Eine Milliarde 3D-Punkte mit Standardhardware verarbeiten

Processing One Billion 3D Points on a Standard Computer

Jan Elseberg, Dorit Borrmann, Andreas Nüchter

Dieser Beitrag stellt eine neue Implementation der Octree-Datenstruktur vor. Sie ermöglicht es, eine Milliarde 3D-Punkte in 8 GB Hauptspeicher exakt zu repräsentieren und effiziente Algorithmen zu implementieren. Der Octree gibt eine Hierarchie vor, die dazu verwendet werden kann, große Punktwolken zu inspizieren und flüssig in ihnen zu navigieren. Des Weiteren schlagen wir in diesem Artikel ein effizentes binäres Dateiformat für den Austausch von 3D-Scans vor.

Schlüsselbegriffe: 3D-Punktwolke, Octree, baumartige Datenstrukturen, verlustfreie Datenkomprimierung, Dateiformate

In this paper we describe a new implementation of the spatial data structure called octree. We present an encoding that is capable of storing one billion points in 8 GB memory. The octree imposes a hierachy that can be used to inspect and visualize large point clouds and to navigate smoothly in it. In addition, we propose an efficient file format for exchanging 3D scans.

Key words: 3D point cloud, octree, tree like data structures, lossless compression, file formats

1 EINLEITUNG

Moderne 3D-Laserscanner tasten Umgebungen mit vielen Millionen 3D-Punkten ab. So kann beispielsweise der Velodyne HDL-64E über 1.8 Mio., das Mobile Laser Scanning System RIE 6L VMX-450 mit 1,1 Mio., der Z+F IMAGER 5010 bis zu 1.016 Mio. und der FARO Focus^{3D} bis zu 976.000 3D-Punkte in der Sekunde erfassen. Diese riesigen Datenmengen müssen gespeichert und verarbeitet werden. In vielen Anwendungen wird oftmals jeweils nur ein kleiner Teil der Punktwolke verwendet oder es werden nur Approximationen der Rohdaten in Form von Voxeln – also dreidimensionalen Rastergrafiken – verwendet [29].

Wir beschreiben eine räumliche Datenstruktur, die Octree (latein: *octo*, deutsch: *acht*, english: *tree*, deutsch: *Baum*) genannt wird. Durch das Verwenden von aktuellen Ergebnissen aus der Computergrafik erhalten wir eine Implementierung mit vorteilhaften Eigenschaften: Erstens, unsere Datenstruktur speichert die 3D-Punkte direkt, ohne Approximationen durch Voxel zu verwenden. Voxel sind in vielen Anwendungen sehr nützlich, in der Vermessungstechnik jedoch benötigt man Zugriff auf die vermessenen 3D-Punkte, beispielsweise für das Auswählen von Punkten, für Scanmatching oder das Fitting von Primitiven in 3D-Scans. Zweitens, muss die Datenstruktur sehr schnell sein, d.h. Einfüge-, Lösch- und Zugriffsoperationen sollten in $O(\log n)$ geschehen, wobei n die Anzahl der gespeicherten 3D-Punkte ist. In diesem Artikel präsentieren wir eine Octree-Implementation, die beide Kriterien erfüllt und in der Lage ist, *eine Millarde 3D-Punkte in 8 GB Hauptspeicher* zu halten. Im Prinzip ist es sogar möglich, Teile von großen Szenen auf Festplatte auszulagern und bei Bedarf extrem schnell nachzuladen.

Die vorgestellte Datenstruktur ist als Open-Source verfügbar und Teil des *3DTK – The 3D Toolkit* [3]. Neben dem Octree präsentiert dieser Beitrag ein Dateiformat für den Austausch von 3D-Scans und wichtige Algorithmen wie das Frustum Culling und Ray Casting.

