Institut für Informatik VII
Robotik und Telematik

Bachelorarbeit

Analyse und Optimierung einer effizienten k-d-Baum-Implementierung

Johannes Barthelmes

Oktober 2016

Erstgutachter: Prof. Dr. Andreas Nüchter
Zweitgutachter: MSc. Johannes Schauer

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit der k-d-Baum-Implementierung in *3DTK - The 3D Toolkit*, einer Softwaresammlung zur Verarbeitung von 3D-Punktwolken. Der k-d-Baum lässt sich bei seiner Konstruktion nach der Blattgröße und der Partitionsstrategie parametrisieren. Die Blattgröße legt fest, wie viele Punkte in einem Blatt zusammengefasst werden können. Die Partitionsstrategie legt fest, wie bei einem inneren Knoten die Punktemenge aufgeteilt wird. Es wird untersucht, wie sich unterschiedliche Blattgrößen und Partitionsstragtegien auf die Laufzeiten der Konstruktion des Baums und der Nächste-Nachbarn-Suche im Baum auswirken. Dazu wird die Zeit gemessen, die die beiden Operationen auf unterschiedlichen Datensätzen brauchen. Als Ergebnis kann ein Wertebereich für die optimale Blattgröße angegeben werden. Es wurde ebenfalls eine zuverlässig gute Partitionsstrategie gefunden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

3D-Scanner sind wichtige Werkzeuge in vielen Anwendungsgebieten. Sie dienen unter anderem in der Robotik als Sensor der Umgebung [15], im Bauingenierwesen dem Vermessen und Planen von Gebäuden [5], in der Geologie dem Studieren von Steinschlägen [1]

Häufig anzutreffen sind 3D-*Laser*scanner, die die Rückflugzeit oder Phasenverschiebung eines emittierten Laserstrahls messen und so die Distanz zu dem reflektierenden Objekt erfassen [12]. Um die Umgebung zu vermessen wird meist ein Spiegel gegenüber dem Laserstrahler geschwenkt oder der ganze Scanner gedreht. Auf diese Weise entsteht ein Panorama der gemessenen Distanzen.

Da man den horizontalen und vertikalen Winkel des Scanners bei jeder Distanzmessung kennt, kann man das Messergebnis auch zu einer Punktemenge (auch *Punktwolke*) in einem dreidimensionalen Raum umrechnen. Führt man mehrere solcher *Scans* durch, z.B. um ein Gebäude herum, möchte man ihre Punktwolken meist zu einer einzigen zusammenführen, um einen Rundum-Blick zu erhalten.

Dem steht jedoch im Wege, dass der Scanner zwischen den Scans bewegt wird, und dadurch die Verschiebung und die Drehung des Scanners zwischen den Messungen nicht bekannt sind. Die verfügbare Odometrie, wie z.B. GPS, Beschleunigungssensoren oder Drehzahlmesser, reicht alleine nicht aus, um die Bewegung des Scanners genau genug zu ermitteln. Deshalb existieren Verfahren, um alleine von den gescannten Punktwolken auf die Ausrichtung der Scans zueinander zu schließen (*matching*).

Ein solches Verfahren ist der *Iterative Closest Point (ICP)*-Algorithmus [4]. Während dieser die Ausrichtung zweier Scans aneinander ermittelt, muss er sehr häufig zu jedem Punkt des einen Scans den allernächsten Punkt aus dem anderen Scan suchen.

In *3DTK – The 3D Toolkit* [11], sind deshalb die 3D-Punkte in einer Datenstruktur gespeichert, die für die *Nächste-Nachbarn-Suche* besonders effizient ist: dem *k-d-Baum*. Nicht alle k-d-Bäume

sind jedoch gleich geschaffen: man kann einige unterschiedliche Kriterien bei der Konstruktion der Datenstruktur anwenden, durch die sich die Struktur des Baums verändert.

Herauszufinden, welche Konfiguration des k-d-Baums, so wie er in 3DTK implementiert ist, optimal ist, ist Ziel dieser Arbeit.

1.2 Zielsetzung und wissenschaftlicher Beitrag

Das Erfassen und Verarbeiten von 3D-Punktwolken ist ein Forschungsgebiet, in und mit dem noch viel Arbeit geleistet werden kann. Viele Fachgebiete tragen einen Nutzen von der Verwendung von 3D-Scannern. Um diese noch besser nutzen zu können, sind vor allem gute Software-Werkzeuge nötig.

3DTK – The 3D Toolkit [11] ist eine Softwaresammlung, die auf die Arbeit mit 3D-Punktdaten ausgelegt ist. Es verfügt bereits über die schnellste quelloffen verfügbare Implementierung eines k-d-Baums für dreidimensionale Daten [8]. Das Ausrichten von Scans und andere Aufgaben sind trotzdem noch sehr zeitaufwändig.

Durch Arbeit an der k-d-Baum-Implementierung kann 3DTK als Werkzeug verbessert werden und somit allen Anwendern dienen.

Für die vorliegende Arbeit sollte Quellcode produziert werden, der auch sofort ins 3DTK übernommen werden kann.

Des Weiteren wurde beim Aufbau der Experimente und dem Aufbereiten der Ergebnisse großer Wert auf Reproduzierbarkeit gelegt.

Die Experimentierumgebung lässt sich auch für zukünftige Performance-Messungen an 3DTK verwenden.

1.3 Aufbau der Arbeit

Zunächst werden die Algorithmen und Datenstrukturen erklärt, die beim Matching in 3DTK eine Rolle spielen. Dazu zählen der k-d-Baum, der ICP-Algorithmus und die Nächste-Nachbar-Suche im k-d-Baum. Durch eine Analyse der auftretenden Laufzeiten können wir die Ergebnisse besser in Kontext setzen.

Anschließend wird der verwendete Versuchsaufbau beschrieben. Dabei wird erklärt, wie die Laufzeiten der Algorithmen gemessen werden, welche Änderungen dazu an 3DTK vorgenommen wurden, und auf welchen Datensätzen die Messungen stattfanden.

Zuletzt werden die Ergebnisse der Versuche präsentiert. Dazu erfolgt eine Erklärung und eine Diskussion.

Kapitel 2

Erklärung der Algorithmen und Datenstrukturen

2.1 k-d-Bäume

k-d-Bäume dienen als effiziente Datenstruktur für viele geometrische Anfragen, wie z.B. die Nächste-Nachbarn-Suche. Der Name des k-d-Baums rührt daher, dass er im allgemeinen k-dimensionale Daten speichert.

Wir betrachten, wie der k-d-Baum theoretisch beschrieben wird, und wie er in 3DTK optimiert implementiert ist.

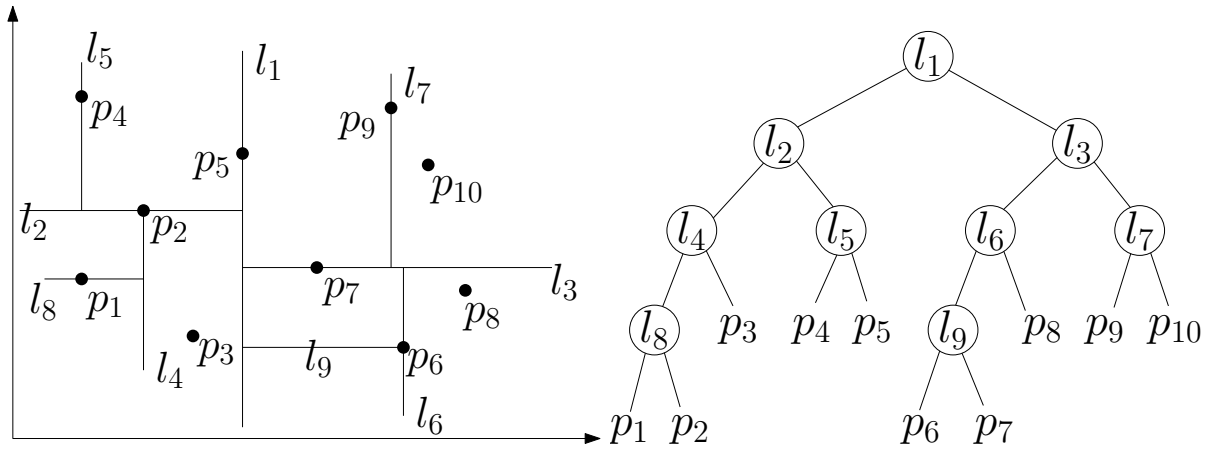
2.1.1 Lehrbuch-k-d-Baum

Ein k-d-Baum ist ein binärer Baum. In den Blättern des k-d-Baums sind die k-dimensionalen Daten (*Punkte*) gespeichert. In den inneren Knoten ist ein einzelner Wert gespeichert, mit dem eine binäre Suche in den richtigen Zweig des Baums findet. Dieser Wert, an dem die Abzweigung entschieden wird, gilt nur für eine bestimmte Komponente aller Punkte.

Das heißt, eine binäre Suche muss zu jedem dieser Werte wissen, mit welcher der Komponenten eines gesuchten Punktes der Wert verglichen werden muss.

Die Regel dafür lautet beim Lehrbuch-k-d-Baum (z.B. [7]), dass die Komponente, mit der verglichen werden muss, je nach der Tiefe der Suche im Baum der Reihe nach feststeht. Auf der Wurzelebene ist das die erste Komponente, auf der nächsten Ebene die zweite Komponente und so weiter, bis mit allen Komponenten einmal die Abzweigung entschieden wurde, wonach wieder die erste an der Reihe ist.

Betrachtet man den k-d-Baum geometrisch, dann sind die k-dimensionalen Datensätze Punkte im k-dimensionalen Raum. Ein innerer Knoten steht dann für eine achsparallele (Hyper-)Ebene, die die Menge der Punkte in Zwei teilt. Das ist in Abbildung 2.1a veranschaulicht.



(a) Geometrische Betrachtung **(b)** Algebraische Betrachtung

Abbildung 2.1: Geometrische und algrebraische („strukturelle“) Betrachtung des selben zweidimensionalen k-d-Baums. Mit p sind die gespeicherten Punkte bezeichnet, mit l die Trennlinien.

Eine Ebene und ihre Vorgänger im Baum begrenzen einen achsparallelen Quader, in dem sich alle Punkte befinden müssen, die im Baum ihre Nachfahren sind. Dieser Quader muss die Punkte jedoch nicht eng umschließen. Insbesondere ist er, bis er von $2k$ Ebenen begrenzt wird, auf mindestens einer Seite offen. Wenn eine geometrische Anfrage den eng umschließenden achsparallelen *Um*quader (engl. *axis-aligned bounding box*, im Folgenden *AABB*) der Punkte benötigt, muss dieser erst ermittelt werden.

Es bleibt noch die Frage, wo die Ebene, die die Punkte aufteilt, platziert wird. Beim Lehrbuch-k-d-Baum wird die Ebene auf dem Median der Punkte platziert. Das hat zum Vorteil, dass (per Definition des Medians) die Punktemenge immer hälftig geteilt wird, und dadurch der Baum am Ende balanciert ist. Eine andere Variante ist, die Ebene einfach die AABB hälftig teilen zu lassen. Dazu muss selbstverständlich die AABB bekannt sein.

Wir können nun darauf schließen, wie ein k-d-Baum konstruiert werden muss. Algorithmus 1 konstruiert einen k-d-Baum, so wie er oft in der Literatur zu finden ist (z.B. [7]). Er teilt die Punktemenge am Median. Ein Algorithmus, der den Median berechnet, ist zum Beispiel bei Cormen et al. [6] zu finden.

Die Richtung, in der er die Punktemenge aufteilt, ist je nach der Tiefe im Baum der Reihe nach festgelegt. Der Grund dafür ist, dass die Richtung, die zeitlich am längsten nicht geteilt wurde, vermutlich auch räumlich noch die längste ist, also die mit dem größten Abstand der extremen Punkte.

Wenn die Punktemengen immer ihrer längsten Achse entlang aufgeteilt werden, werden sie räumlich immer mehr kompakt. Das heißt, die Form der AABBs nähert sich bei zunehmender Tiefe im Baum dem Würfel an. Das ist sehr günstig für die Nächste-Nachbarn-Suche, da man dann eine größere Anzahl an Punkten als nächsten Nachbarn ausschließen kann.

