliothek [35], benutzt. Der Quickhull-Algorithmus wurde in [4] auf beliebige Dimensionen erweitert und die dort vorgestellte Variation ist robust und es wurde empirisch gezeigt, dass der Algorithmus eine gute Laufzeit hat.

Divide and Conquer

Preparata und Hong [28] haben einen Algorithmus zur Berechnung der konvexen Hülle beschrieben, welcher auf dem Divide-and-Conquer-Prinzip basiert und mit einer Laufzeit von $O(nlog(n))$ optimal ist. Dieser Algorithmus kann auch für mehr als 3 Dimensionen benutzt werden.

Bevor die konvexe Hülle berechnet wird, werden die Punkte zuerst nach aufsteigender x-Koordinate sortiert. Um die konvexe Hülle zu berechnen, werden die Punkte dann in zwei möglichst gleich große Mengen aufgeteilt und die konvexen Hüllen beider Mengen berechnet. Zur Berechnung der konvexen Hüllen wird dieser Schritt rekursiv ausgeführt, bis die Berechnung der konvexe Hülle trivial ist. Diese konvexen Hüllen werden dann in linearer Zeit miteinander kombiniert, um die gemeinsame konvexe Hülle zu berechnen. Auf diese Weise werden die Rekursionen aufgelöst und am Ende bleibt die konvexe Hülle aller Punkte übrig.

Kirkpatrick-Seidel und Chan's Algorithmus

Obwohl Algorithmen mit einer Laufzeit von $O(nlog(n))$ bereits eine optimale asymptotische Laufzeit haben, können Algorithmen eine bessere Laufzeit haben, wenn die Laufzeit abhängig von der Größe der konvexen Hülle ist und die konvexe Hülle entsprechend klein ist. Die $O(nh)$ Laufzeit des Gift-Wrapping-Algorithmus [17] ist theoretisch besser als $O(nlog(n))$ wenn $h < log(n)$.

Kirkpatrick und Seidel [19] nutzen diesen Umstand, um einen Algorithmus für planare Punkte zu entwickeln, welcher eine Laufzeit von $O(nlog(h))$ hat und damit für $h = n$ immer noch optimal ist, aber für kleine h eine bessere Laufzeit hat. Der von Kirkpatrick und Seidel entwickelte Algorithmus basiert auf einer abgewandelten Form des Divide-and-Conquer-Prinzips, das sie in ihrer Arbeit *marriage-before-conquest* nennen. Dabei verringern sie die Größe der Teilprobleme, indem sie Punkte eliminieren, welche beim

Kombinieren der Lösungen redundant sind, bevor die Teilprobleme gelöst werden. Auf diese Weise erreichen sie eine bessere Laufzeit.

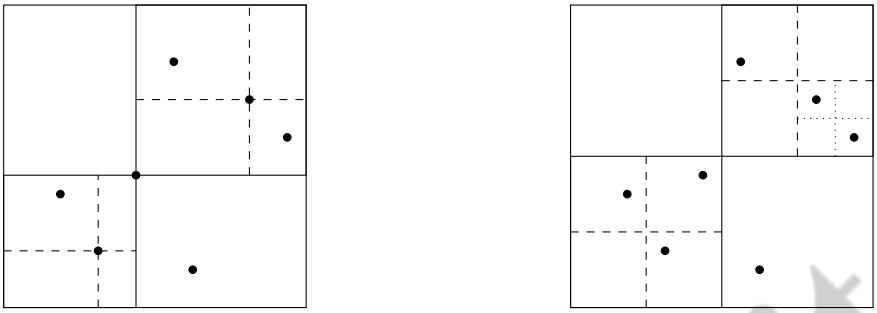
Chan et al. [11] haben diesen Algorithmus zuerst etwas vereinfacht. Später hat Chan [10] auf einer neuen Grundlage zwei neue Algorithmen entwickelt, um effizient konvexe Hüllen im $\mathbb{R}^2$ und $\mathbb{R}^3$ zu berechnen. Bei den von Chan entwickelten Algorithmen werden die Punkte zuerst in Mengen mit m vielen Punkten aufgeteilt. Für diese Mengen wird dann jeweils die konvexe Hülle mit einem optimalen $O(mlog(m))$ Algorithmus berechnet. Um aus diesen konvexen Hüllen die finale konvexe Hülle zu berechnen, wird ein Gift-Wrapping-Algorithmus benutzt. Mit dieser Vorgehensweise wird eine Laufzeit von $O(n(1+\frac{h}{m})log(m))$ erreicht, welche zu $O(nlog(h))$ wird, wenn $m = h$ ist. Um diese Laufzeit zu erreichen, benutzt Chan einen Algorithmus, um h möglichst gut abzuschätzen.

Obwohl die Algorithmen von Chan und der Algorithmus von Kirkpatrick und Seidel eine optimale Laufzeit haben und im besten Fall eine Laufzeit von $O(nlog(h))$ aufweisen, müssen sie nicht unbedingt für reale Probleme geeignet sein. McQueen und Toussaint [25] haben den Algorithmus von Kirkpatrick und Seidel und zwei Variationen miteinander verglichen. Während die Variationen im Vergleich zum Originalalgorithmus gut abschneiden, stellen sie aber auch fest, dass die Laufzeiten im Vergleich zu anderen Arbeiten sehr langsam sind. Sie vergleichen dafür ihre Laufzeiten mit Daten aus [8], geben aber auch an, dass die Laufzeiten nicht zwangsläufig miteinander vergleichbar sind, da die Daten auf unterschiedlichen Systemen ermittelt wurden. Barton [5] hat den Algorithmus von Chan et al. in 4 verschiedenen Versionen implementiert und sie für Daten in $\mathbb{R}^2$ mit Graham Scan und dem Gift Warpping Algorithmus von Jarvis verglichen. Die Algorithmen wurden nur für $n \leq 1000$ miteinander verglichen und lassen damit keine Schlüsse auf das Verhalten der Algorithmen für große Datenmengen zu. In diesen Tests waren die Algorithmen von Chan et al. besser als Jarvis-March, waren aber deutlich langsamer als Graham Scan und die Laufzeiten sind schneller gewachsen als die von Grahams Scan.

2.3 Räumliche Datenstrukturen

Datenstrukturen sind ein grundlegendes Konzept in der Informatik. Sie werden genutzt, um Daten in einem organisierten Muster zu speichern, damit effizient darauf zugegriffen werden kann und bestimmte Operationen oder Anfragen schnell darauf ausgeführt werden können. Wenn Daten als Punkte in einem Raum, häufig einem euklidischen Raum, darstellbar sind, kann dieser Raum in verschiedene Regionen aufgeteilt werden, sodass jeder Punkt genau einer Region zugeteilt werden kann. Eine Datenstruktur, bei der diese Regionen nach einem bestimmten System erstellt werden und die Punkte in die Regionen einsortiert sind, kann dann zum Beispiel genutzt werden, um schnell einen bestimmten Punkt zu finden oder um andere Suchanfragen zu beschleunigen, indem uninteressante Regionen ausgeschlossen werden.

In dieser Arbeit werden Quadtrees, Octrees und k-d-Bäume benutzt. Diese Datenstrukturen können auf verschiedene Weisen implementiert werden. Um diese Versionen voneinander abzugrenzen, werden hier die Namen aus [31] benutzt.



(a) Point-Quadtree (b) PR-Quadtree

Abbildung 2.1: Beispiele für Quadtrees. Die Regionen werden aufgeteilt, bis in den Kindern nur noch ein Punkt enthalten ist.

2.3.1 Quadtrees und Octrees

Ein Quadtree ist eine hierarchische Datenstruktur, bei der 2-dimensionale Punkte in 4 Regionen aufgeteilt werden. Die Region wird dabei parallel zur x- und y-Achse aufgeteilt. Diese Regionen werden dann rekursiv in 4 Regionen aufgeteilt, bis die Regionen nur noch eine bestimmte Menge an Punkten enthalten. Auf diese Weise entsteht ein Baum, in dem jeder Konten bis zu 4 Kinder hat. Hat ein Knoten keine Kinder, ist es ein Blattknoten. Dabei gibt es verschiedene Möglichkeiten, wie die Regionen aufgeteilt werden können. Von diesen Varianten werden in dieser Arbeit Point-Quadtrees und PR-Quadtrees genutzt.

Point-Quadtree

Point-Quadtrees wurden von Finkel und Bentley[15] entwickelt. Bei diesem Quadtree wird in jedem Konten ein Punkt gespeichert und die Region, in welcher der Punkt liegt, wird entlang dieses Punktes in 4 neue Regionen aufgeteilt. Die anderen Punkte aus dieser Region werden dann auf die neuen Regionen aufgeteilt und der Vorgang wiederholt. Siehe Abbildung 2.1.

PR-Quadtree

Samet beschreibt in [31] eine andere Variation, welche Point-Region-Quadtree oder PR-Quadtree genannt wird. Bei diesem Quadtree wird eine Region immer in 4 gleich große Regionen aufgeteilt. Dafür wird der Mittelpunkt der Region gesucht und entlang dieses Punktes aufgeteilt. Dieser Punkt ist nicht Teil der Punkte, welche im Quadtree gespeichert werden, und dementsprechend werden in diesem Quadtree die Punkte nur in den Blättern gespeichert. Da die Regionen nicht entlang von einem Punkt aus dem Baum aufgeteilt werden, kann der Baum sehr tief werden, wenn das Aufteilen der Regionen nicht dazu führt, dass die Punkte in der Region aufgeteilt werden. Siehe Abbildung 2.1.

Octree

Der Octree ist eine Erweiterung des Quadtrees auf 3 Dimensionen. Dabei wird eine Region nicht in 4, sondern in 8 Regionen aufgeteilt. Yau und Srihari haben diese Datenstruktur dann in [37] auf höhere Dimensionen erweitert und so den Hyperoctree für beliebige Dimensionen entwickelt.