2 OCTREES FÜR DAS SPEICHERN VON 3D-PUNKTWOLKEN

Ein Octree ist eine Datenstruktur für die Indizierung dreidimensionaler Daten und erweitert somit das Konzept der Binärbäume und Quadtrees, die eindimensionale bzw. zweidimensionale Daten strukturieren. Jeder Knoten eines Octrees repräsentiert ein Volumen als Quader, wobei häufig die Vereinfachung gemacht wird, dass Octree-Knoten Würfel repräsentieren. Des Weiteren sind die Würfel

oftmals an den Achsen des Koordinatensystems ausgerichtet. Jeder Octree-Knoten besitzt bis zu acht Kinderknoten. Ist ein Knoten ohne Kinder, bedeutet dies, dass der zugehörige Würfel uniform repräsentiert werden kann und keine weitere Unterteilung notwendig ist.

Bei der Repräsentation von volumetrischen Daten ist die weitere Unterteilung eines Knoten auch dann unnötig, wenn das abgebildete Volumen vollständig uniform besetzt ist. Da Punkte keine Ausdehnung haben, lässt sich dieses Uniformitätskriterium nicht für das Speichern von Punkten verwenden. Daher erfordert diese Anwendung ein alternatives Abbruchkriterium, wie beispielsweise die Definition einer maximalen Tiefe. Äquivalent lässt sich eine minimale Würfelgröße definieren. Würfel mit minimaler Würfelgröße enthalten eine Liste von 3D-Punkten. *Abb. 1* stellt ein typisches Octree-Schema vor. Wir verwenden nicht die lehrbuchartige Octree-Repräsentation, bei der alle inneren Knoten mit gleicher Tiefe exakt 8 Kinderknoten haben, sondern weichen in zwei Merkmalen davon ab: Erstens werden leere Knoten, d.h. Würfel, die keine 3D-Punkte enthalten, sofort abgeschnitten. Fehlende Knoten repräsentieren also Freiraum. Zweitens werden Würfel, die nur einen 3D-Datenpunkt enthalten, nicht weiter unterteilt, sondern direkt zu Blättern gemacht. Da 3D-Laserscans nur Oberflächen und keine Volumina erfassen, ist zu erwarten, dass sehr viele Knoten im Octree nur sehr wenige Kinderknoten besitzen. Der Octree ist dünn besetzt.

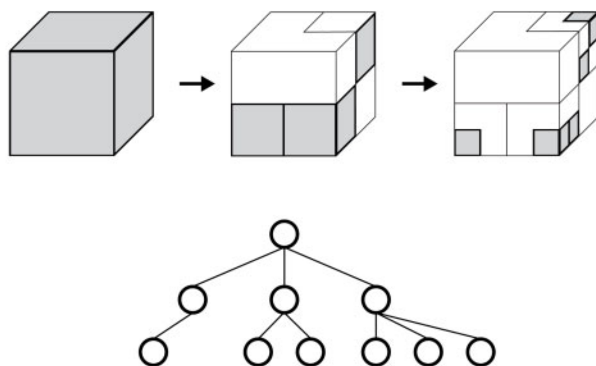

Abb. 1 | Ein Octree mit Tiefe 3. Grau unterlegte Würfel müssen weiter unterteilt und mit Nachfolgerknoten repräsentiert werden. Links: Volumen; rechts: Baumstruktur.

```
struct OcTree {
  float center[3];
  float size[3];
  OcTree *child[8];
  int nr_points;
  float **points;
};
```

Abb. 2 | Ineffiziente C/C++-Implementierung eines Octree-Knotens, der Position, Größe und 8-Zeiger auf Nachfolgerknoten enthält. Auf einer 64-Bit-Architektur, benötigt diese Darstellung mindestens 100 Bytes. Quelle: [3], rev. r155. Ähnliche Implementierungen finden sich mit (72+x) Bytes in *OctoMap – 3D occupancy mapping* [29,15] und mit (64+x) Bytes der *Point Cloud Library (PCL ver. 1.1.1)* [24, 21]. Zusätzlich bietet die PCL eine effizientere C++-Implementierung an (low memonry base), die (25+x) Bytes pro Knoten benötigt. Durch die Verwendung von C++-Templates kann x dabei variieren. Des Weiteren benötigen alle Referenzimplementierungen 8 weitere Bytes, um virtuelle Funkionen mit einem Eintrag in der vtable abbilden zu können.