Algorithmus 1 : LehrbuchKDBaum($k \in \mathbb{N}$, $P \subset \mathbb{R}^k$, $d \in \mathbb{N}_0$)

Eingabe : Dimension k, Punkte P im k-dimensionalen Raum, jetzige Tiefe d im Baum
Ausgabe : Die Wurzel eines k-d-Baums für P

```
// Abbruchbedingung
if |P| > 1 then
    // Teile P auf, und zwar der Reihe nach in jeder Richtung des Raumes.
    // Die Richtung, die an der Reihe ist, ist die r-te.
    r ← d mod k;
    // Der Median gibt die Grenze für die Aufteilung an.
    // Suche den Median über der r-ten Komponente aller Punkte.
    m ← Median({p_r | (p_0, p_1, ..., p_{k-1}) ∈ P});
    P_1 ← {p | p = (p_0, p_1, ..., p_{k-1}) ∈ P : p_r ≤ m};
    P_2 ← {p | p = (p_0, p_1, ..., p_{k-1}) ∈ P : p_r > m};
    Sei v ein neuer Knoten;
    // v.key ist der Schlüssel für die Suche im Baum. Entspricht den l_i im
       Diagramm.
    v.key ← m;
    v.left ← LehrbuchKDBaum(k, P_1, d + 1);
    v.right ← LehrbuchKDBaum(k, P_2, d + 1);
    return v;
else
    // Es verbleibt ein einziger Punkt in P
    {p} ← P;
    return p
```

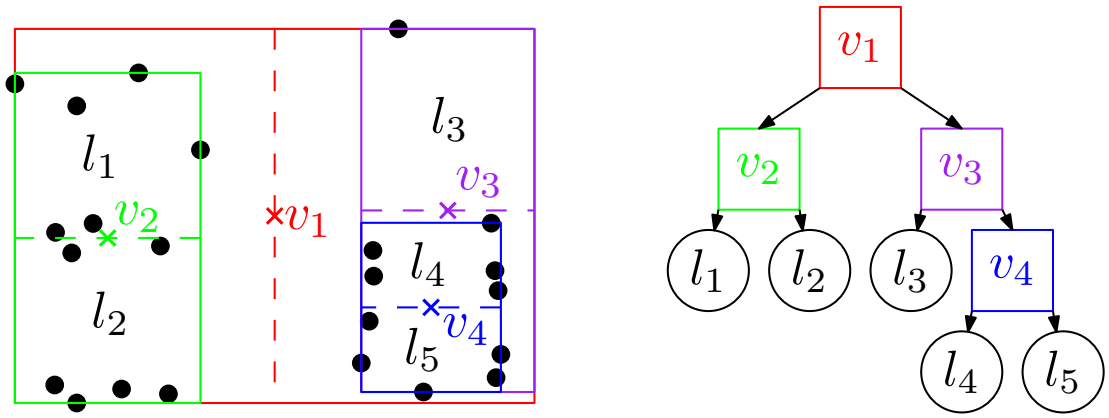
2.1.2 Der k-d-Baum in 3DTK

Zwischen dem theoretischen k-d-Baum und dem, der in 3DTK implementiert ist, bestehen einige Unterschiede.

Alle Aufgaben des k-d-Baums in 3DTK, einschließlich der Nächste-Nachbarn-Suche, profitieren davon, schon während des Konstruierens für jeden inneren Knoten die AABB zu berechnen. Dadurch kann eine Suchoperation schneller abgebrochen werden, wenn sie in den falschen Zweig des Baums läuft.

Anhand der vorberechneten AABB kann außerdem festgestellt werden, welche Achse der Punktmenge tatsächlich am längsten ist. Deshalb legt in 3DTK nicht die Tiefe im Baum die Achse der Aufteilung fest, sondern die Achse wird als die längste Seite der AABB ermittelt und in dem Knoten gespeichert, der gerade konstruiert wird.

Ursprünglich wurde außerdem nicht die Punktemenge an ihrem Median geteilt, sondern an der Hälfte der längsten AABB-Seite. Das entspricht einer Teilung am Mittelpunkt der AABB. Das garantiert keine balancierte Aufteilung mehr, aber da die AABB schon bekannt ist, kostet es nur



(a) Geometrische Betrachtung. Die schwarzen Punkte stehen für die gespeicherten Punkte. Die durchgezogenen Rechtecke sind die AABBs der jeweiligen Punktemenge. Die gestrichelten Linien sind die Trennlinien bei Halbierung der längsten Seite der AABB. Die Kreuze markieren den Mittelpunkt der AABBs.

(b) Baumstruktur. Innere Knoten sind mit einem Quadrat und Blätter mit einem Kreis dargestellt.

Abbildung 2.2: Der k-d-Baum in 3DTK speichert zu jedem inneren Knoten die AABB. Mit v (*vertex*) sind die inneren Knoten bezeichnet, mit l die Blätter (*leaf*). Die Konstruktion stoppt bei einer Blattgröße von zehn. Grafik inspiriert von Schauer et al. [13].

eine sehr kleine Zahl an Rechenschritten. Für diese Arbeit wurde die Möglichkeit hinzugefügt, auch nach dem Durchschnitt und Median der Punkte aufzuteilen, um die Laufzeit aller Methoden vergleichen können.

In der Praxis wird die Konstruktion des Baums auch nicht erst abgebrochen, wenn nur ein einziger Punkt in einem Zweig übrig bleibt, sondern schon früher. In einem Blatt des Baums können die übrigen Punkte dann in einem Array gespeichert werden. Alle Operationen, die an ein Blatt gelangen, müssen dann eine lineare Suche über die Punkte machen. Dadurch verkleinert man den Baum um viele Ebenen. Man erspart sich das Verfolgen der Kind-Pointer auf diesen Ebenen und die damit verbundenen Cache Misses, wenn neue Knoten in den Prozessor-Cache geladen werden müssen. Es ist interessant, zu wissen, ab welcher Anzahl an Punkten in einem Blatt die lineare Suche oder das Verfolgen von Pointern langsamer ist.

In Abbildung 2.2a wird zweidimensional veranschaulicht, wie der k-d-Baum in 3DTK aufgebaut ist. v_1 ist die Wurzel des Baums. Sie soll 23 Punkte speichern. Das ist mehr als die Blattgröße von zehn, weshalb sie die Punktemenge aufteilen muss. Ihre längste Achse ist die horizontale, also wird nach dieser geteilt (die Trennlinie ist dann vertikal). In diesem Diagramm werden die Punkte am Mittelpunkt der AABB geteilt. Bei v_2 geschieht das gleiche; der Knoten soll zwölf Punkte speichern und muss deshalb die Punktemenge teilen. v_3 soll elf Punkte Speichern.

Wie man deutlich erkennen kann, wird durch das Teilen am Mittelpunkt der AABB gelegentlich eine sehr ungleichmäßige Aufteilung geschaffen. Bei v_4 sieht man, dass die Achse der Teilung die

Listing 2.1: k-d-Baum-Attribute in 3DTK. Zur besseren Verständnis leicht abgeändert.

```
class KDTree {
int npts; // Anzahl an Punkten im Blatt bzw. 0 bei inneren Knoten
union {
  struct { // Innerer Knoten
    double center[3]; // Mittelpunkt der AABB
    double d[3]; // Abstaende der AABB-Seiten zur Mitte
    int splitaxis; // Richtung der Aufteilung
    double splitval; // Position der Aufteilung
    KDTree *child1, *child2; // Linkes und rechtes Kind
  } node;
  struct { // Blatt
    double** p; // Pointer auf das Punkt-Array
  } leaf;
};
// Speichert Parameter und Zwischenergebnisse der NNS und anderer Methoden
static struct {...} params;
...
}
```

Tiefe im Baum nicht beachtet, wie es bei der Lehrbuchdefinition der Fall wäre. Bei den Blättern wird einfach die Punktemenge in ein Array geschrieben.

Listing 2.1 zeigt, wie die Daten konkret in den Knoten des Baums gespeichert sind. Ob ein Objekt der Klasse `KDTree` ein Blatt oder ein innerer Knoten ist, kann man an `npts` feststellen. Für innere Knoten wird hier der Wert `0` gespeichert, bei Blättern die Anzahl der Punkte, die sie speichern. Die Unterscheidung ist nötig, um zu wissen, ob die *C++ Union*, die folgt, als `node` für innere Knoten oder als `leaf` für Blattknoten angesprochen werden muss. Bei einer C++ Union teilen sich die enthaltenen Variablen (hier also die Structs `node` und `leaf`) die gleichen Bytes im Speicher. Der Compiler interpretiert den Speicher dann mit den Datentypen der Variablen deren Namen man verwendet.

Für eine Erklärung des `params`-Attributs verweise ich auf Abschnitt 2.3. Es speichert die Parameter und Zwischenergebnisse der aktuell laufenden Methode des k-d-Baums. Eine genauere Beschreibung der k-d-Baum-Implementierung in 3DTK kann man außerdem bei Schauer und Nüchter [13] finden.

Wie die AABB in 3DTK abgespeichert ist, veranschaulicht Abbildung 2.3. Die AABB wird in 3DTK durch ihren Mittelpunkt und die halben Seitenlängen repräsentiert. Der Mittelpunkt ist in `center` gespeichert und die halben Seitenlänge in den `d[i]`. Die Eckpunkte der AABB sind dann (`center[0]` $\pm$ `d[0]`, `center[1]` $\pm$ `d[1]`, `center[2]` $\pm$ `d[2]`).

Die Variablen `splitaxis` und `splitval` ergeben zusammen die aufteilende Ebene für die Punktemenge. Dabei steht `splitaxis` für den Richtungsvektor, der der Normalvektor zur Ebene ist. Wie auch bei den Indices für die Punkte steht hier 0 für die x-Richtung, 1 für die y-Richtung und 2 für die z-Richtung. Die Position der Ebene wird durch `splitval` in absoluten Koordinatenwerten angegeben.

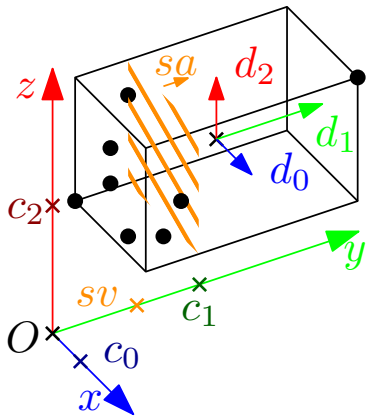


Abbildung 2.3: Die AABB in 3DTK. Vergleiche mit Listing 2.1. c_i entsprechen `center[i]`, d_i entsprechen `d[i]`. *sa* steht für `splitaxis`, *sv* für `splitvalue`.

Algorithmus 2 fasst in Pseudocode zusammen, wie der k-d-Baum in 3DTK konstruiert wird. Die Punktemenge wird als Feld übergeben, wobei der aktuelle rekursive Aufruf sich auf das Teilfeld beschränken muss, das durch $left$ und $right$ begrenzt wird.

Zuerst wird die AABB der Punkte ermittelt. Oft werden AABBs über ihre Eckpunkte definiert. Die Eckpunkte min und max dienen hier aber nur dazu, den Mittelpunkt und die halbe Seitenlänge der AABB zu berechnen, denn das sind die Werte, über die hier die AABB definiert wird. Anschließend kann die längste Seite der AABB ermittelt werden, damit gleich danach, je nach der Partitionsstrategie, der Trennwert ermittelt wird. Bei den Partitionsstrategien steht *halb* für die Halbierung der längsten AABB-Seite, *durchschnitt* für den Durchschnittswert der Punkte in der Trennachse, und *median* für den Median der Punkte in der Trennachse.

Ein Partitionsalgorithmus, wie er z.B. von Cormen et al. [6] beschrieben wird, ordnet die Punkte im Feld dann um. Das Ergebnis davon ist, dass zwei Teilfelder entstehen, wobei sich in dem linken Teilfeld alle Punkte kleiner als der Trennwert befinden und in dem rechten alle größer als der Trennwert. Die Indices dieser Teilfelder können dann an die rekursiven Aufrufe für die Kinder übergeben werden. Dadurch erspart man sich, neuen Speicher für deren Teilmengen der Punkte zu reservieren.

Falls die Anzahl der Punkte, die gespeichert werden sollen, aber klein genug ist (angegeben durch den Parameter b), wird ein Blatt erstellt. Dazu werden die maximal b Punkte in das Blatt-Array $leaf.p$ kopiert.

Zusammenfassend unterscheidet sich der k-d-Baum in 3DTK insofern, dass man ihn in der Größe seiner Blätter und der Art der Aufteilung der Punkte parametrisieren kann. Wir werden später sehen, welche Auswirken die Wahl unterschiedlicher Parameter hat.

2.2 Iterative Closest Point

Der *Iterative Closest Point* Algorithmus [4] richtet zwei Punktwolken aneinander aus. Das bedeutet, er sucht eine Translation und Drehung der einen Punktwolke relativ zu der anderen, sodass die in beiden enthaltenen Punkte so gut wie möglich übereinstimmen. Als Metrik dafür verwendet er die quadrierten Distanzen aller *zusammengehörigen* Punktepaare.