2.3.2 k-d-Bäume

Ein k-d-Baum ist eine hierarchische Datenstruktur, welche von Bentley [6] einwickelt wurde und für k-dimensionale Punkte verwendet werden kann. Bei dieser Datenstruktur werden die Regionen rekursiv in 2 Regionen aufgeteilt. Bei diesem k-d-Baum werden die Regionen immer abwechselnd entlang der Koordinatenachsen aufgeteilt. Dafür wird, aus den Punkten der Region, der Median entlang dieser Achse gesucht. Die Regionen werden dann so aufgeteilt, dass eine Region alle Punkte enthält, deren Wert entlang der Achse größer oder gleich dem Wert des Medians ist, und die andere Region die restlichen Punkte enthält.

Eine Abwandlung dieses Baums ist k-d-Trie von Orenstein [27]. Dieser wird von Samet auch PR-k-d-Baum genannt, da hier, genau wie bei PR-Quadtrees, die Region nicht entlang von einem Punkt aufgeteilt wird, sondern entlang von einem Mittelwert.

Beispiele für beide Varianten sind in Abbildung 2.2 dargestellt.

(a) k-d-Baum (b) PR-k-d-Baum / k-d-Trie

Abbildung 2.2: Beispiele für k-d-Bäume. Die Regionen werden aufgeteilt, bis in den Kindern nur noch ein Punkt enthalten ist.

2.4 CGAL - *The Computational Geometry Algorithms Library*

Die *Computational Geometry Algorithms Library*(CGAL) ist eine Open-Source-Bibliothek in C++. Und wird auf ihrer Startseite wie folgt beschrieben:

> CGAL ist ein Softwareprojekt, das einen einfachen Zugang zu effizienten und zuverlässigen geometrischen Algorithmen in Form einer C++-Bibliothek bie-

> tet. CGAL wird in verschiedenen Bereichen eingesetzt, in denen geometrische Berechnungen erforderlich sind, z. B. in geografischen Informationssystemen, computergestütztem Design, Molekularbiologie, medizinischer Bildgebung, Computergrafik und Robotik. [35]

Die Bibliothek implementiert Datentypen und Funktionen zum Arbeiten mit Punkten, Vektoren und geometrischen Objekten und viele der bereits besprochenen Konzepte. Zur Berechnung von konvexen Hüllen in zwei Dimensionen bietet CGAL viele verschiedene Algorithmen an, darunter auch der Gift-Wrapping-Algorithmus von Jarvis und eine modifizierte Version des Graham-Scans. Für drei Dimensionen existiert eine Implementierung von Quickhull.

Quadtrees und Octrees existieren als spezialisierte Form von Orthtrees, eine Generalisierung, um das Konzept von Quad- und Octrees auf beliebige Dimensionen zu übertragen. Die Implementierung entspricht den vonSamet beschriebenen PR-Quadtrees und teilt die Ebenen des Baums in gleich große Sektionen auf.

K-d-Bäume sind so implementiert, dass bestimmt werden kann, wie die Daten im k-d-Baum aufgeteilt werden. Damit lassen sie sich sowohl als normale k-d-Bäume als auch als PR-k-d-Bäume benutzen.

3 Algorithmus und Implementierung

In diesem Kapitel wird zuerst ein Algorithmus beschrieben, welcher räumliche Datenstrukturen nutzt, um effizient die Extremalpunkte zu bestimmen, welche gefunden werden, wenn eine Linie oder Ebene um eine Achse rotiert wird. Der Algorithmus wird für Punkte im $\mathbb{R}^2$ beschrieben, da er sich für diesen Fall leichter verbildlichen lässt, kann aber ohne viele Änderungen für Punkte im $\mathbb{R}^{\not\vdash}$ angewendet werden. Von diesem Algorithmus wird dann die theoretische Laufzeit untersucht und erklärt, wie er benutzt werden kann, um konvexe Hüllen mithilfe von Gift Wrapping schneller zu berechnen.

Anschließend wurden verschiedene Versionen des Algorithmus für Punkte im $\mathbb{R}^3$ implementiert. Für die Implementierung wurden verschiedene Datenstrukturen benutzt und Algorithmen zum Finden von Extremalpunkten, sowie zur Berechnung von konvexen Hüllen implementiert.

3.1 Benachbarten Extremalpunkt finden

P sei eine Menge von n vielen Punkten im $\mathbb{R}^2$, p_e ein Punkt und l eine Linie durch p_e, sodass alle Punkte von P auf einer Seite der Linie liegen. Um die Frage zu beantworten, welchen Punkt p die Linie l als Erstes trifft, wenn l um p_e rotiert wird, kann ein simpler Algorithmus entwickelt werden. Der Vektor v ist ein Vektor, der in p_e anfängt und parallel zu l ist. Aus allen Punkten von P wird dann der Punkt p gesucht, welcher den Winkel zwischen v und dem Vektor u, der von p_e nach p geht, minimiert. Dieser Algorithmus braucht $O(n)$ viele Schritte, da der Winkel für jeden Punkt aus P berechnet wird. Es ist leicht zu sehen, dass diese Frage ohne zusätzliche Informationen nicht schneller als $O(n)$ beantwortet werden kann, weil es sonst impliziert, dass nicht alle Punkte betrachtet werden. Mit den Eigenschaften von konvexen Hüllen kann diese Frage auch so interpretiert werden, dass ein Extremalpunkt p aus der konvexen Hülle von P gesucht wird, welcher als Erstes von l getroffen wird. Wenn p_e Teil der konvexen Hülle von P ist, entspricht dieses Problem der Suche nach einem benachbarten Extremalpunkt von p_e und wird als Teilproblem des Gift-Wrapping-Algorithmus gelöst.

Um diese Frage effizienter zu beantworten, werden daher zusätzliche Informationen über die Punkte aus P benötigt. Das Speichern der Punkte in einer Datenstruktur kann diese Informationen bereitstellen. Die Datenstruktur, welche von Edelsbrunner und Maurer [14] genutzt wird, eignet sich sehr gut, um dieses Problem effizient zu lösen. Um diese Datenstruktur zu bauen, muss aber zuerst die konvexe Hülle berechnet werden. Dieser Ansatz eignet sich also nicht, wenn die Extremalpunkte gesucht werden, um die konvexe Hülle zu berechnen, da die konvexe Hülle dafür schon bekannt sein muss. Um nicht von der Berechnung der konvexen Hülle abhängig zu sein, können räumliche Datenstruktu-

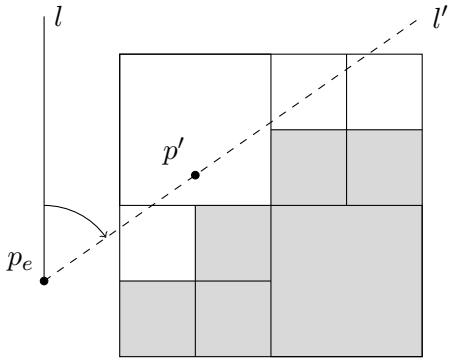


Abbildung 3.1: Regionen die ausgeschlossen werden können, wenn l um p_e rotiert wird, am Beispiel eines PR-Quadtrees. Wenn Kandidat p' gefunden wurde können die schraffierten Regionen ausgeschlossen werden.

ren, wie Quadtrees oder k-d-Bäume, genutzt werden. Diese Datenstrukturen sind weit verbreitet und werden bereits häufig genutzt, um viele Punkte zu speichern. Da diese Datenstrukturen die Punkte in bekannte Regionen aufteilen, können diese Informationen genutzt werden, um herauszufinden, welche Regionen als erstes von l passiert werden. Die Punkte können damit teilweise nach ihrem Winkel um p_e sortiert werden und interessante Punkte so zuerst überprüft werden. Zusätzlich können Regionen ausgeschlossen werden, wenn bereits ein Kandidat p' gefunden wurde, welcher besser ist als alle Punkte, die in diesen Regionen existieren könnten. Ein Beispiel dafür ist in Abbildung 3.1 dargestellt.

Auf der Seite 17 wird der Algorithmus findAdjacentExtremePoint beschrieben, welcher auf dieser Grundlage basiert. Um den nächsten Extremalpunkt auf der konvexen Hülle zu finden, wird dort, ähnlich wie bei Gift-Wrapping, ein Punkt $p \in P$ gesucht, welcher die Funktion $\theta_{p_e,p_c}(p)$ maximiert.

$$\theta_{p_e,p_c}(p) = atan2(det(p - p_e, p_c - p_e), dot(p - p_e, p_c - p_e)) \tag{3.1}$$

Die Funktion berechnet den gerichteten Winkel zwischen p und p_c um den Punkt p_e. In findAdjacentExtremePoint ist p_e ein Extremalpunkt von P, für den ein benachbarter Extremalpunkt gesucht wird, und p_c ist ein Punkt innerhalb der konvexen Hülle von P. In Abbildung 3.2 ein Beispiel dargestellt, wie sich die Funktion für verschiedene Punkte p_t und p'_t verhält.

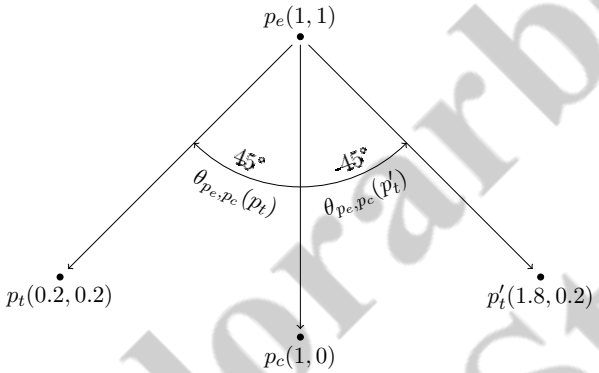


Abbildung 3.2: Ein Beispiel dafür, wie $\theta_{p_e,p_c}(p)$ geometrisch aussieht.