Bei vielen Implementationen von Octrees steht nicht die Optimierung des Speicherverbrauchs im Vordergrund. *Abb. 2* zeigt ein typisches Code-Beispiel, das redundant die Position und Größe speichert. Im Folgenden zeigen wir Vorteile unserer Implementierung auf.

2.1 Definition eines effizienten Octrees

Unsere Implementation zielt darauf ab, einen Octree effizient darzustellen und dennoch sehr schnell auf die gespeicherten Daten zugreifen zu können. Daher scheidet eine serialisierte Darstellung, die frei von Zeigern ist, aus, weil diese nur den Zugriff auf die Datenpunkte in linearer Zeit erlaubt. Der Zugriff auf die Daten soll in logarithmischer Zeit geschehen.

Der größte Teil der Information in einem inneren Knoten lässt sich während des Durchlaufens des Baums berechnen. Offensichtlich ist die Tiefe eines Knotens gleich der Tiefe des Vaterknotens plus eins. Durch die Eigenschaft, dass ein Octree den Würfel regelmäßig unterteilt, ergibt sich, dass die Größe eines Würfels eine Funktion seiner Tiefe ist. Auch die Position eines Knotens lässt sich auf Grundlage der Position des Vaterknotens berechnen. Daher muss nur die Wurzel des Baumes Informationen wie Position und Größe des umspannenden Volumens enthalten. Zeiger zu Vaterknoten lassen sich berechnen, indem man sie auf einem Stapel zwischenspeichert. Es ist sogar möglich, Zeiger benachbarter Knoten durch ein Indexschema zu berechnen. Leider benötigt diese Operation Backtracking über die Vaterknoten und ist daher weniger effizient.

Um speichereffizient Knoten zu repräsentieren, lassen wir sämtliche Informationen aus Knoten weg, die online berechnet werden können. Dies reduziert jedoch die Größe eines Knotens im obigen Beispiel nur um 24 von 100 Bytes. Allein 64 Bytes werden für die Speicherung der Nachfolgerknoten benötigt. Dies kann vermieden werden, indem man die Information, ob ein Knoten vorhanden ist oder nicht, in seinen Vaterknoten verschiebt. Durch ein Byte, bei dem jedes Bit einen Nachfolgerknoten repräsentiert, lässt sich dies erreichen. Des Weiteren sparen wir 58 Bytes, indem wir nur einen Zeiger zu allen Nachfolgerknoten speichern und dieser relative Zeiger 6 Bytes groß ist. Zusätzlich verwenden wir ein Byte, wobei jedes Bit anzeigt, ob der Nachfolgerknoten ein innerer Knoten oder ein Blatt ist. Dadurch wird die Information über die Punkte überflüssig. *Abb. 3* (links) stellt diese Knoten-Struktur vor. Alternativ kann eine konstante Tiefe definiert werden, in der sich alle besetzten Blätter des Octrees befinden. Der rechte Teil in *Abb. 3* stellt die dann zu verwendende Knotenrepräsentation vor.

Für das Speichern des Zeigers auf die Nachfolgerknoten verwenden wir 6 Bytes. Dies reicht aus, um 256 Terrabyte zu adressieren. Es ist also nicht notwendig, ein weiteres Bit zum Signalisieren eines

Abb. 3 | Die zwei vorgeschlagenen Kodierungen für Knoten in einem Octree. Der Zeiger auf Nachfolgerknoten nimmt den größten Speicherplatz ein. Es ist ein relativer Zeiger, für den wir auf einem 64-Bit-System 48 Bits reservieren. valid und leaf sind jeweils 8 bit groß. Links: Kodierung mit separaten Bitfeldern für valid und leaf. Rechts: Alternative Knotenrepräsentation für Octrees mit konstanter Tiefe.