Es seien $M_1, M_2, \ldots, M_m$ die Punkte in dem Ausgangsdatensatz (*model set*), dessen Position im Raum als richtig angenommen wird. Außerdem seien $D_1, D_2, \ldots, D_d$ die Punkte in dem Zieldatensatz (*data set*), der relativ zu dem anderen bewegt werden soll. Die Zusammengehörigkeit der Punkte sei in der Matrix $\omega \in \{0,1\}^{m \times d}$ gespeichert, wobei 1 dafür steht, dass die Punkte zusammengehören. Dann ergibt sich folgende Fehlerfunktion für eine Translationsmatrix $\mathbf{T}$ und Rotationsmatrix $\mathbf{R}$ für den Zieldatensatz:

$$E(\mathbf{T}, \mathbf{R}) = \sum_{i=1}^{m} \sum_{j=1}^{d} \omega_{i,j} \| M_i - (\mathbf{R} D_j + \mathbf{T}) \|^2$$

Das ist die Summe der quadrierten Entfernungen aller zusammengehörigen Punkte in dem originalen Modelldatensatz und dem gedrehten und verschobenen Zieldatensatz, abhängig von der Verschiebung und Drehung.

Man kann die Betrachtung von ω in dieser Formel ignorieren, indem man im Voraus schon in M_i und D_i, also bei dem gleichen Index, die Punkte eines Paares speichert. Dann ist selbstverständlich $m = d$.

Arun et al. [3] haben gezeigt, dass man dann die Fehlerfunktion so vereinfachen kann, dass eine geschlossene Form für sie existiert. Dazu betrachtet man zunächst die Datensätze relativ zu ihren Centroiden c_M und c_D:

$$c_M = \frac{1}{m} \sum_{i=1}^{m} M_i \qquad c_D = \frac{1}{d} \sum_{i=1}^{d} D_i$$
$$M'_i = M_i - c_M \qquad D'_i = D_i - c_D$$

Ohne ins Detail zu gehen, kann man dann zuerst durch Minimieren von

$$E'(\mathbf{R}) = \sum_{i=1}^{m} \| M'_i - \mathbf{R} D'_i \|^2$$

mit einer *Singulärwertzerlegung* die optimale Rotationsmatrix finden. Anschließend findet man trivial die Translation $\mathbf{T} = \mathbf{R} c_D - c_M$.

Dieser Ansatz funktioniert in den meisten Fällen. Die möglichen Fehlerfälle wurden von Arun et al. [3] auch beschrieben.

Nachdem die Translation und Rotation gefunden wurden, werden sie auf den Zieldatensatz angewandt und es werden erneut Punktzusammengehörigkeiten und danach bessere Translations- und Rotationsmatrizen gesucht. Das Verfahren wird *iteriert*, bis die gefundene Translation und Rotation gut genug ist oder eine Maximalzahl an Iterationen überschritten ist.

Das fehlende Puzzlestück ist noch die Definition der Zusammengehörigkeit: ein Punkt gehört zu einem anderen Punkt, wenn er dessen *closest point*, der nächste Nachbar, ist.

Um die Zusammengehörigkeiten der Punkte festzustellen, muss der ICP-Algorithmus in jeder Iteration zu jedem Punkt des Modelldatensatzes den nächsten Nachbarn im Zieldatensatz suchen. Diese Operation ist deshalb sehr zeitkritisch. Deshalb wird in 3DTK der k-d-Baum verwendet, um diese Operation auszuführen, denn er ermöglicht eine schnelle Nächste-Nachbarn-Suche.

Ein sinnvoller Weg, die Dauer der Suche weiter zu beschränken, ist, eine maximale Distanz für die Nachbarschaft festzulegen. Dadurch werden Außenseiter, deren Position beim quadrierten Abstand stark ins Gewicht fallen würde, eliminiert. Außerdem findet die Suche durch die Maximaldistanz schnell in den richtigen Zweig des k-d-Baums.

2.3 Nächste-Nachbarn-Suche

Die Nächste-Nachbarn-Suche sucht zu einem Anfragepunkt den nähesten Punkt in einer Menge von Punkten. Der Anfragepunkt muss nicht selbst in der Menge enthalten sein.

Die Nächte-Nachbarn-Suche im k-d-Baum macht sich die geometrische Struktur des Baums zu Nutze. Damit wird erreicht, dass man viele Punkte der gespeicherten Punktwolke nicht auf ihre Nähe zum Anfragepunkt überprüfen muss.

Anstatt jeden Punkt betrachten zu müssen, kann man dank der Aufteilung des Raumes durch die Trennebenen einen großen Teil der Punkte als nächsten Nachbarn ausschließen. Das funktioniert, da man alle Punkte ausschließen kann, die weiter entfernt als der bisher beste gefundene nächste Nachbar liegen.

Das bedeutet auch, dass man alle ihre Elternknoten nicht mehr besuchen muss, sofern deren AABB nicht mehr von der „Suchkugel“ um den Anfragepunkt bis zum besten Kandidaten geschnitten oder eingeschlossen wird. Den Radius der Suche kann man anfänglich schon festlegen, um nur nächste Nachbarn zu finden, die auch innerhalb dieses Radius liegen.

Die genaue Parametrisierung der Suche erfolgt in 3DTK mit dem `params`-Struct, der in Listing 2.2 aufgeführt ist. Mit `params.p` wird der Punkt übergeben, dessen nächster Nachbar gesucht ist. In `params.closest` wird die Suche den besten Kandidaten festhalten, den sie bisher gefunden hat. In `params.closest_d2` wird zum einen die Suche den quadrierten Abstand festhalten, der zwischen dem Anfragepunkt und dem besten Kandidaten besteht. Zum anderen kann hiermit vor Beginn der Suche der gewünschte maximale Suchradius angegeben werden, denn der hier gespeicherte Wert ist der, mit dem bei der Suche die fernen Punkte ausgeschlossen werden. Wie besprochen ist es auch sinnvoll, hiermit eine maximale Distanz für die Nachbarschaft festzulegen.

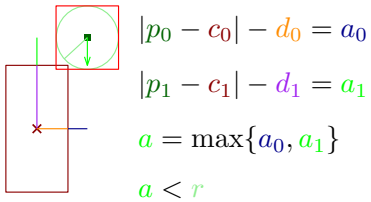


Abbildung 2.4: Visuelle Darstellung des Abbruchkriteriums. Das kleine Quadrat ist der Anfragepunkt. Der hellgrüne Kreis zeigt den bisher besten Abstand zum Anfragepunkt. Das dunkelrote Rechteck ist die überprüfte AABB. Das Kreuz ist der Mittelpunkt der AABB. p steht für den Anfragepunkt, c für den Mittelpunkt der AABB, d steht für die Abstände der AABB-Seiten, a steht für `approx_dist_bbox`. In diesem Beispiel wird die AABB nicht übersprungen, weil $a < r$ ist. Das hellrote Quadrat zeigt die äquivalente Betrachtungsweise.

Um zu überprüfen, welche Punkte ausgeschlossen werden können, wird der *quick check whether to abort* verwendet. Dabei handelt es sich um eine Heuristik, ob die Suchkugel die aktuell betrachtete AABB überlappt. Die Vorgehensweise dafür ist in Abbildung 2.4 veranschaulicht.

Um eine AABB von der Suche ausschließen zu können, muss man ihre Distanz zum Anfragepunkt überprüfen. Das Aufwändige daran ist, herauszufinden, welcher Seitenfläche der AABB der Anfragepunkt am nähesten ist, und anschließend die Distanz zu ihr zu berechnen. Stattdessen wird in 3DTK überprüft, wie weit die Suchkugel maximal in die AABB hineinragt. Wenn festgestellt wird, dass sie hineinragen könnte, wird der Knoten zu der AABB überprüft. Andernfalls wird er übersprungen, da sich hier garantiert kein näherer Nachbar befindet.

Man kann das auch als eine Überprüfung interpretieren, ob ein Würfel um die Kugel sich mit der AABB schneidet. In Abbildung 2.4 sieht man, in welchem Spezialfall diese Vorgehensweise unnötig einen Knoten überprüft: wenn die Ecke des imaginären Würfels die Ecke der AABB schneidet, wird fälschlicherweise angenommen, dass man den Knoten dazu noch überprüfen muss.

Algorithmus 3 stellt die NNS in 3DTK in Pseudocode dar. Zuerst wird anhand von `npts` zwischen Blättern und inneren Knoten unterschieden. Alle Punkte, die sich in einem Blatt befinden,

Listing 2.2: Der `params`-Struct. Zur besseren Verständnis leicht abgeändert.

```
class KDTree {
...
// Speichert Parameter und Zwischenergebnisse der NNS und anderer Methoden
static struct {
  double p[3]; // Der Anfragepunkt
  double closest[3]; // Der beste Kandidat
  double closest_d2; // Quadrierter Abstand zwischen den beiden
    ...
} params;
...
}
```

müssen einzeln überprüft werden, ob sie näher am Anfragepunkt liegen, als der bisherige Kandidat.

Falls der aktuelle Knoten ein innerer Knoten ist, wird anhand des *quick check whether to abort* entschieden, ob er untersucht werden muss. Falls dem so ist, müssen beide Kinder rekursiv überprüft werden. Dabei wird das Kind, das vermutlich näher am Anfragepunkt ist, vorgezogen. Wenn sich dort bereits ein Kandidat gefunden hat, der näher ist, als alle Punkte im anderen Kind sein könnten, muss dieses nicht mehr überprüft werden.

2.4 Laufzeitanalyse

Wir haben nun die Algorithmen zum Konstruieren von k-d-Bäumen, für die Nächste-Nachbarn-Suche und ICP kennengelernt. Im folgenden betrachten wir die asymptotischen Laufzeiten der behandelten Algorithmen.

2.4.1 k-d-Baum-Konstruktor

Um die asymptotische Konstruktionsdauer des k-d-Baums zu erfahren, überlegen wir uns, wie viele Punkte in jedem Aufruf behandelt werden und wie lange die Operationen auf diesen Punkten dauern. Wir stellen damit eine Rekursionsgleichung für die Laufzeit auf.

Die Laufzeit für Blätter ist asymptotisch konstant, selbst wenn wir ihre Punkte in sie hineinkopieren müssen, denn ihre Größe ist immer durch die maximale Blattgröße b beschränkt und b hängt nicht von n ab. Der Basisfall der Rekursionsgleichung lautet also $T(b') = b'$ für alle $b' \leq b$.

Wenn die Punkte im Baum nach dem Median aufgeteilt werden, ist garantiert, dass die beiden rekursiven Aufrufe halb so viele Punkte behandeln müssen, wie der jetzige.

Die Operationen, die auf der aktuellen Punktemenge ablaufen, sind das Berechnen der AABB und zunächst auch das Finden des Medians der Punkte. Das Berechnen der AABB dauert lineare Zeit, nämlich für das Finden der Minima und Maxima.

Es existieren Algorithmen, um den Median in worst-case linearer Zeit zu berechnen [6]. Für diese Arbeit wird der Median mit `std::nth_element` aus der C++ STL gesucht. Dieser Algorithmus ist in der GNU STL mit dem Introselect-Algorithmus realisiert, welcher eine Worst-Case-Laufzeit in $\mathcal{O}(n \log n)$ hat [10].

Nun können wir die Rekursionsgleichungen für die theoretisch mögliche asymptotische Laufzeit und die in der Implementierung tatsächlich auftretende Laufzeit bei einer Teilung nach dem Median aufstellen.

Theoretisch ist eine Laufzeit von $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$ möglich. Diese Rekursionsgleichung lässt sich mit der Meistermethode [6] durch $\mathcal{O}(n \log n)$ beschränken. Die theoretisch mögliche Worst-Case-Laufzeit liegt also in $\mathcal{O}(n \log n)$.

Die Rekursionsgleichung für den implementierten Algorithmus lautet $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n \log n)$. Die mit der Meistermethode ermittelte Schranke hierfür liegt bei $\mathcal{O}(n \log^2 n)$. Sie ist also um einen logarithmischen Faktor asymptotisch schlechter als die mögliche Laufzeit.

Bei der Teilung am Mittelpunkt der AABB oder nach dem Durchschnitt spart man sich die Suche nach dem Median, hat aber keine Garantie mehr, dass die Punktemenge hälftig aufgeteilt wird. Tatsächlich kann man für beide Strategien Beispiele konstruieren, die eine asymptotisch quadratische Laufzeit hervorrufen. Das erreicht man, indem man den Algorithmus zwingt, einen degenerierten Baum zu konstruieren, der einer verketteten Liste gleicht. Er kann dann pro Aufruf nur einen einzigen Punkt in ein Blatt verpacken und braucht deshalb im nächsten Aufruf $\Theta(n)$ Schritte, um die AABB und evtl. den Durchschnitt der $n - 1$ Punkte zu berechnen. Wie ein solches Beispiel aussehen muss, haben wir schon ansatzweise bei der lilafarbenen AABB in Abbildung 2.2a gesehen. Man muss diese nur in sich selbst verschachteln, um ein Beispiel zu erhalten, das eine schlechte Aufteilung erzwingt.