Theorem 3.1.1. *Der Punkt $p \in P$, welcher die Funktion $\theta_{p_e,p_c}(p)$ maximiert, ist ein Extremalpunkt von P.*

Beweis. Angenommen p ist kein Extremalpunkt von P, maximiert aber die Funktion $\theta_{p_e,p_c}(p)$. Daraus folgt, dass ein Extremalpunkt $p' \in P$ existiert, sodass die Linie l, welche durch p und p_e geht, den Punkt p' von den restlichen Punkten in P trennt. Dann gibt es zwei Fälle, welchen Wert $\theta_{p_e,p_c}(p')$ haben kann:

1. $\theta_{p_e,p_c}(p') > \theta_{p_e,p_c}(p)$
2. $\theta_{p_e,p_c}(p') < \theta_{p_e,p_c}(p) - 180°$

Warum der erste Fall nicht eintreten kann, ist leicht zu sehen. Wenn $\theta_{p_e,p_c}(p') > \theta_{p_e,p_c}(p)$ wäre, dann maximiert p nicht die Funktion $\theta_{p_e,p_c}(p)$ für alle $p \in P$ und es entsteht ein Widerspruch. Damit könnte nur noch der zweite Fall eintreten. In diesem Fall bilden die Punkte p_e, p_c, p und p' ein Viereck, welches an der Ecke von p_e einen Innenwinkel $|\theta_{p_e,p_c}(p')| + |\theta_{p_e,p_c}(p)| > 180°$ hat. Der Punkt p_e kann dann als Konvexkombination von p_c, p und p' dargestellt werden und wäre damit kein Extremalpunkt von P.

Da die Existenz von p' in beiden Fällen zu einem Widerspruch führt, kann p' nicht existieren und p ist damit ein Extremalpunkt von P. □

Ein wichtiger Teil des Algorithmus besteht darin, die Reihenfolge zu bestimmen, in welcher die Regionen der Datenstruktur abgearbeitet werden. Dafür werden die Ecken der Region als potenzielle Kandidaten für den gesuchten Extremalpunkt betrachtet und bestimmt, welche Ecke c die Funktion $\theta_{p_e,p_c}(c)$ maximiert. Anhand dieser Information können die Kinder dieser Region teilweise danach sortiert werden, in welcher Reihenfolge sie von der Linie l passiert werden. Wenn bereits ein Kandidat p_b gefunden wurde, sodass alle Punkte in der Region einen kleineren Wert haben als $\theta_{p_e,p_c}(p_b)$, kann die gesamte Region ignoriert werden.

Für diese beiden Schritte wird die Funktion FINDBESTCORNER benutzt. Dort wird zuerst die Ecke c_{max} mit dem größten Wert und die Ecke c_{min} mit dem kleinsten Wert für θ_{p_e,p_c} bestimmt. Wenn $\theta_{p_e,p_c}(c_{max})$ einen größeren Wert hat als $\theta_{p_e,p_c}(p_b)$, muss dieser Teil der Region noch untersucht werden und wird als erstes von l geschnitten. Die Funktion gibt dann den Index von c_{max} zurück. Ist dies nicht der Fall, gibt es immer noch die Möglichkeit, dass alle Ecken einen kleineren Wert als $\theta_{p_e,p_c}(p_b)$ haben, die Region aber trotzdem so liegt, dass sie einen besseren Kandidaten als p_b enthalten könnte. In diesem Fall müssen $\theta_{p_e,p_c}(c_{max})$ und $\theta_{p_e,p_c}(c_{min})$ ein unterschiedliches Vorzeichen haben und die Summe $|\theta_{p_e,p_c}(c_{max})| + |\theta_{p_e,p_c}(c_{min})|$ muss größer sein als 180°. In diesem Fall wird dann der Index von c_{min} zurückgegeben. Die Region mit dieser Ecke liegt entweder so, dass alle Punkte darin kleiner sind als $\theta_{p_e,p_c}(p_b)$, und kann dann als nächstes ausgeschlossen werden, oder sie enthält einen Bereich zwischen, in dem Punkte liegen können mit $\theta_{p_e,p_c}(p_b) <= \theta_{p_e,p_c}(p) <= 180°$. Ansonsten wird -1 als Index zurückgegeben, um zu signalisieren, dass diese Region ignoriert werden kann.

Wie die Kinder der Region sortiert werden, ist dann abhängig von der Datenstruktur. Bei dem in FINDADJACENTEXTREMEPOINT verwendeten Quadtree wird dann zuerst der Quadrant untersucht, welcher zu dem von FINDBESTCORNER gefundenen Index gehört. Anschließend werden dann die zu diesem Quadranten angrenzenden Quadranten untersucht und der gegenüberliegende Quadrant wird als Letztes untersucht. Dafür werden sie in umgekehrter Reihenfolge auf einen Stack gelegt. Der Vorgang wird dann für das oberste Element wiederholt. Auf diese Weise werden die Regionen und Punkte in einer Reihenfolge untersucht, in der nicht immer alle Punkte untersucht werden müssen und ein gewisser Teil der Punkte ausgeschlossen werden kann.

Dieser Algorithmus sollte auf eine Vielzahl von räumlichen Datenstrukturen anwendbar sein, solange die Bounding Box einer Region effizient berechnet werden kann. Dafür muss nur TREE durch eine andere Datenstruktur ersetzt und die Zeile 13-15 angepasst werden, um die Teilregionen der neuen Datenstruktur sinnvoll zu sortieren. Auf diese Weise kann der Algorithmus FINDADJACENTEXTREMEPOINT für die in 2.3 vorgestellten Datenstrukturen genutzt werden.

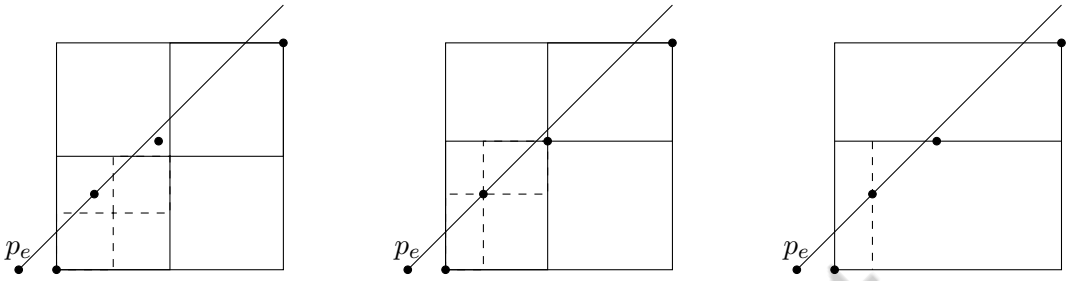
Algorithm 1 findAdjacentExtremePoint

Input:
$Tree :=$ QUADTREE(P)
$p_e :=$ ein Extremalpunkt von P
$p_c :=$ ein Punkt in der konvexen Hülle von P(z.b. der Mittelpunkt von P)
Output:
Punkt $p \in P$ der $\theta_{p_e,p_c}(p)$ maximiert

```
1:  procedure FINDADJACENTEXTREMEPOINT(Tree, p_e, p_c)
2:      Stack.push(Tree.root())
3:      p_b ← p_c
4:      while Stack not empty do
5:          p ← Stack.pop()
6:          if p is leaf then
7:              if θ_{p_e,p_c}(p) > θ_{p_e,p_c}(p_b) then
8:                  p_b ← p
9:              end if
10:         else
11:             index ← FINDBESTCORNER(p.bbox(), p_b, θ_{p_e,p_c})   ▷ Siehe Algorithmus 2
12:             if index >= 0 then
13:                 Stack.push(p.child(opposite to index))
14:                 Stack.push(p.child(adjacent to index))
15:                 Stack.push(p.child(index))
16:             end if
17:         end if
18:     end while
19:     return p_b
20: end procedure
```

Algorithm 2 findBestCorner

```
1:  procedure FINDBESTCORNER(bbox, p_b, p_e, p_c)
2:      min ← corner c from bbox that minimizes θ_{p_e,p_c}(c)
3:      max ← corner c from bbox that maximizes θ_{p_e,p_c}(c)
4:      if θ_{p_e,p_c}(max) > θ_{p_e,p_c}(p_b) then
5:          return index of max
6:      else if θ_{p_e,p_c}(max) and θ_{p_e,p_c}(min) have different sign then
7:          if |θ_{p_e,p_c}(max)| + |θ_{p_e,p_c}(min)| > 180° then
8:              return index of min
9:          end if
10:     end if
11:     return -1
12: end procedure
```



(a) PR-Quadtree (b) Point-Quadtree (c) k-d-Baum

Abbildung 3.3: Beispiele, in denen der Algorithmus alle Punkte untersuchen muss. Bei (b) und (c) ist es Abhängig davon, ob die Punkte in den Knoten oder Blättern gespeichert werden.

3.1.1 Laufzeit

Um die Zeitkomplexität des Algorithmus zu untersuchen, wird die von Bentley et al. [7] entwickelte Methode benutzt, mit der die Laufzeit von Divide-and-conquer bestimmt werden kann. Diese Methode wird oft als *Master Theorem* oder *master theorem for divide-and-conquer recurrences* bezeichnet. Dafür wird davon ausgegangen, dass die Teilprobleme rekursiv gelöst und dann kombiniert werden. Obwohl der Algorithmus FINDADJACENTEXTREMEPOINT nicht rekursiv definiert ist, kann er zu einem solchen Algorithmus umgewandelt werden, um die Laufzeit zu untersuchen.