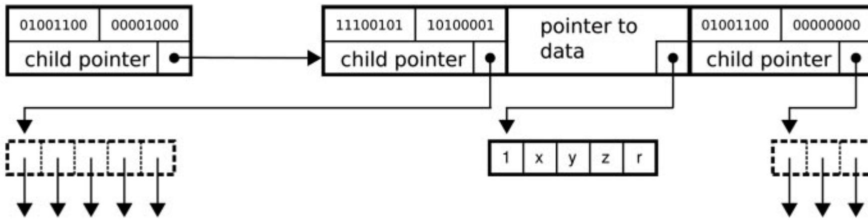

Abb. 4 | Beispiel eines Octrees mit der vorgeschlagenen Kodierung. Der Knoten links besitzt 3 Kinderknoten, von denen einer ein Blatt ist. Der Zeiger auf die Nachfolger repräsentiert daher drei hintereinander gespeicherte Knoten. Der Blattknoten ist ein einfacher Zeiger auf ein Feld mit der Anzahl der Punkte und den Werten.

far-Zeigers wie in [19] zu verwenden. Auch das Sparen von Bits in der leaf Repräsentation erscheint nicht sinnvoll. Zwar könnte man die 3^8=6561 möglichen Kombinationen bereits mit 13 statt 16 Bits repräsentieren, jedoch müsste diese geringfügige Reduktion durch weitere Berechnungen wettgemacht werden.

In den Blättern werden ausschließlich 3D-Punkte gespeichert und somit benötigen sie eine andere Repräsentation. In *Abb. 4* ist ein Knoten mit drei Nachfolgerknoten dargestellt, wobei einer dieser Knoten ein Blatt mit einem Punkt ist. Blätter enthalten Zeiger zu Listen mit Punkten. Der erste Eintrag ist die Anzahl der Punkte, gefolgt von einer Liste der Koordinaten und den Attributen. In dieser Darstellung würden Blattknoten *n*-Bytes größer sein, wobei *n* die Anzahl der Bytes ist, die für die Zahl der Punkte benötigt wird. In unserem Fall ist es ausreichend, 4 Bytes dafür zu reservieren. Eine derartige Listenrepräsentation ist bereits speichereffizienter als die übliche Darstellung als float**, da ein Zeiger pro Punkt eingespart wird.

2.2 Komprimierung von 3D-Punktwolken mit Octrees

Der Speicherbedarf unseres Octrees ist nun so gering, dass die 3D-Punkte mit ihren Attributen den größten Platz einnehmen. Daher muss die Punktliste selbst komprimiert werden. Dabei verwenden wir 2 Bytes für jede Koordinate. 2 Bytes sind bestens geeignet, da dies der Auflösung der zusätzlichen Punktattribute wie Reflektivität der meisten terrestrischen Laserscanner entspricht. Die Auflösung der 3D-Daten darf sich jedoch durch das Verwenden von 2-Byte-Koordinaten nur unmerklich ändern. Dies geschieht folgendermaßen: Jede der 2-Byte-Koordinaten wird als $s/(2^{16})$ Inkrement von der unteren, vorderen, linken Ecke eines Blattknotens angenommen, wobei s die Seitenlänge des Octree-Würfels ist. Dadurch speichert jede Stufe des Octrees ein signifikantes Bit jeder Koordinate. Ein 4-Byte-float hat eine Auflösung von zirka [100]μm bei einer maximalen Distanz von [500]m und [1] μm bei [1.5]m Distanz. Um ebenfalls eine Genauigkeit von [1] μm zu erzielen, muss der Octree-Würfel kleiner als [6.5]cm sein. In unserer Implementation haben wir die Seitenlänge des Würfels auf [65]cm gesetzt und erzielen so Genauigkeiten von [10] μm, das ca. zwei Größenordnungen besser ist als typische Messgenauigkeiten.