Im schlechtesten Fall hat die Aufteilung nach der Hälfte oder dem Durchschnitt also eine quadratische Laufzeit. Die Datensätze, die in der Praxis vorkommen, unterscheiden sich natürlich weit von den schlechtestmöglichen. Bei den hier durchgeführten Experimenten werden auch Datensätze mit gleichverteilten Punkten verwendet, welche wiederum der Bestfall für diese beiden Aufteilungsstrategien sind. Sie teilen dann nämlich die gleichverteilte Punktemenge erwartet hälftig auf und haben deshalb nur eine Laufzeit in $\mathcal{O}(n \log n)$.

2.4.2 Nächste-Nachbarn-Suche

Um die Laufzeit der Nächste-Nachbarn-Suche zu analysieren, müssen wir auf eine geometrische Betrachtung des Problems zurückgreifen. Grundlegend interessiert uns, wie viele Knoten wir besuchen, weil der Suchradius nicht ausschließt, dass sich dort noch ein näherer Nachbar befindet.

In der Analyse von Friedman et al. [9] wird die Argumentation mit Wahrscheinlichkeiten geführt. So können sie angeben, wie groß der erwartete Abstand r zum nächsten Nachbarn ist.

Bei Aufteilung nach dem Median entlang der längsten AABB-Seite nähern sich die Formen der AABBs bei zunehmender Tiefe im Baum dem Würfel. Friedman et al. [9] können deswegen auch eine erwartete Seitenlänge s der Blätter angeben.

Nimmt man diese beiden Größen zusammen, dann kann man die erwartete Anzahl an Blättern l, die sich mit der Kugel um den Anfragepunkt mit Radius r schneiden, von oben beschränken. Dazu berechnet man die Anzahl an Blättern L, die den Würfel um die Kugel schneiden. Dies wird folgendermaßen ausgedrückt:

$$l \leq L = (\frac{2r}{s} + 1)^3$$

Die wichtigste Erkenntnis daraus ist, dass l nicht von der Anzahl an Punkten abhängt. Die Ursache dafür erklären Friedman et al. [9] damit, dass die Kugel und die Blätter sich „ähneln“. Sie enthalten eine begrenzte Anzahl an Punkten (nämlich höchstens respektive zwei und die

Blattgröße) und sind räumlich kompakt. Wenn die Anzahl an Punkten wächst, schrumpfen die erwartete Seitenlänge der Blätter s und der Abstand zum nächsten Nachbarn r gemeinsam.

Die Anzahl an betroffenen Blättern l hilft uns nun, die Laufzeit zu finden. Um zu den l Blättern zu gelangen, braucht man in einem balancierten k-d-Baum mit n Punkten maximal $\mathcal{O}(l \log n)$ Zeit. In den l Blättern müssen jeweils maximal b, also insgesamt maximal $l \cdot b$ Punkte überprüft werden. Da weder l noch b von n abhängen, ergibt sich bei Friedman et al. [9] eine erwartete Laufzeit der Nächste-Nachbarn-Suche in $\mathcal{O}(\log n)$.

Die Blattgröße ändert also asymptotisch nichts an der Laufzeit der NNS. Wie wir sehen werden, ist sie praktisch trotzdem ein interessanter Parameter.

2.4.3 Iterative Closest Point

Der ICP-Algorithmus sucht, wie beschrieben, in jeder Iteration zu jedem Punkt des Modelldatensatzes den nächsten Nachbarn im Zieldatensatz. Wenn m die Größe des Modelldatensatzes und d die Größe des Zieldatensates ist, dann dauert eine Iteration asymptotisch $\mathcal{O}(m \log d)$ Zeit.

Da man in der Praxis die Anzahl der Iterationen, die gemacht werden, meist beschränkt, ist es nicht sinnvoll, hier die Anzahl der nötigen Iterationen für zwei gegebene Datensätze zu ermitteln. Es gibt jedoch Publikationen, die versuchen, die asymptotische Laufzeit von ICP zu beschränken [2].

Algorithmus 2 : KDTree(double $**P$, int $left$, int $right$, unsigned int b, $s \in$ {halb, durchschnitt, median})

Eingabe : Feld P mit dreidimensionalen Punkten, Beginn des Teilfelds $left$, Ende des Teilfelds $right$, maximale Blattgröße b, Partitionsstrategie s

```
// Kleine Definition für die Lesbarkeit
```
$Q \leftarrow P[left : right]$
```
// Abbruchbedingung
```
if $right - left > b$ **then**
```
    // Berechne die AABB von Q.
```
$min \leftarrow (\min_x Q, \min_y Q, \min_z Q)$;
$max \leftarrow (\max_x Q, \max_y Q, \max_z Q)$;
```
    // Ermittle den Mittelpunkt
```
for $i = 0$ **to** 2 **do** $node.center[i] \leftarrow (min[i] + max[i])/2$;
```
    // Ermittle die halben Seitenlängen
```
for $i = 0$ **to** 2 **do** $node.d[i] \leftarrow (max[i] - min[i])/2$;
```
    // Finde die längste Achse der AABB
```
$node.splitaxis \leftarrow \arg\max_{i \in \{0,1,2\}} node.d[i]$;
```
    // Bestimme die Position der Aufteilung
```
switch s **do**
    case *halb* **do**
        $node.splitval \leftarrow node.center[node.splitaxis]$;
    case *durchschnitt* **do**
        $node.splitval \leftarrow \sum_{p \in Q} p[node.splitaxis]/(right - left)$;
    case *median* **do**
        $node.splitval \leftarrow$ Median($\{p[node.splitaxis] \mid p \in Q\}$);
```
    // Partitioniere die Punkte
```
$mid \leftarrow$ Partition($P, left, right, node.splitval$);
```
    // Konstruiere rekursiv die Kinder
```
$node.child1 \leftarrow$ KDTree($P, left, mid, b, s$);
$node.child2 \leftarrow$ KDTree($P, mid, right, b, s$);
return *this*;
else
```
    // Es verbleiben b Punkte in Q
```
$leaf.p \leftarrow Q$;
return *this*;

Algorithmus 3 : KDTree.NNS()

Eingabe : Der Anfragepunkt in *params.p*, die maximale Entfernung quadriert in *params.closest_d2*

Ausgabe : Der nächste Nachbar von *params.p* in *params.closest*, deren quadrierter Abstand in *params.closest_d2*

```
// Unterscheidung zwischen Blatt und innerem Knoten
1  if npts > 0 then
       // In einem Blatt muss jeder Punkt einzeln überprüft werden
2      for i = 0 to npts − 1 do
           // Quadrierte Distanz zum Anfragepunkt
3          current_d2 ← Σ_{j=0}^{2} (leaf.p[i][j] − params.p[j])^2;
4          if current_d2 < params.closest_d2 then
               // Ein näherer Punkt wurde gefunden
5              params.closest_d2 ← current_d2;
6              params.closest ← leaf.p[i];
7  else
       // Quick check whether to abort (s. Abbildung 2.4)
8      approx_dist_bbox = max_{i∈{0,1,2}} |params.p[i] − node.center[i]| − node.d[i];
9      if approx_dist_bbox < 0 or approx_dist_bbox^2 < params.closest_d2 then
           // Wir können den Knoten nicht ausschließen
           // Bestimme die Distanz zur Trennachse
10         current_d ← node.splitval − params.p[node.splitaxis];
           // Überprüfe zuerst das Kind, das dem Anfragepunkt am nähesten ist
11         if current_d ≥ 0 then
12             node.child1.NNS();
               // Eventuell muss das ferne Kind nicht mehr überprüft werden
13             if current_d^2 < params.closest_d2 then
14                 node.child2.NNS();
15         else
16             node.child2.NNS();
17             if current_d^2 < params.closest_d2 then
18                 node.child1.NNS();
```

Kapitel 3

Versuchsaufbau

Es wurde die Laufzeit der k-d-Baum-Konstruktion und die des ICP-Algorithmus für die verschiedenen Aufteilungsstrategien (AABB-Mittelpunkt, Durchschnitt, Median), für verschiedene Blattgrößen (1-100), und jeweils auf verschiedenen Datensätzen, die teilweise zufällig generiert sind, gemessen.

Die Messungen wurden mit einem AMD FX-8320 bei 4.1 GHz durchgeführt. Diese Generation von AMD-Prozessoren verfügt über vier FPUs, die parallel arbeiten können [16]. Der Arbeitsspeicher sind 8GB DDR3-1666 RAM. 3DTK wurde mit GCC 5.4.0 mit `-O3` kompiliert. Diese Optimierungsstufe ist in 3DTK standardmäßig aktiviert und wirkt sich nicht auf die Genauigkeit der Algorithmen aus [14]. Das Betriebssystem ist Linux 4.4.0.

Die Messungen wurden mit möglichst hoher Scheduling-Priorität (`nice -n -20`) durchgeführt, um Interferenz von anderen Prozessen zu vermeiden.

3.1 Zeitmessung

Das Programm, das in 3DTK den ICP-Algorithmus durchführt, heißt *SLAM6D – simultaneous localization and mapping* mit 6 Freiheitsgraden (3 Dimensionen Verschiebung und 3 Winkel Drehung). SLAM6D hat bereits eingebaute Routinen, um die Zeitdauer aller Abschnitte des Programms, vom Laden der Scans bis zur ICP-Durchführung, zu messen. Dabei wird die angegebene ICP-Dauer als die Laufzeit des ganzen Programms ermittelt, von der die Laufzeiten der anderen Teile, wie dem Einlesen der Scan-Dateien, abgezogen werden. Die gemessene Zeit ist also frei von Input-Output-Operationen und tatsächlich auf die Dauer des ICP-Algorithmus beschränkt. Sie ist deshalb wesentlich durch die Interaktion von Prozessor, Cache und Arbeitsspeicher bestimmt.

Die von SLAM6D selbstgemessenen Zeiten schreibt es in seine Standardausgabe. Aus dieser wird von einem Python-Skript mit regulären Ausdrücken die gemessene Zeit geparst. Durch die zwischenzeitliche Repräsentation der Zeitmessung als String statt als Fließkommazahl geht ein Teil der Präzision verloren. Standardmäßig gibt die C++-Standardbibliothek eine Fließkomma-

zahl als String mit sechs gültigen Stellen aus. Der Verlust an Präzision ist deshalb bei den hier auftretenden Messungen unerheblich.

Um eine repräsentative Zeitmessung zu erhalten, werden für jede Kombination aus Datensatz und Parameter mehrere Messungen durchgeführt. Standardmäßig sind das nur zehn Messungen, aber das Python-Skript ist so geschrieben, dass es noch beliebig viel mehr Messungen durchzuführen kann. Dadurch kann, je nach verfügbarer Zeit, das Messergebnis beliebig verfeinert werden. Die hier präsentierten Zeiten ergeben sich aus ungefähr einem Tag an Messungen. Die exakte Zahl der Messungen wurde nicht ermittelt.

Der ICP-Algorithmus arbeitet deterministisch, also sollte jeder Aufruf mit den gleichen Parametern auch die gleiche Zeit dauern. Da jedoch das Betriebssystem-Scheduling und andere Störfaktoren immer eine unmessbare Extradauer bei der Zeitmessung dazufügen, ist es sinnvoll, das Minimum mehrerer Zeitmessungen als endgültigen Messwert herzunehmen. Nähme man den Durchschnitt oder den Median, würde man viel mehr messen, was das System sonst gerade beschäftigt. Die ermittelten Zeiten stehen also für die beste Dauer, die trotz Störfaktoren erreicht wurde.

Die Resultate der Zeitmessungen werden als GNUplot-kompatible Textdateien gespeichert. Mit GNUplot-Skripten wurden daraus die hier präsentierten Graphen erstellt. Ein Makefile automatisiert alle diese Prozesse. Dadurch können die Ergebnisse sehr leicht reproduziert werden und die Testprogramme können mit leichten Veränderungen auch für andere Benchmarks verwendet werden.

Außerdem sei darauf hingewiesen, dass die Nächste-Nachbarn-Suche in 3DTK multithreaded ist, die Konstruktion des k-d-Baums jedoch nicht. Das liegt daran, dass die existierende parallele Implementierung der k-d-Baum-Konstruktion langsamer ist als die sequentielle Implementierung. Ob das Problem grundlegend im Parallelisierungs-Overhead liegt oder andere Gründe hat, wurde nicht untersucht.