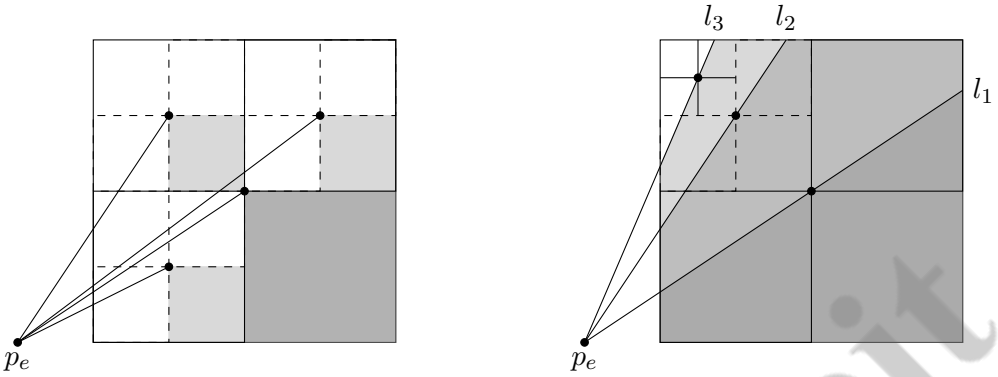
Eine untere Schranke für die Worst-Case-Laufzeit lässt sich leicht anhand eines Beispiels bestimmen. In Abbildung 3.3 ist ein Beispiel für verschiedene Datenstrukturen dargestellt, bei denen alle Regionen, welche Punkte enthalten, untersucht werden müssen und damit auch alle Punkte, die in der Datenstruktur gespeichert sind. Da alle Punkte in mindestens konstanter Zeit untersucht werden, ergibt sich damit eine Laufzeit von mindestens $O(n)$. Dass diese Laufzeit auch die obere Grenze für die Worst-Case-Laufzeit ist, kann mit dem *Master Theorem* gezeigt werden. Dafür wird in jeder der 4 Teilregionen rekursiv nach dem besten Punkt gesucht und die Ergebnisse miteinander verglichen. Das Aufteilen und Kombinieren der Ergebnisse kann dabei in konstanter Zeit durchgeführt werden. Die Laufzeit des Problems kann dann in der folgenden Weise dargestellt werden:

$$\begin{aligned} T(1) &= 1, \\ T(n) &= 4T(n/4) + O(1) \end{aligned}$$

Damit das *Master Theorem* darauf angewandt werden kann, wird es zu

$$T(n) = 4^p T(n/4) + n^p g(n)$$

umgeformt. Dabei sind $p = log_4(4) = 1$, $g(n) = O(n^{-1})$ und $\tilde{g}(n) = O(1)$. Jetzt kann das



(a) Abschätzung anhand der Teilregionen (b) Abschätzung anhand der Fläche

Abbildung 3.4: Die schattierten Regionen müssen nicht mehr untersucht werden, wenn die eingezeichneten Punkte gefunden wurden. Damit lässt sich der Suchraum auf die nicht schattierten Regionen begrenzen.

Master Theorem darauf angewandt werden und daraus folgt:

$$\begin{aligned} T(n) &= n^p[T(1) + (\tilde{g}(n))] \\ &= n^p[T(1) + O(1)] \\ &= O(n^p) = O(n^1) \end{aligned}$$

Nach dem *Master Theorem* ist damit die obere Schranke das gleiche wie die untere Schranke für die Worst-Case-Laufzeit, und die Worst-Case-Laufzeit ist damit $O(n)$. Für diese Laufzeit ist es egal, ob p_e innerhalb oder außerhalb des Baumes liegt, denn es kann ein nahezu identisches Beispiel konstruiert werden, wenn der Punkt in der unteren linken Ecke als p_e gewählt wird.

Der Algorithmus muss aber nicht immer diese Laufzeit haben. In den Abbildungen 3.1 und 3.4 werden Beispiele gezeigt, in denen die Menge der zu untersuchenden Punkte erheblich verringert werden kann. Die Beispiele in Abbildung 3.3 können leicht zu einem Fall abgeändert werden, indem Punkte ausgeschlossen werden können. Aus diesem Grund wäre es interessant, die durchschnittliche Laufzeit des Algorithmus zu bestimmen. Eine gute Abschätzung für die durchschnittliche Laufzeit zu finden, ist aber aus mehreren Gründen schwierig. Zum einen können die unterschiedlichen Datenstrukturen zu einem anderen Verhalten führen, zum anderen kann es für die Laufzeit relevant sein, ob p_e innerhalb oder außerhalb der Datenstruktur liegt. Trotzdem sollen hier zwei mögliche Einschränkungen untersucht werden, um die durchschnittliche Laufzeit einzugrenzen. Für die folgenden Fälle wird angenommen, dass als Datenstruktur ein balancierter Point-Quadtree benutzt wird und der Punkt p_e außerhalb des Baumes liegt.

Der erste Ansatz versucht, die Anzahl der Regionen zu verringern, welche in der Divide-and-conquer-Version des Algorithmus untersucht werden müssen. Dafür wird der Umstand genutzt, dass der Knoten k, entlang dem die Punkte aufgeteilt werden, zu den

Punkten gehört, welche im Point-Quadtree gespeichert werden. Damit ist in jeder Stufe der Rekursion ein potenzieller Kandidat für den Extremalpunkt bekannt. Wenn eine Linie von p_e zu diesem Punkt k gezogen wird, liegt immer mindestens eine der 4 Regionen vollständig unter der Linie, sodass alle Punkte in der Region schlechtere Kandidaten sind als k. Für diese Region muss die Rekursion dann nicht wiederholt werden. In Abbildung 3.4a ist dafür ein Beispiel für 2 Rekursionen abgebildet. Die Laufzeit kann dann als $T(n) = 3T(\frac{n}{4}) + O(1)$ dargestellt werden und durch das *Master Theorem* ergibt sich dann eine Laufzeit von $O(n^{0.7925})$. Dieser Fall kann auch auf 3 Dimensionen erweitert werden. Dort wird auch nur eine der Teilregionen ausgeschlossen und durch Einsetzen folgt dann daraus $T(n) = 7T(\frac{n}{8}) + O(1) \approx O(n^{0.9358})$.

Für den zweiten Ansatz ist eine Darstellung in Abbildung 3.4b gegeben. In der Abbildung ist leicht zu erkennen, dass der Algorithmus, so wie er in FINDADJACENTEXTREMEPOINT beschrieben ist, die zu untersuchende Fläche des Baums um einen Faktor $q \geq 1$ reduziert. Wie groß dieser Faktor genau ist, ist schwer zu bestimmen, da er von der Position der Knoten im Baum und p_e abhängt. In Abbildung 3.4b ist zu erkennen, dass die Fläche in jeder Iteration etwas mehr als halbiert wird. Der Einfachheit halber wird deswegen angenommen, dass $q = 2$ ist. Die Laufzeit kann dann auf zwei Arten abgeschätzt werden. Zum einen kann angenommen werden, dass möglich ist, die Anzahl der Teilprobleme um den Faktor q zu verringern. Damit ergibt sich nach dem *Master Theorem* $T(n) = \frac{4}{q}T(\frac{n}{4}) + O(1) = 2T(\frac{n}{4}) + O(1) = O(\sqrt{n})$. Auch dieser Fall kann auf drei Dimensionen erweitert werden und dann ergibt sich eine Laufzeit von $O(\sqrt[3]{n^2})$.

Diese beiden Ansätze lassen sich dazu verallgemeinern, dass $m \in \{1, ..., 4\}$ Teilregionen des Baums untersucht werden müssen. Mit dem *Master Theorem* ergibt sich damit die Laufzeit $O(n^{\log_4(m)})$, wenn $m \geq 2$ und $O(logn)$, wenn $m = 1$. Analog dazu folgt für drei Dimensionen dann $O(n^{\log_8(m)}), m \in \{1, ..., 8\}$, wenn $m \geq 2$, und $O(logn)$, wenn $m = 1$.

In Kapitel 4 wird dann untersucht, wie groß m für verschiedene Datenmengen ist.

3.1.2 Anwendung für Gift-Wrapping Algorithmus

Dieser Algorithmus kann benutzt werden, um die lineare Suche zu ersetzen, welche bei Gift-Wrapping benutzt wird, um angrenzende Extremalpunkte zu finden. Dafür muss nur zuvor ein Punkt innerhalb der konvexen Hülle bestimmt werden, damit FINDADJACENTEXTREMEPOINT benutzt werden kann. Dieser Punkt kann in $O(n)$ gefunden werden, indem zum Beispiel der Mittelpunkt von allen Punkten gebildet wird. Da der Algorithmus mit einer Laufzeit $O(n)$ im Worst Case die gleiche Laufzeit hat wie eine lineare Suche, bleibt die Laufzeit für diese veränderte Version von Gift-Wrapping gleich. Falls die Punkte nicht gegeben sind, kommt zu dieser Laufzeit noch die Laufzeit zur Konstruktion der Datenstruktur hinzu, welche für viele $O(nlog(n))$ beträgt [6, 15]. Damit kommt der Algorithmus auf eine Laufzeit von $O(nh + nlog(n))$. Diese Laufzeit ist zunächst vermeintlich schlechter als andere Algorithmen zur Berechnung von konvexen Hüllen, aber wenn die durchschnittliche Laufzeit von FINDADJACENTEXTREMEPOINT $O(n^c), c < 1$ oder $O(\sqrt{n})$ erreicht, ist er mit einer Laufzeit von $O(n^{1+c})$ für $h = n$ schneller als der normale Gift-Wrapping Algorithmus und kann die konvexe Hülle für $h \ll n$ in $O(n^c h)$ be-

rechnen, wenn die Datenstruktur bereits vorhanden ist. Wenn sie nicht vorhanden ist, braucht der Algorithmus für $h \ll n$ genauso lange wie andere optimale Algorithmen zur Berechnung von konvexen Hüllen.