2.3 Experimente und Ergebnisse

Um die Leistungsfähigkeit des vorgeschlagenen Octrees zu evaluieren, haben wir den Speicherbedarf für verschiedene Baumtiefen berechnet. Dies wurde für drei repräsentative Datensätze, die in *Abb. 5* dargestellt sind, durchgeführt. Die Auswertungen sind in den *Tabellen 1* bis *3* für verschiedene Blattgrößen wiedergegeben. Dabei ist zu beachten, dass die angegebene Blattgröße nur die Hälfte der Seitenlänge eines Octreewürfels ist. Des Weiteren wurden für alle Tests die Wurzel des Octree und damit alle seine Würfel entlang der Achsen des Koordinatensystems ausgerichtet. Die Größe der Wurzel ist so gewählt, dass sie minimal ist, aber alle 3D-Punkte enthält.

Tabelle 1 | Benötigter Speicherplatz für Octrees bei einem dünn besetzten 3D-Scan und verschiedenen Auflösungen. Die erste Spalte gibt die Größe (halbe Seitenlänge) des kleinsten Würfels im Baum wieder. Die zweite und dritte Spalte enthalten die Anzahl der inneren Knoten und Blätter, gefolgt von der Anzahl der inneren Knoten in der erwähnten 7-Byte-Implementation. Die Anzahl der Blätter der 7-Byte Implementation ist im Vergleich zu der 8-Byte-Implementation unverändert. Die letzten drei Spalten geben den Speicherbedarf für die 8 und 7 Byte Implementierung sowie die Referenz mit den 100 Bytes (vgl. *Abb. 2*) an.

Blatt größe cm	# Knoten	# Blätter	# Knoten 7 Bytes	Speicherbedarf	Speicherbedarf 7 Bytes	Speicherbedarfreferenz
876	1	8	1	104 B	103 B	954 B
438	9	26	9	384 B	375 B	3.71 B
219	34	77	35	1.19 kB	1.16 kB	11.87 kB
109.5	103	202	112	3.24 kB	3.2 kB	33.28 kB
54.76	282	505	314	8.31 kB	8.25 kB	86.81 kB
27.38	698	1352	819	21.8 kB	21.95 kB	230.12 kB
13.69	1777	3688	2171	58.47 kB	59.45 kB	621.05 kB
6.846	4121	8840	5859	139.04 kB	147.09 kB	1.55 MB
3.423	9327	19064	14699	303.38 kB	331.66 kB	3.57 MB
1.711	17219	34697	33763	554.11 kB	652.7 kB	7.25 MB
0.855	27668	52836	68460	855.37 kB	1.11 MB	12.85 MB
0.427	37351	68253	121296	1.11 MB	1.66 MB	20.09 MB