Die präsentierten Zeiten für ICP beinhalten also die Nutzung der 4 FPUs des Testprozessors, die Zeiten der Konstruktion aber nicht. Die relativen Zeiten von ICP und der Konstruktion sind jedoch unerheblich, denn bei der praktischen Verwendung von SLAM6D wird der k-d-Baum pro Scan genau einmal konstruiert, der ICP-Algorithmus läuft je nach Datensatz aber unterschiedlich oft. Bei den hier durchgeführten Benchmarks findet nur ein ICP-Aufruf statt.

3.1.1 Zeitmessung für unterschiedliche Blattgrößen

Es wurde die Auswirkung der erlaubten Maximalzahl an Punkten pro Blatt (*Blattgröße*) auf die Laufzeit des ICP-Algorithmus (Matching) und der k-d-Baum-Konstruktion untersucht.

Dazu wurde zunächst eine Zeitmessung für jede Blattgröße von 1-65 angesetzt. Bei den größeren Messwerten war ein Aufwärtstrend in der Matching-Zeit zu erkennen. Um diesen zu bestätigen, wurde von 65-100 in Fünferschritten die Zeit gemessen.

3.1.2 Zeitmessung für unterschiedliche Partitionsstrategien

Es wurde die Auswirkung der Positionierung der aufteilenden Ebene (*Partitionsstrategie*) auf die Laufzeit des ICP-Algorithmus und der k-d-Baum-Konstruktion untersucht.

Dazu wurde die Laufzeit für die bisherige Partition nach dem Mittelpunkt der AABB sowie nach dem Durchschnitt der Punkte und nach dem Median gemessen. Der Median wurde auf zwei minimal unterschiedliche Arten implementiert, für die je eine Messung angestellt wurde. Der Grund dafür wird in Abschnitt 3.2 erklärt.

3.2 Änderungen an SLAM6D

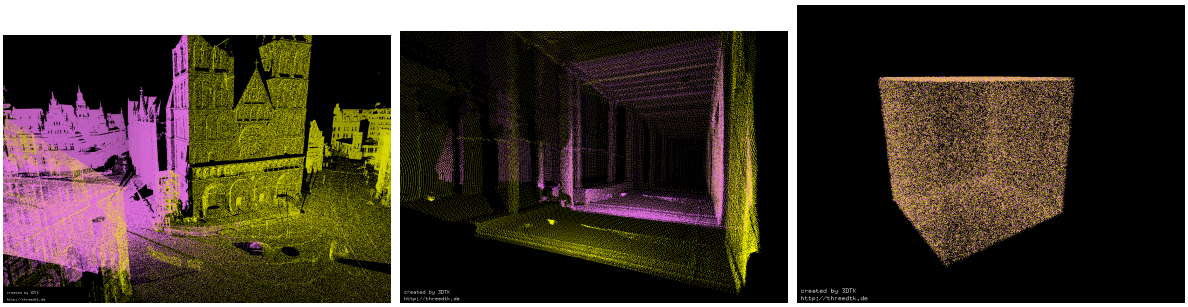
Für diese Arbeit musste der Quellcode von SLAM6D, dem Programm, das in 3DTK den ICP-Algorithmus ausführt, angepasst werden. Besonderes Ziel dieser Arbeit war, dass der produzierte Quellcode auch sofort wieder ins 3DTK übernommen werden kann.

Wie erwähnt verfügt SLAM6D über eingebaute Routinen zur Zeitmessung. Diese Routinen haben `gettimeofday` verwendet, um die aktuelle Zeit in Millisekunden seit der *Epoche* (01.01.1970) zu ermitteln. Die Zeit, die `gettimeofday` zurückgibt, hängt von der Systemzeit ab und ändert sich, wenn sich zwischen zwei Aufrufen die Systemzeit ändert, z.B. indem sie über NTP (Netzwerkzeitprotokoll) automatisch synchronisiert wird. Deshalb kann man nicht ordentlich eine Zeitdauer durch die Differenz von zwei `gettimeofday`-Aufrufen messen. Für diese Arbeit wurde das korrigiert, indem stattdessen `clock_gettime(CLOCK_MONOTONIC, ...)` verwendet wird.

Die Maximalzahl an erlaubten Punkten pro Blatt war auf zehn festgesetzt. Sie wurde als Kommandozeilenparameter zur Konfiguration freigegeben, damit sie leicht zugänglich für das Benchmarking-Programm ist. Um sie konfigurierbar zu machen, muss der gewünschte Wert von der Kommandozeile bis zur Konstruktionsmethode als Funktionsparameter durchgereicht werden, so wie in Algorithmus 2 angedeutet. Die anderen Methoden des k-d-Baums in 3DTK funktionieren schon ohne Veränderung mit dynamischen, bzw. zur Laufzeit festgelegten Blattgrößen.

Um die Partitionsstrategie, also die Position der aufteilenden Ebene in der Punktemenge, zu implementieren, mussten mehr Änderungen vorgenommen werden. Die gewünschte Strategie wird auch hier von der Kommandozeile bis zur Konstruktions-Methode als Funktionsparameter durchgereicht. Bei der Konstruktion wird die Punktemenge wie gewünscht aufgeteilt. Diese Herangehensweise erlaubt auch, dynamisch zu entscheiden, welche Strategie verwendet werden soll. Dadurch könnte z.B. auf den oberen Ebenen eine zeitgünstigere Strategie (Halbieren der AABB und Durchschnitt) verwendet werden, und auf den unteren eine balancierendere (Median). Es ist eine Aufteilung am Mittelpunkt der AABB, dem Durchschnitt der Punkte, und dem Median mit Fallback auf je eine der beiden anderen Aufteilungen möglich.

Der Grund dafür ist ein Problem beim Aufteilen nach dem Median: Wenn bei den behandelten Punkten in der behandelten Komponente sehr oft der gleiche Wert vorkommt, kann man die Punkte nicht sinnvoll nach dem Median aufteilen. Ein Beispiel: der Median von $[1, 1, 2]$ ist 1.



(a) Bremer Innenstadt **(b)** Büroflur **(c)** Würfelflächen

Abbildung 3.1: Testdatensätze

Versucht man nun, alle Werte nach „größer gleich dem Median“ und „kleiner als der Median“ aufzuteilen, stellt man fest, dass es in diesem Beispiel keine Werte kleiner als der Median gibt. Beim Aufteilen in die Zweige des k-d-Baums würde man also einen leeren Zweig produzieren, was nicht mit unseren Definitionen eines k-d-Baums vereinbar ist.

Das tritt immer genau dann ein, wenn der Median gleich dem Minimum ist. In einem solchen Fall werden die Punkte nicht durch den Median aufgeteilt, sondern auf eine der beiden anderen Arten. Um voreilige Schlüsse zu vermeiden, wurden beide Fallbackmechanismen getestet. Wie oft genau ein solcher Regressionsfall eintritt, wurde nicht untersucht. Er trat jedoch oft genug ein, um hier erwähnt zu werden.

Dieses Problem tritt nicht bei den anderen beiden Partitionsstrategien auf, da hier garantiert ein Wert kleiner dem Partitionierungswert existiert, da sonst alle Werte gleich sein müssten. Dieser Fall wurde schon vor dieser Arbeit in SLAM6D behandelt, indem alle Knoten mit ausreichend kleiner Umkugel – also insbesondere, wenn alle Punkte gleich sind – zu einem Blatt gemacht werden.

Zuletzt waren noch die restlichen Methoden des k-d-Baums außer der Konstruktion von der Wahl der Aufteilung abhängig. Sie nahmen an, dass der Wert der Aufteilung sich am Mittelpunkt der AABB befindet. Das manifestierte sich in Segfaults bei der Nächste-Nachbarn-Suche. Der Wert der Aufteilung wird nun im Knoten gespeichert und die Methoden wurden so geändert, dass sie den gespeicherten Wert auslesen, anstatt ihn anhand des Mittelpunkts anzunehmen.

3.3 Datensätze

Als Testdatensätze werden vier zufällig generierte geometrische Formen verwendet, sowie zwei echt gemessene Datensätze. Abbildung 3.1 veranschaulicht die Testdatensätze.

Bei den zufällig generierten Formen handelt es sich um den Würfel und die Kugel, jeweils einmal als gefüllte Form und als ihre Außenfläche(n). Sie bestehen alle aus 60000 Punkten mit einem Durchmesser bzw. einer Seitenlänge von 100 Einheiten. Es wurden gleichverteilt Punkte generiert und anschließend einmal kopiert und jeweils zufällig um bis zu eine Einheit in jede

Richtung verschoben. Das Resultat sind je zwei „Scans“, in denen nächste Nachbarn höchstens $2\sqrt{3}$ Einheiten auseinanderliegen.

Die echten Scans stammen von einem Büroflur und der Innenstadt Bremens. Der Büroflur-Datensatz wird mit 3DTK mitgeliefert. Er wurde mit einem SICK LMS200, einem planaren Laserscanner mit geringem Sichtfeld, aufgenommen, der nach oben und unten geschwenkt wurde, um die Scans aufzunehmen. Die resultierenden Scans weisen deshalb viele Linien an Punkten auf. Sie sind jeweils ungefähr 81000 Punkte groß. Der Bremer Datensatz wurde von einem RIEGL VZ-400 aufgenommen. Die meisten nächsten Nachbarn finden sich in diesem Datensatz am Boden und an den Kanten des Doms und der Häuser gegenüber.

Diese Datesätze wurden gewählt, weil Elseberg et al. [8] mit diesen Datensätzen (und einem weiteren) erstmals verschiedene NNS-Implementierungen im Hinblick auf ihre Performance getestet haben. Von jedem Datensatz sind nur zwei Scans nötig, um ICP durchzuführen. Es wurden jeweils die ersten beiden nach ihrer Nummerierung genommen.

Die Bremer Scans sind so groß, dass es sehr zeitaufwändig ist, mehrere Zeitmessungen an ihnen durchzuführen. Die Anzahl an Punkten in ihnen wurde deshalb verringert. Das geschieht, indem ein Octree über der Punktemenge aufgebaut wird, mit einer minimalen Kantenlänge von 5 cm der Blätter. Alle Punkte, die sich dann in einem Blatt befinden, werden zusammengefasst. Nimmt man den Mittelpunkt oder einen zufälligen Punkt aus jedem Blatt als Repräsentanten, hat man eine deutlich kleinere Anzahl an Punkten zu behandeln. Für diese Arbeit wurden beide Verfahren verwendet. Es war unklar, ob die Regelmäßigkeit durch die Wahl des Mittelpunktes oder der Zufall die Messergebnisse verfälschen können. Nach der Reduktion beinhalten sie immernoch je fast 4,9 Millionen Punkte.

Wie bereits angemerkt, sind gleichverteilte Punkte, wie sie bei den künstlichen Datensätzen vorkommen, die optimalen Eingaben für eine schnelle Konstruktion bei der Partitionierung am Mittelpunkt der AABB oder dem Durchschnitt. Beim Interpretieren der Ergebnisse sollte darauf geachtet werden.

Kapitel 4

Ergebnisse

Hier werden die Ergebnisse der beschriebenen Messungen präsentiert. Sie sind nach Messungen der Performance bei unterschiedlichen Blattgrößen und unterschiedlichen Partitionsstrategien gegliedert.

Es werden keine Abbildungen präsentiert, bei denen eine Summe aus Konstruktionszeit und Matching-Zeit gebildet wird. Der Grund dafür liegt darin, dass eine solche Abbildung irreführend wäre: der k-d-Baum wird pro Datensatz nur einmal erstellt, der ICP-Algorithmus iteriert aber unbekannt oft, abhängig vom Datensatz. Man könnte daraus also keine nutzbaren Schlüsse für die Praxis ziehen.

4.1 Blattgröße

Die hier präsentierten Ergebnisse (Abbildungen 4.1-4.8) zeigen, wie sich das Zusammenfassen mehrerer Punkte in einem Blatt im k-d-Baum auf die Laufzeit des ICP-Algorithmus und der k-d-Baum-Konstruktion auswirkt. Als Blattgröße ist die maximale Anzahl an Punkten bezeichnet, die in einem Blatt zusammengefasst werden dürfen.

Wie in Abschnitt 3.1.1 beschrieben, wurde je für Blattgrößen von 1-65 und in Fünferschritten für Blattgrößen von 65-100 eine Messung angesetzt. Sie wurden für eine bessere Übersicht mit einer Linie verbunden.

Bei den Messergebnissen für den Flur (Abb. 4.5a) ist die Skala der y-Achse zu beachten. Der Graph sieht sehr zufällig aus, weil er nicht, wie die anderen Graphen, durch einen extrem hohen Wert für eine Blattgröße von eins enger skaliert wird.