3.2 Implementierungsdetails

Auf dieser Grundlage wurden verschieden Algorithmen für Punkte im $\mathbb{R}^3$ implementiert. Diese Algorithmen sind in C++ geschrieben und nutzen die Datenstrukturen für Octrees und k-d-Bäume aus CGAL [35]. Die Octrees werden dabei auf eine maximale Tiefe von 50 limitiert. Bei den k-d-Bäumen werden die Punkte immer entlang der Dimension aufgeteilt, in der sie am weitesten verteilt sind. Dafür wird der Median aus dieser Verteilung gesucht und die Punkte dann entlang dieses Punktes aufgeteilt. Bei beiden Datenstrukturen kann zudem bestimmt werden, ab welcher Menge von Punkten die Blätter nicht mehr aufgeteilt werden müssen.

3.2.1 Extremalpunkte finden

Zum Finden von Extremalpunkten wurden 3 verschiedene Algorithmen implementiert. Zwei davon sind eine Erweiterung von FINDADJACENTEXTREMEPOINT auf 3 Dimensionen. Dabei wird mit $\theta_{p_e,p_c}(p)$ nicht der Winkel zwischen zwei Punkten relativ zu einem dritten Punkt berechnet, sondern der Winkel zwischen 2 Ebenen, wenn sie um dieselbe Achse rotiert werden. Sowohl im $\mathbb{R}^2$, als auch im $\mathbb{R}^3$ ist das Berechnen von $\theta_{p_e,p_c}(p)$ abhängig von $atan2(y,x)$. Weil die Berechnung von $atan2(y,x)$ sehr zeitintensiv ist, wurde sie durch die Funktion $DiamondAngle(y,x)$ ersetzt, welche den Winkel in Manhattan-Metrik berechnet.

```
double DiamondAngle(double y, double x){
    double res;
    if (y >= 0)
        res = (x >= 0 ? y/(x+y) : 1-x/(-x+y));
    else
        res = (x < 0 ? -2-y/(-x-y) : -1+x/(x-y));
    return (isnan(res)) ? 0 : res;
}
```

Da die Algorithmen nicht die Differenz zwischen den Winkeln benötigen, sondern nur vergleichen, welcher Winkel größer oder kleiner ist, macht der Wechsel zu diesen Winkeln keinen Unterschied, auch wenn die Winkel nicht gleichmäßig verteilt sind. Die berechneten Winkel liegen dabei zwischen -2 und 2.

Bei den 3 Algorithmen handelt es sich um:

LINEARSEARCH Bei diesem Algorithmus wird der Extremalpunkt mithilfe von linearer Suche bestimmt. Dafür wird, ausgehend von einer bekannten Seite F der konvexen Hülle, ein Referenzvektor bestimmt. Dieser Vektor ist orthogonal zu F und orthogonal zu der Kante, von der aus der nächste Extremalpunkt bestimmt werden soll. Der Vektor zeigt

dabei zur Außenseite von F. Mit linearer Suche wird dann der Punkt bestimmt, welcher den Winkel um die Kante zu diesem Vektor minimiert. Wie dies Funktion genau aussieht, wird in [34, 12, 29] beschrieben.

OctreeExtremal Dieser Algorithmus ist eine Implementierung von findAdjacentExtremePoint. Als Datenstruktur wird dafür der Octree aus CGAL benutzt. Ähnlich wie beim Quadtree wird hier zuerst das Kind mit der besten Ecke untersucht, dann dessen Nachbarn, dann deren Nachbarn und zum Schluss das Kind mit der gegenüberliegenden Ecke.

KDExtremal Dieser Algorithmus ist eine Implementierung von findAdjacentExtremePoint. Als Datenstruktur wird dafür der k-d-Baum aus CGAL benutzt. Da es in jedem Knoten im Baum immer nur zwei Kinder gibt, ist die Reihenfolge sehr einfach. Zuerst wird das Kind mit der besten Ecke untersucht und dann das Andere. Da der k-d-Baum die Punkte immer nur in 2 statt 8 Mengen aufteilt, existieren in diesem Baum mehr innere Knoten als im Octree und findBestCorner muss öfter ausgeführt werden, was die Laufzeit verschlechtert.

3.2.2 Konvexe Hüllen

Wie in 3.1.2 beschrieben, können die Algorithmen zum Finden von Extremalpunkten genutzt werden, um einen Gift Wrapping Algorithmus zur Berechnung von konvexen Hüllen zu entwickeln. Dafür wurde ein simpler Gift Wrapping Algorithmus entwickelt, wie er in [29] beschrieben wird. Diese Implementierung kann, wenn es mehr als drei komplanare Punkte auf der konvexen Hülle gibt, fehlerhafte oder unvollständige konvexe Hüllen berechnen. Die Algorithmen für die konvexe Hülle unterscheiden sich nur darin, welcher Algorithmus zum Finden der Extremalpunkte genutzt wurde. OctreeWrap benutzt den Algorithmus OctreeExtremal, KDWrap benutzt den Algorithmus KDExtremal und linearSearch wird benutzt, um den normalen GiftWrapping Algorithmus zu erhalten.

4 Analyse und Vergleich

Die verschiedenen Algorithmen werden alle für Datensätze in 3D miteinander verglichen. Dafür wurden sie auf einem Computer mit Windows 10 64-Bit, einer AMD Ryzen 5 3600 CPU und 32 GB RAM ausgeführt. Dafür wurden verschiedene Datensätze genutzt. Die 3D-Objekte aus der Tabelle 4.5 wurden von GitHub[1] bezogen. Für die anderen Tabellen wurden die Punkte zufällig generiert. Dabei wurden die Punkte größtenteils uniform auf der Oberfläche einer Kugel, beziehungsweise im Volumen einer Kugel, verteilt. Die Implementierungen für die Algorithmen und die Generierung der Punkte sind auf https://github.com/FabianStroschke/finding_extremal_points_in_3D verfügbar.

4.1 Einen Extremalpunkt finden

In den Tabellen 4.1 und 4.2 werden die Algorithmen aus Kapitel 3.2.1 miteinander verglichen. Diese Tabellen wurden in Abbildung 4.1 und 4.2 als Diagramme dargestellt. Dafür wurden zufällig Punkte in einer Kugel und auf einer Kugel generiert. Für diese Punkte wurde dann die konvexe Hülle berechnet. Ausgehend von dieser konvexen Hülle wurden dann bis zu 1000 Seiten ausgewählt und für diese Seiten die benachbarten Extremalpunkte bestimmt. Die in den Tabellen gemessenen Werte entsprechen dem Mittelwert aus diesen Messungen. Die Zeit zum Bauen der Datenstruktur ist für diese Tabellen nicht mit aufgeführt. Bei den Algorithmen KDExtremal und OctreeExtremal wurden zudem die Datenstrukturen variiert. Dabei wurden die Datenstrukturen einmal mit maximal 2 Punkten pro Blatt und einmal mit maximal 20 Punkten pro Blatt getestet.

In den Tabellen ist zu erkennen, dass die Laufzeit von KDExtremal und OctreeExtremal deutlich langsamer wächst als linearSearch. In den entsprechenden Diagrammen wurde zusätzlich $O(n^{\log_8(m)})$, aus Kapitel 3.1.1, für verschiedene m eingezeichnet. Für Punkte auf der Oberfläche einer Kugel wächst OctreeExtremal ungefähr mit $O(n^{\log_8(3)})$, während KDExtremal für große n schneller wächst. Für Punkte in einer Kugel wachsen beide Algorithmen langsamer als $O(n^{\log_8(3)})$ und nähern sich fast $O(n^{\log_8(2)})$ an.

In den Tabellen ist auch zu erkennen, dass KDExtremal und OctreeExtremal davon profitieren können, wenn die maximale Anzahl an Punkten in den Blättern erhöht wird. Der Grund dafür ist wahrscheinlich, dass es ab einem bestimmten Punkt effizienter ist, mehr Punkte pro Blatt zu testen, als findBestCorner häufiger auszuführen, um Punkte auszuschließen.

[1] https://github.com/alecjacobson/common-3d-test-models

4.2 Berechnen der konvexen Hülle

In den Tabellen 4.3,4.4 und 4.5 werden der Quickhull Algorithmus aus CGAL und die drei Gift Wrapping Versionen, OctreeWrap, KDWrap und GiftWrapping, aus Kapitel 3.2.2 miteinander verglichen. In den Abbildungen 4.3 und 4.4 wurden die Tabellen 4.3 und 4.4 als Diagramme dargestellt. Auch hier wurden zufällig Punkte in einer Kugel und auf einer Kugel generiert. Zudem wurden die Algorithmen für einige 3D-Objekte getestet. Die Datenstrukturen wurden hierfür immer mit einer Blattgröße von zwei gebaut. Da die Größe der konvexen Hülle relevant für die Laufzeit der Gift Wrapping Algorithmen ist, wurde mit h angegeben, wie viele Punkte Teil der konvexen Hülle sind. Da das Bauen der Datenstrukturen bei großen n einem signifikanten Teil der Laufzeit entspricht, werden die Laufzeiten, zum Berechnen der konvexen Hülle und zum Bauen der Datenstrukturen, zusammengefasst. Wie viel der Laufzeit zum Bauen der Datenstrukturen genutzt wurde, ist in Klammern dahinter angegeben.

Während OctreeWrap und KDWrap für kleine n eine ähnliche Laufzeit haben wie GiftWrapping, sind sie für große n 1 bis 2 Größenordnungen schneller. Die Laufzeiten aus der Tabelle 4.3 können bedingt mit den Ergebnissen aus [1] verglichen werden, da die Punkte sehr ähnlich generiert wurden. Der dort Vorgestellte Algorithmus braucht im Durchschnitt ungefähr 51% weniger Zeit als der normale Gift Wrapping Algorithmus und wächst ähnlich schnell. Da die Laufzeit der hier vorgestellten Algorithmen OctreeWrap und KDWrap langsamer wachsen als GiftWrapping, lassen sich die Prozentsätze, um welche sie schneller sind, nur bedingt mit den Ergebnissen aus [1] vergleichen. Wenn alle Punkte Teil der konvexen Hülle sind, brauchen die Algorithmen ungefähr 70-87% weniger Zeit für 70000 Punkte als GiftWrapping. Bei Punkten, welche in einer Kugel generiert werden, wird die Zeit im besten Fall sogar um 99.5% reduziert. Diese Reduktionen in der Laufzeit sind so signifikant, dass angenommen werden kann, dass die hiesige Methode schneller ist als der Algorithmus von An et al. [1]. Auch für den Fall, dass der Gift Wrapping Algorithmus hier weniger effizient implementiert sein sollte.

Für die 3D-Objekte in Tabelle 4.5 sind OctreeWrap und KDWrap auch deutlich schneller als GiftWrapping.

Wie in Tabelle 4.4 zu sehen ist, erreicht OctreeWrap, für in der Kugel verteilte Punkte, eine ähnliche Laufzeit wie Quickhull und für große n sogar eine bessere Laufzeit. Für alle anderen Daten, bei denen das Verhältnis zwischen h und n größer ist, ist Quickhull hingegen um ein Vielfaches schneller.

In Abbildung 4.3 wurde $O(n^{log_8(3)+1})$ mit eingezeichnet, da die Funktionen in Abschnitt 4.1 ungefähr mit $O(n^{log_8(3)})$. Während KDWrap schneller als diese Funktion wächst, entspricht die Laufzeit von OctreeWrap fast der Funktion $O(n^{log_8(3)+1})$.

n	linearSearch	KDExtremal		OctreeExtremal	
		b=2	b=20	b=2	b=20
100	1.8	8.1	3.9	5.4	5.1
200	3.4	8.9	5.1	7.3	6.5
400	6.5	11.9	7.2	9.5	8.2
800	12.5	16.0	9.4	13.1	10.6
1000	15.5	19.0	10.4	15.4	11.6
2000	30.5	23.1	14.1	18.2	16.0
4000	59.7	29.7	18.7	23.3	21.1
8000	121.8	37.3	24.6	29.8	26.1
10000	146.3	40.8	29.2	31.9	28.7
20000	290.9	53.1	36.7	39.7	37.7
40000	575.1	66.0	44.9	51.2	42.9
80000	1150.0	85.4	57.4	63.9	58.2
100000	1429.4	92.8	61.6	69.2	65.1
200000	2845.3	123.9	79.0	88.0	82.2
400000	5706.7	161.6	100.4	108.5	96.2
800000	11364.0	215.1	134.8	139.4	131.1

Tabelle 4.1: Laufzeiten (in Mikrosekunden), um einen benachbarten Extremalpunkt zu finden, wenn die Punkte in einer Kugel verteilt sind.
b - maximale Anzahl an Punkten pro Blatt

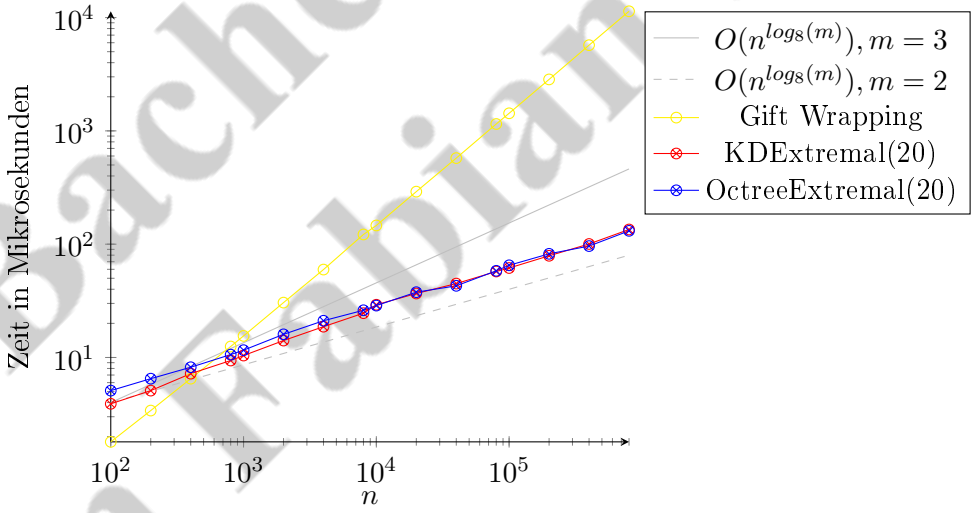


Abbildung 4.1: Laufzeiten (in Mikrosekunden), um einen benachbarten Extremalpunkt zu finden, wenn die Punkte in einer Kugel verteilt sind. Siehe Tabelle 4.1

n	linearSearch	KDExtremal		OctreeExtremal	
		b=2	b=20	b=2	b=20
100	1.8	6.3	4.0	6.1	5.0
200	3.4	10.7	5	8.3	6.0
400	6.5	11.7	6.7	11.7	8.7
800	12.5	17.0	9.6	16.6	11.8
1000	15.3	19.3	10.5	18.7	13.2
2000	29.9	30.4	14.6	27.7	21.0
4000	58.5	45.7	21.7	37.6	27.9
8000	115.6	71.8	33.2	52.2	40.6
10000	144.5	85.1	38.5	58.0	45.3
20000	285.4	134.3	58.2	81.7	61.5
40000	568.6	229.5	92.0	117.1	89.7
80000	1131.0	379.7	149.3	168.1	124.0
100000	1412.0	456.4	178.7	191.4	139.6
200000	2816.4	806.7	300.0	273.4	197.2
400000	5635.9	1475.2	524.5	397.1	284.3
800000	11295.3	2808.8	930.6	588.7	411.0

Tabelle 4.2: Laufzeiten (in Mikrosekunden), um einen benachbarten Extremalpunkt zu finden, wenn die Punkte auf der Oberfläche einer Kugel verteilt sind.
b - maximale Anzahl an Punkten pro Blatt

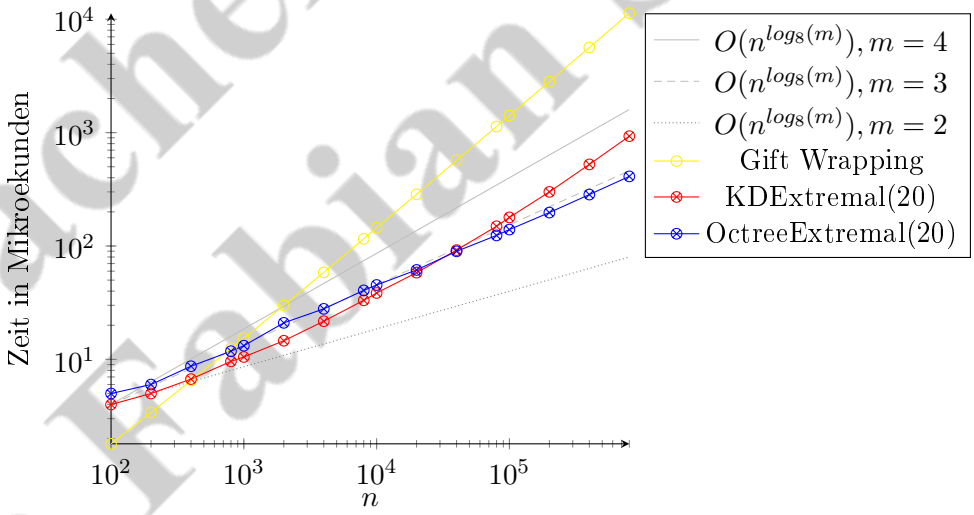


Abbildung 4.2: Laufzeiten (in Mikrosekunden), um einen benachbarten Extremalpunkt zu finden, wenn die Punkte auf der Oberfläche einer Kugel verteilt sind. Siehe Tabelle 4.2

n	Quickhull	Octree Wrap	Octree	KD Wrap	k-d-Baum	Gift Wrapping
1000	0.004	0.044	(0)	0.043	(0)	0.037
2000	0.008	0.125	(0.001)	0.155	(0.001)	0.146
5000	0.019	0.481	(0.001)	0.553	(0.001)	0.977
7000	0.027	0.703	(0.002)	1.099	(0.002)	2.003
10000	0.040	1.391	(0.003)	1.828	(0.003)	4.140
15000	0.064	2.574	(0.004)	3.997	(0.004)	9.379
20000	0.089	3.459	(0.005)	6.636	(0.006)	16.746
25000	0.114	5.255	(0.006)	9.761	(0.008)	26.225
30000	0.141	6.853	(0.008)	13.317	(0.009)	37.732
40000	0.197	11.771	(0.010)	22.689	(0.013)	67.182
50000	0.255	14.575	(0.012)	28.739	(0.016)	104.997
60000	0.333	21.908	(0.015)	46.814	(0.020)	151.407
70000	0.404	25.704	(0.018)	62.760	(0.024)	206.023

Tabelle 4.3: Laufzeiten (in Sekunden), um die konvexe Hülle zu berechnen, wenn die Punkte auf der Oberfläche einer Kugel verteilt sind.