Bei allen Datensätzen ist deutlich zu erkennen, dass zu kleine Blattgrößen sowohl eine hohe Matching-Laufzeit als auch eine hohe Konstruktions-Laufzeit zur Folge haben.

Bei allen Datensätzen ist für die Konstruktionszeit ein ungefähr hyperbolischer Abstieg mit steigender Blattgröße zu erkennen.

Die Spitze in der Matching-Zeit bei kleinen Blattgrößen ist bei allen Datensätzen bereits bei einem Wert von zehn abgeflacht. Nach dem Tal in den Messwerten zwischen 10 und ungefähr 25 ist bei allen Datensätzen ein ungefähr linearer Anstieg der Matching-Zeit erkennbar.

Interessant ist auch, dass beide Bremer Datensätze (Abb. 4.6a, Abb. 4.7a) ihr Minimum bei einer Blattgröße von 20 haben. Allgemein lässt sich nicht genau sagen, welcher Wert optimal für die Matching-Dauer ist. Der Kurvenverlauf deutet aber an, dass irgendwo zwischen 10 und 25 der beste Wert bei allen getesteten Datensätzen liegt.

In Abbildung 4.8 sind die gemessenen Zeiten für das Matching von allen Datensätzen dargestellt. Von den jeweiligen Messungen wurde das Minimum ermittelt und die Messwerte an ihm normalisiert. Dadurch ist ersichtlich, wie viel länger die anderen möglichen Blattgrößen auf einem Datensatz brauchen und wie sich das Verhalten der Datensätze zueinander gegenüber dem Optimum unterscheidet.

Wenn man die Zeiten für den Wert zehn betrachtet, der die aktuelle Standardwahl für die Blattgröße ist, kann man erkennen, um wie viel langsamer diese Wahl als die beste gemessene ist.

Besonders zu erkennen ist, dass sich bei allen echten Datensätzen eine kleine Blattgröße viel weniger auswirkt, als bei den zufällig generierten.

4.2 Partitionsstrategien

Die hier präsentierten Ergebnisse (Abblidungen 4.9-4.15) zeigen, wie sich die Wahl der Partition, also die Platzierung der Trennebenen, auf die Laufzeit des ICP-Algorithmus und der k-d-Baum-Konstruktion auswirkt. Die beiden Fallbacks des Medians auf Halbierung der AABB und den Durchschnitt sind respektive mit „Median H.“ und „Median D.“ abgekürzt.

Hier fällt besonders auf, wie viel mehr Zeit die Konstruktion mit dem Median dauert. Vergleicht man diese Zeiten mit der Ersparnis, die der Median für das Matching bringt, muss man feststellen, dass er sich nur wenig lohnt. Das Matching wird bei den echten Datensätzen und bei der gefüllten Kugel durch den Median schneller als mit der Standardstrategie.

Die Berechnung des Durchschnitts scheint sich aber noch mehr zu lohnen. Das Matching ist auf allen Datensätzen außer den Würfelflächen (Abb. 4.10a) mit ihm schneller als mit der Standardstrategie. Bei der Konstruktion fügt er nur einen kleinen Overhead hinzu. Beim Flurdatensatz (Abb. 4.13b) läuft er sogar schneller als das Halbieren der AABB. Die Gründe dafür können entweder durchgängig unglückliche Messungen für die Halbierung beim Flurdatensatz oder eine glücklicherweise balancierte Aufteilung der Punkte sein, wodurch er einen flacheren Baum produziert.

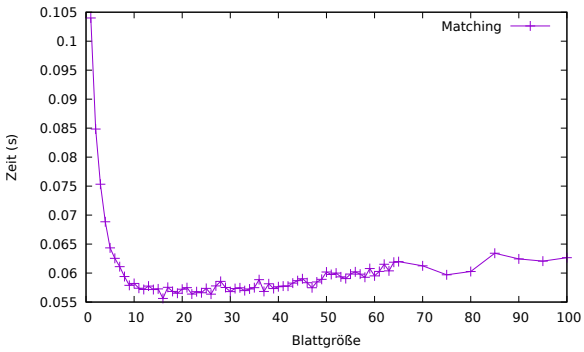
Den Overhead der Durchschnittsbildung bei der Konstruktion kann man zusätzlich womöglich eliminieren, indem man gleichzeitig mit der AABB-Berechnung den Durchschnitt in allen Achsen berechnet. Es ist nötig, ihn in allen Achsen zu berechnen, weil man noch nicht die längste Achse kennt. Dadurch, dass man das Punkte-Array nur einmal durchläuft, erlangt man eine bessere

Cache-Kohärenz, als wenn man es später noch einmal durchläuft. Selbst wenn man dann dreimal so oft den Durchschnitt berechnen muss, kann sich das lohnen.

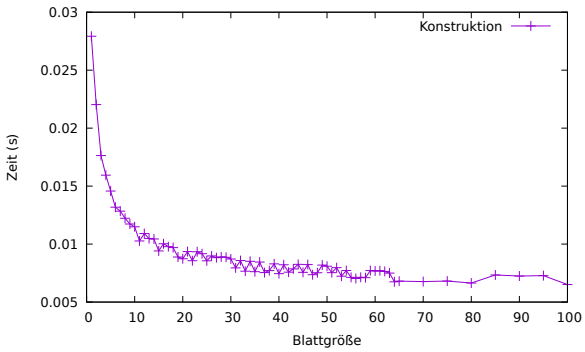
4.3 Folgerungen

Wir haben gesehen, dass die Wahl der Blattgröße sich stark auf die Konstruktionsdauer und Matching-Dauer auswirken kann. Wenn man aber einen Wert von 10 bis 25 wählt, leidet man weder unter dem enormen Aufwand, dank zu kleiner Blätter zu viele Pointer verfolgen zu müssen, noch leidet man unter der linearen Suche durch viele Punkte in großen Blättern. Da wir uns einen allgemeingültigen Wert für die beste Blattgröße wünschen, müssen wir aus diesem Bereich einen auswählen. Elseberg et al. [8] haben auf dem Flurdatensatz einen Wert von 15 als besten Wert ermittelt, sie haben jedoch in diesem Bereich nur in Fünferschritten gemessen. Der Verlauf der Kurven bei den hier aufgeführten Messungen lässt einen ähnlichen Wert als den besten vermuten.

Was die Partitionsstrategie betrifft, lohnt es sich nicht, den Median der Punkte zu berechnen. Statt dessen ist der Durchschnitt ein vielversprechender Kandidat. Wie beschrieben lässt er sich in der Konstruktionsdauer vermutlich noch verbessern. Er kann dann dauerhaft in 3DTK als bessere Strategie zum Partitionieren integriert werden.