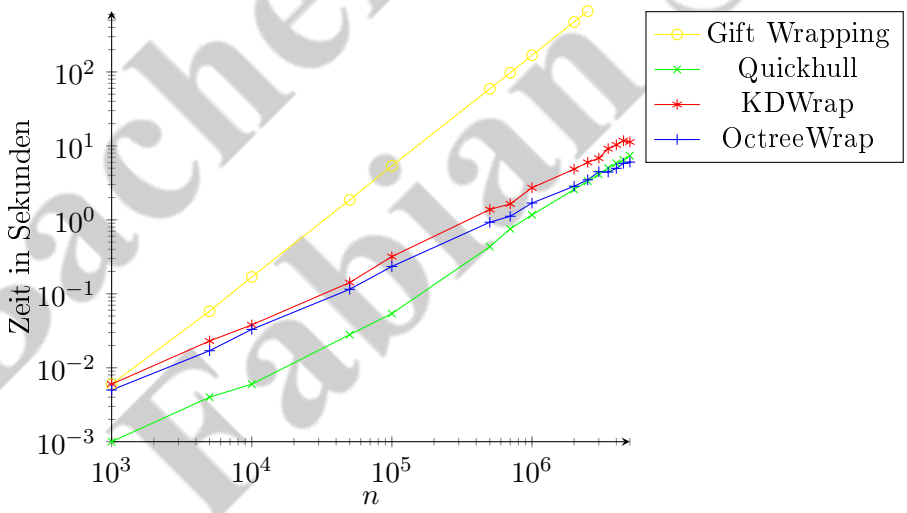


Abbildung 4.3: Laufzeiten (in Sekunden), um die konvexe Hülle zu berechnen, wenn die Punkte auf der Oberfläche einer Kugel verteilt sind. Siehe Tabelle 4.3

n	h	Quickhull	Octree Wrap	Octree	KDWrap	k-d-Baum	Gift Wrapping
1000	150	0.001	0.005	(0)	0.006	(0)	0.006
5000	304	0.004	0.017	(0.001)	0.023	(0.002)	0.058
10000	439	0.006	0.033	(0.002)	0.038	(0.003)	0.169
50000	956	0.028	0.115	(0.010)	0.142	(0.017)	1.860
100000	1382	0.054	0.234	(0.021)	0.318	(0.036)	5.374
500000	3041	0.434	0.928	(0.115)	1.379	(0.198)	59.134
700000	3585	0.756	1.118	(0.162)	1.642	(0.311)	97.484
1000000	4318	1.165	1.686	(0.238)	2.714	(0.493)	167.924
2000000	6100	2.577	2.843	(0.499)	4.853	(1.164)	474.676
2500000	6820	3.336	3.476	(0.623)	5.990	(1.474)	664.255
3000000	7483	4.124	4.474	(0.745)	6.825	(1.839)	
3500000	8081	5.017	4.431	(0.863)	9.164	(2.268)	
4000000	8605	5.811	4.969	(0.984)	10.321	(2.562)	
4500000	9202	6.453	5.802	(1.115)	11.759	(2.987)	
5000000	9649	7.387	6.033	(1.255)	11.306	(3.358)	

Tabelle 4.4: Laufzeiten (in Sekunden), um die konvexe Hülle zu berechnen, wenn die Punkte in einer Kugel verteilt sind. Gift Wrapping wurde nur bis 2500k Punkte getestet.

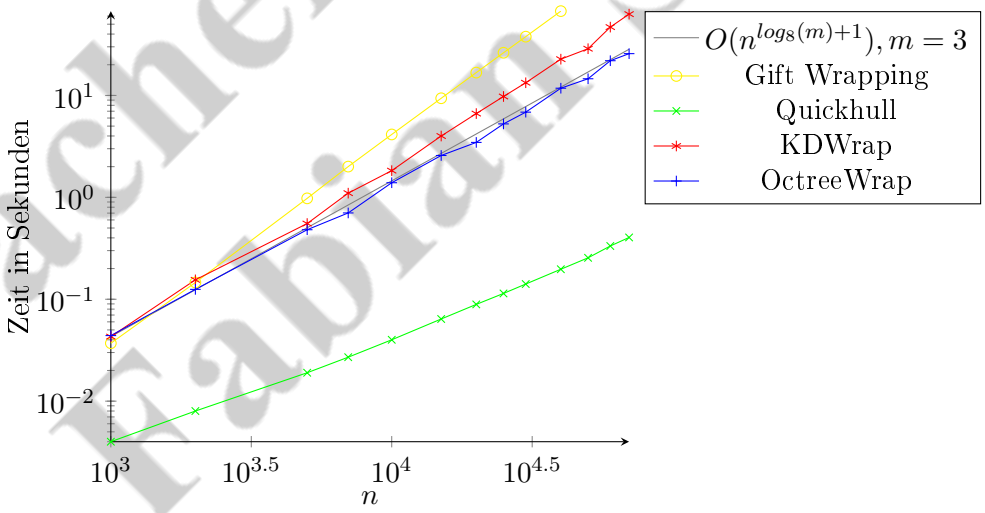


Abbildung 4.4: Laufzeiten (in Sekunden), um die konvexe Hülle zu berechnen, wenn die Punkte in einer Kugel verteilt sind. Gift Wrapping wurde nur bis 2500k Punkte getestet. Siehe Tabelle 4.4

Objekt	**n (h)**	**Quickhull**	**OctreeWrap (Octree)**	**KDWrap (k-d-Baum)**	**Gift Wrapping**
Homer	6002 (514)	0.004	0.039 (0.001)	0.049 (0.002)	0.096
Rocker Arm	10044 (1237)	0.009	0.123 (0.002)	0.171 (0.003)	0.380
Stanford Bunny	35947 (1562)	0.021	0.289 (0.007)	0.488 (0.011)	1.768
Lucy	49987 (299)	0.016	0.055 (0.011)	0.091 (0.015)	0.416
Max Planck	50077 (5916)	0.047	1.200 (0.011)	2.329 (0.015)	8.887
Bimba	112455 (4073)	0.058	1.401 (0.026)	3.261 (0.034)	13.289

Tabelle 4.5: Die Laufzeit (in Sekunden), um die konvexe Hülle von 3D-Objekten zu bestimmen, welche aus n vielen Punkte bestehen und h viele Punkte auf der konvexen Hülle haben. Für *OctreeWrap* und *KDWrap* ist in Klammern angegeben, wie viel der Laufzeit zum Bauen der Datenstrukturen genutzt wurde. Die Zeiten entsprechen dem Durchschnitt aus 3 Ausführungen.

5 Fazit

In dieser Arbeit wurde ein Algorithmus entwickelt, um effizient Extremalpunkte zu bestimmen, indem eine Linie oder eine Ebene um eine Achse rotiert wird. Es wurde gezeigt, dass der Algorithmus mit verschiedenen Datenstrukturen genutzt werden kann. Auf diese Weise wurde der Gift-Wrapping-Algorithmus zur Berechnung von konvexen Hüllen erheblich beschleunigt.

Obwohl die Worst-Case-Laufzeit zum Finden der Extremalpunkte $O(n)$ entspricht, konnte durch die Tests in Kapitel 4 gezeigt werden, dass die durchschnittliche Laufzeit der Algorithmen deutlich langsamer als $O(n)$ wächst und die Algorithmen effizienter sind als ein vergleichbarer Ansatz, der lineare Suche benutzt. Für die getesteten Eingaben erreichen die Algorithmen im Durchschnitt sogar schnellere Laufzeiten als $O(n^{log_8(3)})$. Die modifizierten Versionen von Gift Wrapping sind deutlich effizienter als der normale Gift Wrapping Algorithmus und scheinen auch schneller zu sein als der verbesserte Gift Wrapping Algorithmus von An et al. [1]. Für einige Fälle sind sie sogar schneller als der Quickhull Algorithmus aus CGAL [35].

Die hier entwickelten Algorithmen für Extremalpunkte sind deutlich schneller als vergleichbare Ansätze und könnten deshalb für Anwendungen genutzt werden, wo bereits räumliche Datenstrukturen gegeben sind und einzelne Extremalpunkte effizient bestimmt werden sollen. Die Algorithmen für konvexe Hüllen haben keine optimale asymptotische Laufzeit, sind aber deutlich schneller als Gift Warpping und könnten deshalb in Anwendungen benutzt werden, wo die Benutzung von Gift Wrapping sinnvoll ist oder wo Teile der konvexen Hülle schnell berechnet werden sollen. Wenn das Verhältnis von h zu n sehr klein ist, könnten die hier vorgestellten Algorithmen auch statt Quickhull genutzt werden, um eine bessere Laufzeit zu erreichen.

Die hier entwickelten Algorithmen sind noch nicht optimal und können noch in vielen Aspekten verbessert werden. Verbesserungen und Ideen, welche in zukünftigen Arbeiten umgesetzt oder untersucht werden könnten, sind:

1. Konvexe Hüllen, welche Seiten mit mehr als 3 komplanaren Punkten enthalten, richtig zu bestimmen, indem die Vorschläge von Sugihara [34] umgesetzt werden.

2. Den Algorithmus so anpassen, dass ungerichtete Winkel statt gerichteter Winkel benutzt werden. Damit könnte die Laufzeit um ein Vielfaches verringert werden.

3. Statt im $\mathbb{R}^3$ den Winkel zwischen 2 Ebenen miteinander zu vergleichen, könnte ein ähnlicher Ansatz wie in [12] untersucht und implementiert werden, um die Laufzeit zu verbessern und die Algorithmen besser auf höhere Dimensionen zu erweitern.

4. Das Ordnen der Regionen könnte so angepasst werden, dass weniger Ecken von findBestCorner untersucht werden müssen, die Berechnung von redundanten

Winkeln vermieden werden kann und die Regionen vielleicht sogar besser sortiert werden können.