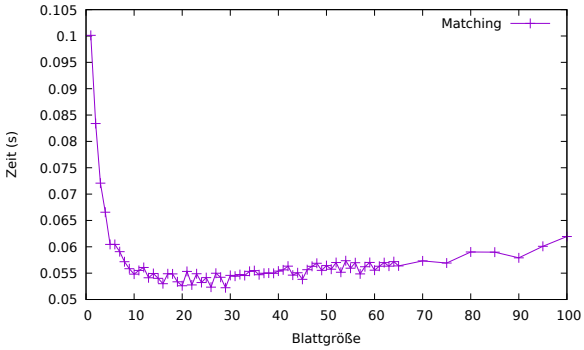


(a) Matching

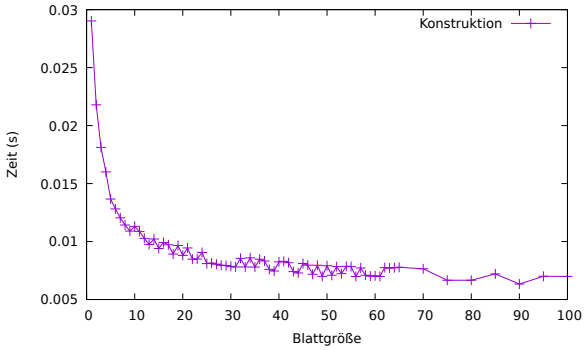


(b) Konstruktion

Abbildung 4.1: Gefüllter Würfel

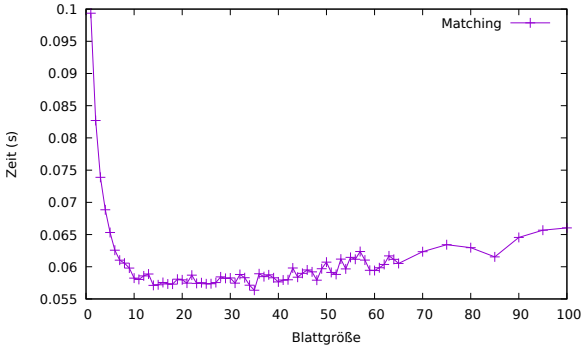


(a) Matching

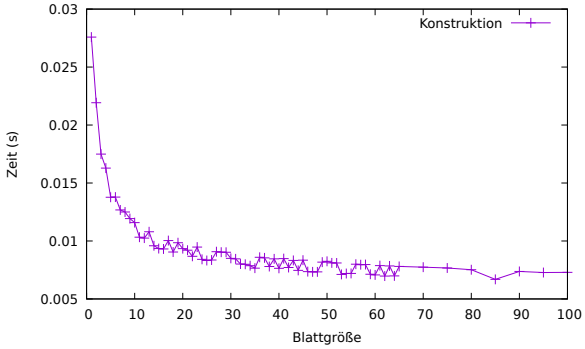


(b) Konstruktion

Abbildung 4.2: Würfelflächen

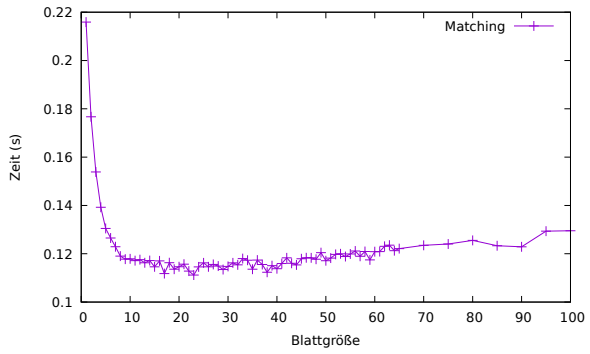


(a) Matching

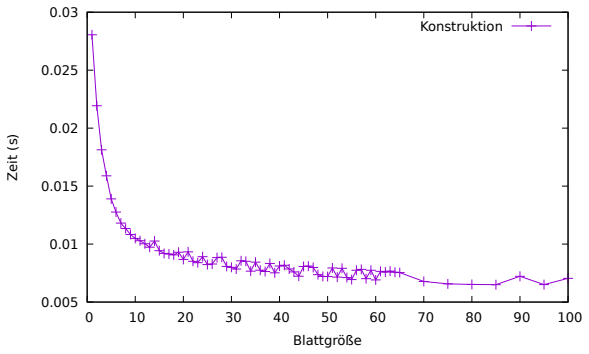


(b) Konstruktion

Abbildung 4.3: Gefüllte Kugel

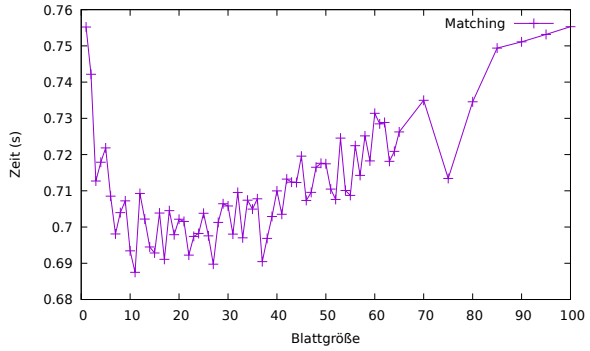


(a) Matching

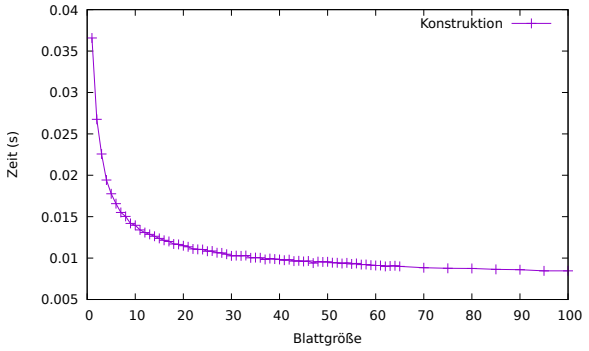


(b) Konstruktion

Abbildung 4.4: Kugelrand

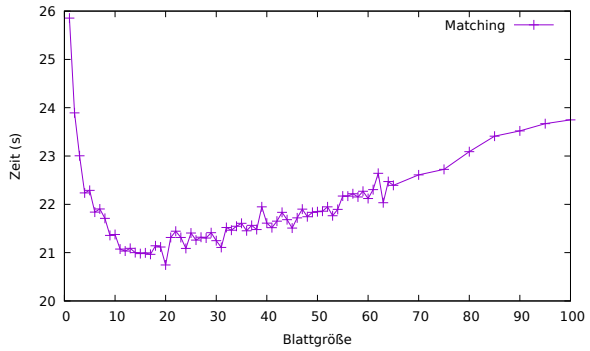


(a) Matching

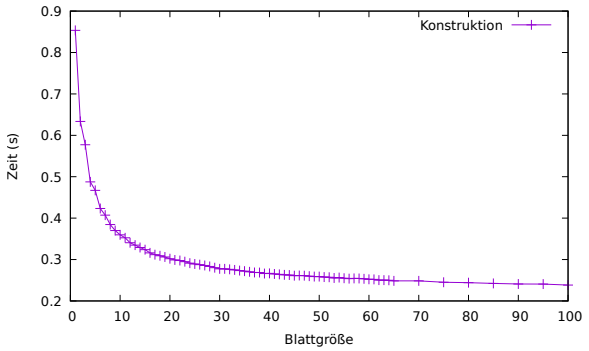


(b) Konstruktion

Abbildung 4.5: Flur

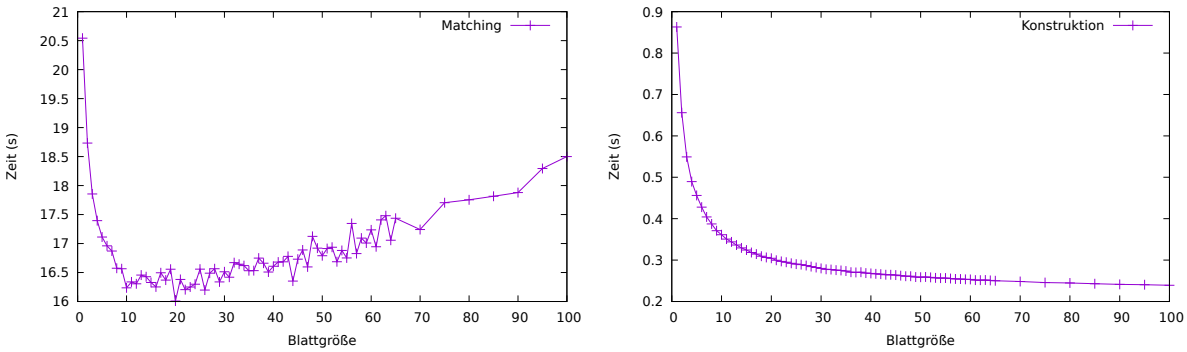


(a) Matching



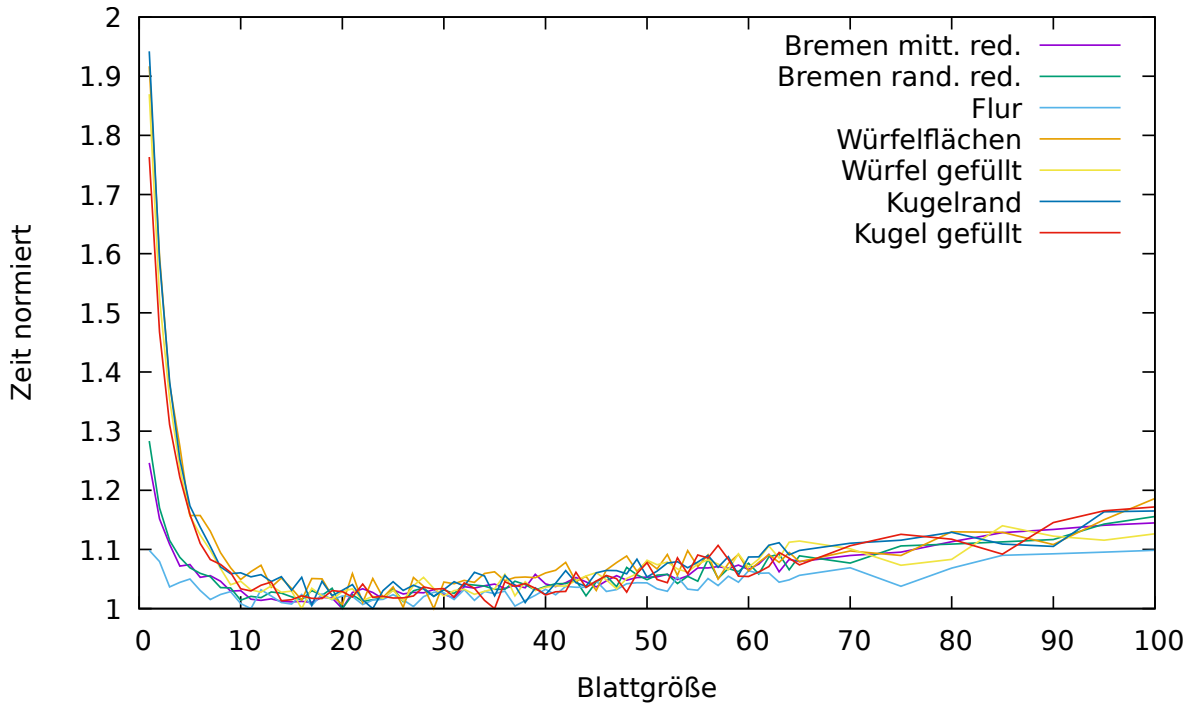
(b) Konstruktion

Abbildung 4.6: Bremen auf Mittelpunkte reduziert

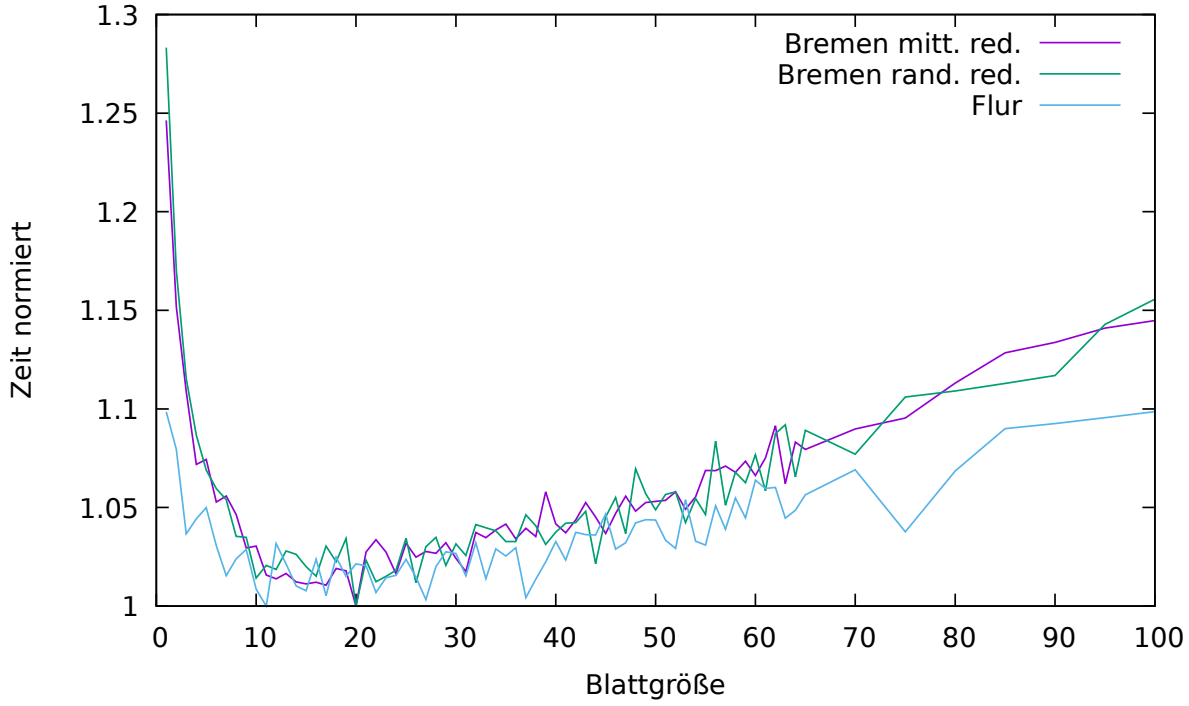


(a) Matching **(b)** Konstruktion

Abbildung 4.7: Bremen randomisiert reduziert

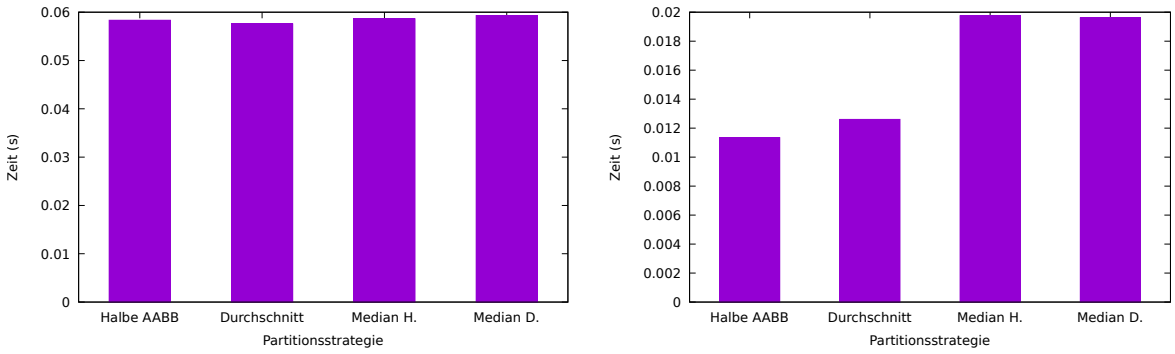


(a) Alle Datensätze



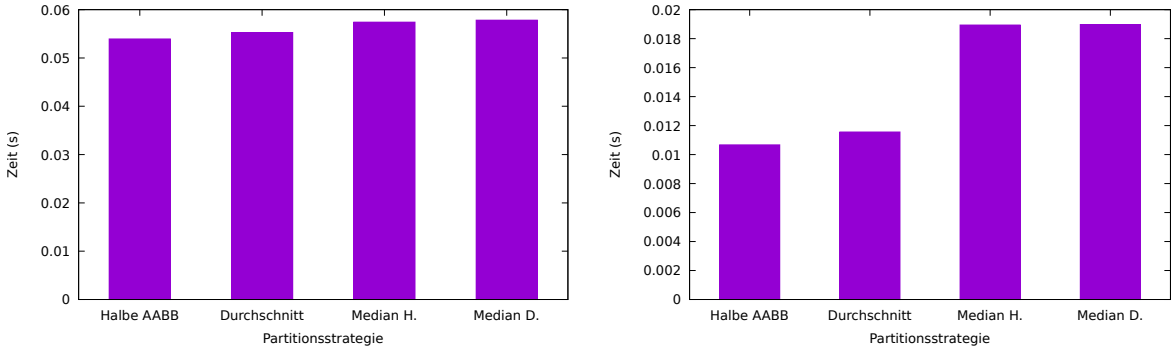
(b) Ohne künstliche Datensätze

Abbildung 4.8: Am Minimum normalisierte Zeiten für das Matching



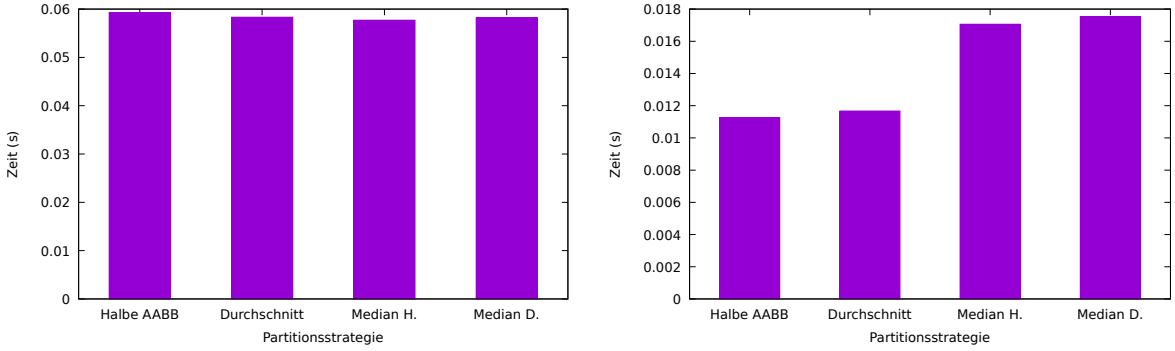
(a) Matching **(b)** Konstruktion

Abbildung 4.9: Gefüllter Würfel



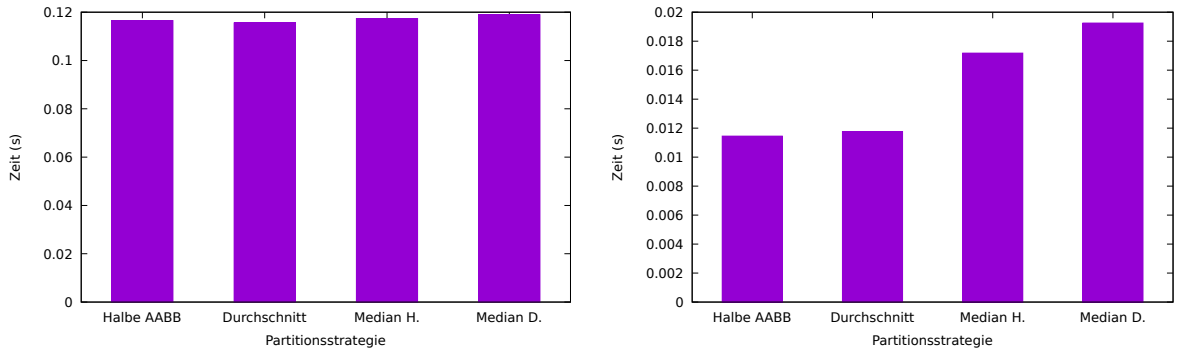
(a) Matching **(b)** Konstruktion

Abbildung 4.10: Würfelflächen



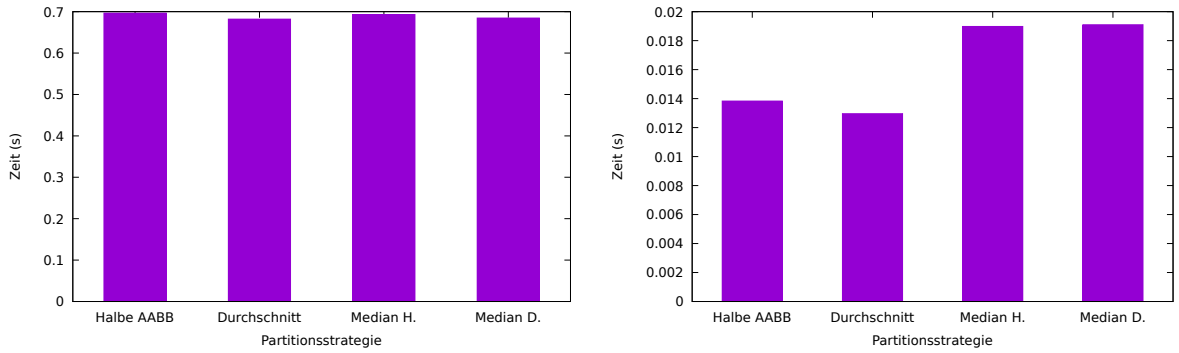
(a) Matching **(b)** Konstruktion

Abbildung 4.11: Gefüllte Kugel



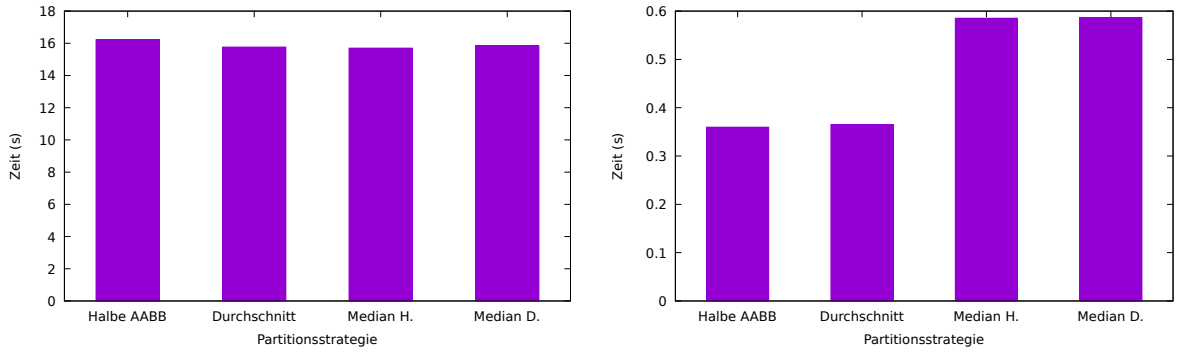
(a) Matching **(b)** Konstruktion

Abbildung 4.12: Kugelrand



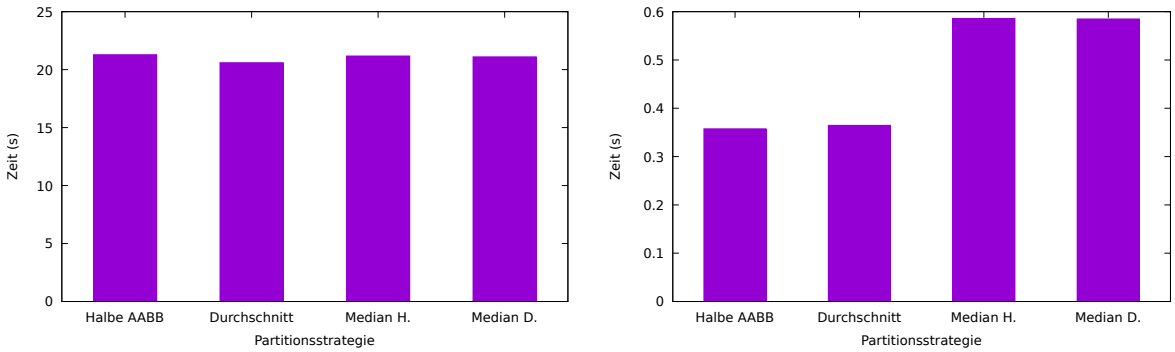
(a) Matching **(b)** Konstruktion

Abbildung 4.13: Flur



(a) Matching **(b)** Konstruktion

Abbildung 4.14: Bremen auf Mittelpunkte reduziert



(a) Matching **(b)** Konstruktion

Abbildung 4.15: Bremen randomisiert reduziert

Kapitel 5

Zusammenfassung und unbehandelte Probleme

Wir haben den k-d-Baum untersucht, so wie er im Lehrbuch beschrieben wird, und wie er in 3DTK implementiert ist. Wir haben dabei gesehen, wie man den Baum mit unterschiedlichen Parametern konstruieren kann. Anschließend haben wir die Auswirkungen der Parameter auf die Laufzeiten der Konstruktion, NNS und ICP analysiert.

Auf die Konstruktionszeit wirkt sich eine größere Blattgröße verständlicherweise positiv aus. Die Partition nach der AABB-Mitte und dem Durchschnitt können keine gute Aufteilung und deshalb auch keine gute Laufzeit garantieren, der Median jedoch schon. Praktisch macht das nichts aus, denn die Datensätze unterscheiden sich stark vom Worst-Case. Für die Nächste-Nachbarn-Suche macht die Blattgröße asymptotisch keinen Unterschied mehr.

Es wurde erklärt, warum die präsentierten Zeiten das Minimum vieler Messungungen sind. Die Experimente haben gezeigt, dass sich die Wahl der Blattgröße durchaus noch auf die hier verwendeten Datensätze auswirkt. Bei allen Datensätzen war eine große Dauer sowohl der Konstruktion als auch des Matchings bei kleinen Blattgrößen erkennbar. Die Dauer der Konstruktion sinkt bei steigenden Blattgrößen. Die Dauer des Matchings nimmt jedoch bei größeren Blattgrößen wieder zu. Das Minimum der Matching-Dauer liegt interessanterweise bei allen Datensätzen zwischen 10 und 25, wobei eine genaue eingrenzung wegen der Messungenauigkeit schwierig ist. Die Bremer Datensätze zeigen sogar beide das gleiche Minimum bei 20.

Die Konstruktion hat durch die Berechnung des Medians deutlich länger gedauert, durch den Durchschnitt nur ein wenig länger. Die Aufteilung nach dem Median zahlt sich auf den meisten Datensätzen weniger als die Aufteilung nach dem Durchschnitt aus. Die Aufteilung nach dem Durchschnitt lohnt sich nur bei einem bestimmten künstlichen Datensatz nicht.

Es ist also sinnvoll, die Standard-Blattgröße in 3DTK leicht nach oben anzupassen und die Blattgröße generell parametrisierbar zu lassen. Die Aufteilung nach dem Durchschnitt sollte in 3DTK aufgenommen werden, indem sie gleichzeitig mit der Berechnung der AABB stattfindet.