5. Ein ähnlicher Algorithmus könnte vielleicht auch genutzt werden, um andere Algorithmen, wie Quickhull, zu verbessern. Die Verbesserung der Laufzeit könnte dort aber sehr klein ausfallen.

Literatur

[1] Phan Thanh An, Nam Dũng Hoang und Nguyen Kieu Linh. „An efficient improvement of gift wrapping algorithm for computing the convex hull of a finite set of points in $\mathbb {R}^{n}$“. In: *Numerical Algorithms* 85.4 (1. Dez. 2020), S. 1499–1518. ISSN: 1572-9265. DOI: 10.1007/s11075-020-00873-1.

[2] Mikhail J. Atallah, Susan Fox und Suzanne Lassandro. *Algorithms and Theory of Computation Handbook*. 1st. USA: CRC Press, Inc., 1998. ISBN: 0-8493-2649-4.

[3] Franz Aurenhammer. „Voronoi diagrams—a survey of a fundamental geometric data structure“. In: *ACM Computing Surveys* 23.3 (1. Sep. 1991), S. 345–405. ISSN: 0360-0300. DOI: 10.1145/116873.116880.

[4] C. Bradford Barber, David P. Dobkin und Hannu Huhdanpaa. „The quickhull algorithm for convex hulls“. In: *ACM Transactions on Mathematical Software* 22.4 (1. Dez. 1996), S. 469–483. ISSN: 0098-3500. DOI: 10.1145/235815.235821.

[5] Alan Barton. „CHAN's PLANAR CONVEX HULL ALGORITHM: A Brief Survey and Sequential Experimental Comparison“. In: *Journal of Scientific and Practical Computing: Part A* 1.1 (Dez. 2006).

[6] Jon Louis Bentley. „Multidimensional binary search trees used for associative searching“. In: *Communications of the ACM* 18.9 (1. Sep. 1975), S. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[7] Jon Louis Bentley, Dorothea Haken und James B. Saxe. „A general method for solving divide-and-conquer recurrences“. In: *ACM SIGACT News* 12.3 (1. Sep. 1980), S. 36–44. ISSN: 0163-5700. DOI: 10.1145/1008861.1008865.

[8] BK Bhattacharya und GT Toussaint. „Time- and storage-efficient implementation of an optimal planar convex hull algorithm“. In: *Image and Vision Computing* 1.3 (1. Aug. 1983), S. 140–144. ISSN: 0262-8856. DOI: 10.1016/0262-8856(83)90065-3.

[9] G.S. Brodal und R. Jacob. „Dynamic planar convex hull“. In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings. Nov. 2002, S. 617–626. DOI: 10.1109/SFCS.2002.1181985.

[10] T. M. Chan. „Optimal output-sensitive convex hull algorithms in two and three dimensions“. In: *Discrete & Computational Geometry* 16.4 (1. Apr. 1996), S. 361–368. ISSN: 1432-0444. DOI: 10.1007/BF02712873.

[11] Timothy M. Y. Chan, Jack Snoeyink und Chee-Keng Yap. „Output-sensitive construction of polytopes in four dimensions and clipped Voronoi diagrams in three“. In: *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*. SODA '95. USA: Society for Industrial und Applied Mathematics, 22. Jan. 1995, S. 282–291. ISBN: 978-0-89871-349-7.

[12] Donald R. Chand und Sham S. Kapur. „An Algorithm for Convex Polytopes“. In: *Journal of the ACM* 17.1 (1. Jan. 1970), S. 78–86. ISSN: 0004-5411. DOI: 10.1145/321556.321564.

[13] Bernard Chazelle. „An optimal convex hull algorithm in any fixed dimension“. In: *Discrete & Computational Geometry* 10.4 (1. Dez. 1993), S. 377–409. ISSN: 1432-0444. DOI: 10.1007/BF02573985.

[14] H. Edelsbrunner und H. A. Maurer. „Finding extreme points in three dimensions and solving the post-office problem in the plane“. In: *Information Processing Letters* 21.1 (10. Juli 1985), S. 39–47. ISSN: 0020-0190. DOI: 10.1016/0020-0190(85)90107-3.

[15] R. A. Finkel und J. L. Bentley. „Quad trees a data structure for retrieval on composite keys“. In: *Acta Informatica* 4.1 (1. März 1974), S. 1–9. ISSN: 1432-0525. DOI: 10.1007/BF00288933.

[16] R. L. Graham. „An efficient algorith for determining the convex hull of a finite planar set“. In: *Information Processing Letters* 1.4 (1. Juni 1972), S. 132–133. ISSN: 0020-0190. DOI: 10.1016/0020-0190(72)90045-2.

[17] R. A. Jarvis. „On the identification of the convex hull of a finite set of points in the plane“. In: *Information Processing Letters* 2.1 (1. März 1973), S. 18–21. ISSN: 0020-0190. DOI: 10.1016/0020-0190(73)90020-3.

[18] M. A. Jayaram und Hasan Fleyeh. „Convex Hulls in Image Processing : A Scoping Review“. In: *American Journal of Intelligent Systems* 6.2 (2016), S. 48–58.

[19] David G. Kirkpatrick und Raimund Seidel. „The Ultimate Planar Convex Hull Algorithm?“ In: *SIAM Journal on Computing* 15.1 (1986), S. 287–299. DOI: 10.1137/0215021.

[20] M. Krein und D. Milman. „On extreme points of regular convex sets“. In: *Studia Mathematica* 9.1 (1940), S. 133–138. ISSN: 0039-3223.

[21] Jean-Paul Laumond und Mark Overmars. *Algorithms for Robotic Motion and Manipulation: WAFR 1996*. CRC Press, 11. Feb. 1997. 481 S. ISBN: 978-1-4398-6452-4.

[22] Luca Liparulo, Andrea Proietti und Massimo Panella. „Fuzzy Clustering Using the Convex Hull as Geometrical Model“. In: *Advances in Fuzzy Systems* 2015 (21. Apr. 2015), e265135. ISSN: 1687-7101. DOI: 10.1155/2015/265135.

[23] Rong Liu, Hao Zhang und James Busby. „Convex hull covering of polygonal scenes for accurate collision detection in games“. In: Proceedings - Graphics Interface. 1. Jan. 2008, S. 203–210. DOI: 10.1145/1375714.1375749.

[24] Jiří Matoušek und Otfried Schwarzkopf. „Linear optimization queries“. In: *Proceedings of the eighth annual symposium on Computational geometry*. SCG '92. New York, NY, USA: Association for Computing Machinery, 1. Juli 1992, S. 16–25. ISBN: 978-0-89791-517-5. DOI: 10.1145/142675.142683.

[25] Mary M McQueen und Godfried T Toussaint. „On the ultimate convex hull algorithm in practice“. In: *Pattern Recognition Letters* 3.1 (1. Jan. 1985), S. 29–34. ISSN: 0167-8655. DOI: 10.1016/0167-8655(85)90039-X.

[26] A. P. Nemirko und J. H. Dulá. „Machine learning algorithm based on convex hull analysis“. In: *Procedia Computer Science*. 14th International Symposium Ïntelligent Systems 186 (1. Jan. 2021), S. 381–386. ISSN: 1877-0509. DOI: 10.1016/j.procs.2021.04.160.

[27] Jack A. Orenstein. „Multidimensional tries used for associative searching“. In: *Information Processing Letters* 14.4 (13. Juni 1982), S. 150–157. ISSN: 0020-0190. DOI: 10.1016/0020-0190(82)90027-8.

[28] F. P. Preparata und S. J. Hong. „Convex hulls of finite sets of points in two and three dimensions“. In: *Communications of the ACM* 20.2 (1. Feb. 1977), S. 87–93. ISSN: 0001-0782. DOI: 10.1145/359423.359430.

[29] Franco P. Preparata und Michael I. Shamos. *Computational geometry: an introduction*. Berlin, Heidelberg: Springer-Verlag, Aug. 1985. 390 S. ISBN: 978-0-387-96131-6.

[30] R. Tyrrell Rockafellar. *Convex Analysis*. Princeton University Press, 12. Jan. 1997. 482 S. ISBN: 978-0-691-01586-6.

[31] Hanan Samet. *The design and analysis of spatial data structures*. USA: Addison-Wesley Longman Publishing Co., Inc., 1990. 493 S. ISBN: 978-0-201-50255-8.

[32] R Seidel. „Constructing higher-dimensional convex hulls at logarithmic cost per face“. In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. STOC '86. New York, NY, USA: Association for Computing Machinery, 1. Nov. 1986, S. 404–413. ISBN: 978-0-89791-193-1. DOI: 10.1145/12130.12172.

[33] Ayal Stein, Eran Geva und Jihad El-Sana. „CudaHull: Fast parallel 3D convex hull on the GPU“. In: *Computers & Graphics*. Applications of Geometry Processing 36.4 (1. Juni 2012), S. 265–271. ISSN: 0097-8493. DOI: 10.1016/j.cag.2012.02.012.

[34] Kokichi Sugihara. „Robust gift wrapping for the three-dimensional convex hull“. In: *Journal of Computer and System Sciences* 49.2 (1. Okt. 1994), S. 391–407. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(05)80056-X.

[35] The CGAL Project. *CGAL User and Reference Manual 5.6*. 2023. URL: https://doc.cgal.org/5.6/Manual/packages.html.

[36] Andrew Chi-Chih Yao. „A Lower Bound to Finding Convex Hulls“. In: *Journal of the ACM* 28.4 (1. Okt. 1981), S. 780–787. ISSN: 0004-5411. DOI: 10.1145/322276.322289.

[37] Mann-May Yau und Sargur N. Srihari. „A hierarchical data structure for multidimensional digital images“. In: *Communications of the ACM* 26.7 (1. Juli 1983), S. 504–515. ISSN: 0001-0782. DOI: 10.1145/358150.358158.