5.1 Unbehandelte Probleme

Diese Arbeit hat sich in der Wahl der Datensätze auf eine sehr kleine Auswahl an echten Datensätzen beschränkt. Es wäre interessant, die hier angestellten Messungen auch noch einmal auf viel größeren Datensätzen mit unterschiedlichen Charakteristiken, wie z.B. der Qualität des Scanners, durchzuführen. Als Ergebnis wüsste man, ob es tatsächlich einen allgemein besten Wert für die Blattgröße gibt.

Außerdem kann man dann einen Gesamtvergleich aufstellen, wie sehr sich die Berechnung des Medians lohnt. Die Ergebnisse hier wurden wie erwähnt nur mit zwei Scans produziert.

Es wäre auch interessant, zu ermitteln, ob ein direkter Zusammenhang zwischen der Größe des Prozessor-Caches und der optimalen Blattgröße besteht. In 3DTK könnte dann automatisch das Optimum ausgewählt werden.

Literaturverzeichnis

[1] Abellán, A.; Vilaplana, J. M.; Martínez, J: Application of a long-range Terrestrial Laser Scanner to a detailed rockfall study at Vall de Núria (Eastern Pyrenees, Spain). In: *Engineering geology* 88 (2006), Nr. 3, S. 136–148

[2] Arthur, David; Vassilvitskii, Sergei: Worst-case and smoothed analysis of the ICP algorithm, with an application to the k-means method. In: *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)* IEEE, 2006

[3] Arun, K. S.; Huang, Thomas S.; Blostein, Steven D.: Least-squares fitting of two 3-D point sets. In: *IEEE Transactions on pattern analysis and machine intelligence* (1987), Nr. 5, S. 698–700

[4] Besl, Paul J.; McKay, Neil D.: Method for registration of 3-D shapes. In: *Robotics-DL tentative* International Society for Optics and Photonics, 1992

[5] Bosché, Frédéric: Automated recognition of 3D CAD model objects in laser scans and calculation of as-built dimensions for dimensional compliance control in construction. In: *Advanced engineering informatics* 24 (2010), Nr. 1, S. 107–118

[6] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford S.: *Introduction to Algorithms*. 3. MIT Press and McGraw-Hill, 2009. – ISBN 0–262–03384–4

[7] *Kapitel* 5.2 Kd-Trees. In: de Berg, Mark; Cheong, Otfried; van Kreveld, Marc; Overmars, Mark: *Computational Geometry*. 3. Springer-Verlag Berlin Heidelberg, 2008. – ISBN 978–3–540–77973–5, S. 99–105

[8] Elseberg, Jan; Magnenat, Stéphane; Siegwart, Roland; Nüchter, Andreas: Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. In: *Journal of Software Engineering for Robotics* 3 (2012), Nr. 1, S. 2–12

[9] Friedman, Jerome H.; Bentley, Jon L.; Finkel, Raphael A.: An algorithm for finding best matches in logarithmic expected time. In: *ACM Transactions on Mathematical Software (TOMS)* 3 (1977), Nr. 3, S. 209–226

[10] Musser, David R.: Introspective sorting and selection algorithms. In: *Software: Practice & Experience* 27 (1997), Nr. 8, S. 983–993

[11] Nüchter, Andreas u. a.: *3DTK – The 3D Toolkit*. http://slam6d.sourceforge.net/

[12] Petrie, Gordon; Toth, Charles K.: Terrestrial laser scanners. In: *Topographic Laser Ranging and Scanning Principles and Processing* (2009), S. 87–128

[13] Schauer, Johannes; Nüchter, Andreas: Collision detection between point clouds using an efficient k-d tree implementation. In: *Advanced Engineering Informatics* 29 (2015), Nr. 3, S. 440–458

[14] *Kapitel* 3.10 Options That Control Optimization. In: Stallman, Richard M.; GCC Developer Community: *Using The Gnu Compiler Collection*

[15] Surmann, Hartmut; Nüchter, Andreas; Hertzberg, Joachim: An autonomous mobile robot with a 3D laser range finder for 3D exploration and digitalization of indoor environments. In: *Robotics and Autonomous Systems* 45 (2003), Nr. 3, S. 181–198

[16] Tran, Hoang: AMD faces suit over alleged misrepresentation of new CPU. In: *Legal NewsLine* http://legalnewsline.com/stories/510646458-amd-faces-suit-over-alleged-misrepresentation-of-new-cpu

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Würzburg, Oktober 2016