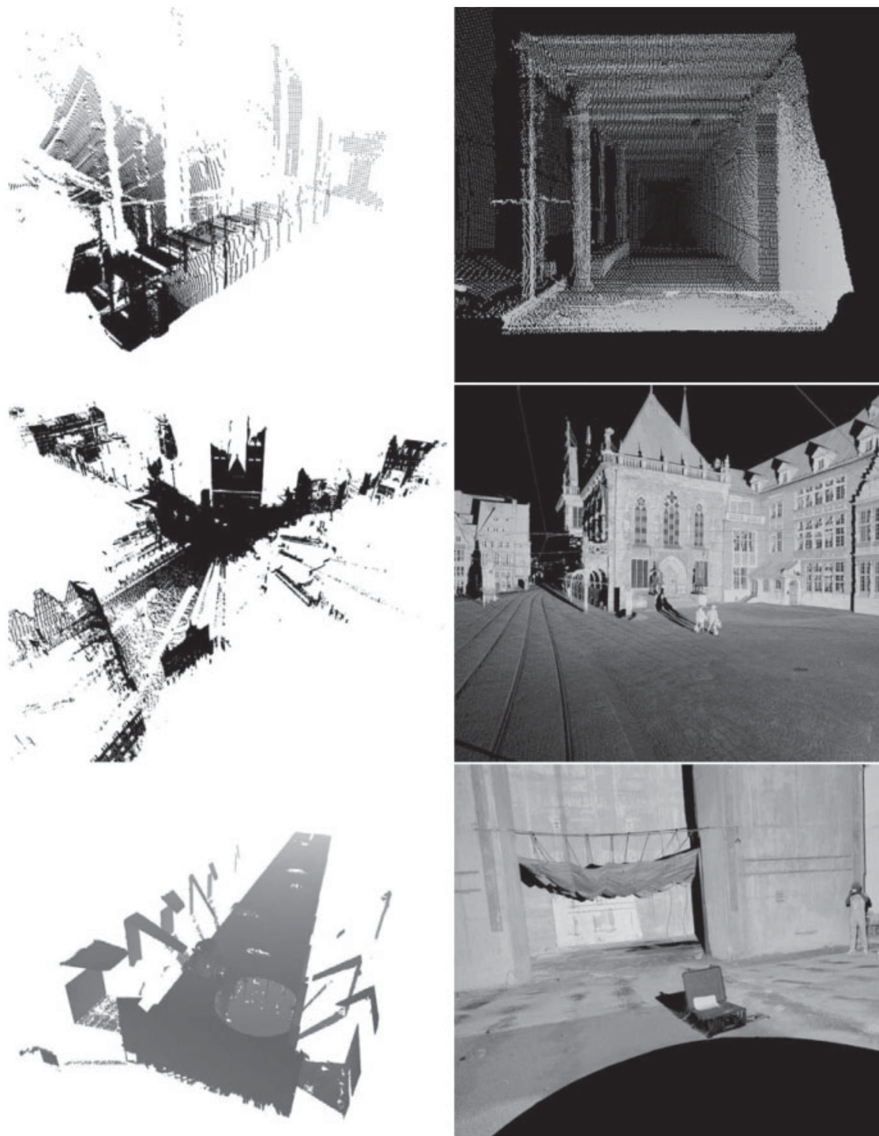

Abb. 5 | Die drei Punktwolken für die Analyse des Octrees. Die erste Zeile zeigt eine Punktwolke, die mit einem rotierenden SICK LMS200 Laserscanner in einer Büroumgebung aufgenommen wurde. Sie enthält 81360 3D-Punkte (≈1.5 MB). Die Statistik über diesen Datensatz befindet sich in *Tabelle 1*. Eine Übersicht über den 3D-Scan sowie eine Detailaufnahme sind angegeben. Der mittlere Scan wurde auf dem Marktplatz in Bremen mit einem Riegl VZ-400 aufgenommen. Die Punktwolke enthält 15896875 3D-Punkte (≈303 MB). *Tabelle 2* enthält Details zu diesem Datensatz. Der untere Scan wurde ebenfalls mit einem Riegl VZ-400 aufgenommen, allerdings in einem sehr großen Innenraum, dem U-Boot Bunker Valentin in Bremen Varge. Daher ist die Punktwolke dicht und enthält 22538374 3D-Punkte (≈420 MB). Ergebnisse dazu befinden sich in *Tabelle 3*.

Tabelle 2 | Statistik für den Bremer Marktplatz Datensatz (vgl. *Tabelle 1*).

Blatt größe cm	# Knoten	# Blätter	# Knoten 7 Bytes	Speicherbedarf	Speicherbedarf 7 Bytes	Speicherbedarfreferenz
8560	6	12	6	192 B	186 B	1.9 kB
4280	18	28	18	480 B	462 B	4.8 kB
2140	46	86	46	1.4 kB	1.3 kB	13.9 kB
1070	129	296	132	4.5 kB	4.4 kB	45.3 kB
535	408	800	428	12.8 kB	12.5 kB	130.1 kB
267	1166	2595	1228	40.4 kB	39.7 kB	405.2 kB
133	3616	7993	3823	124.8 kB	122.6 kB	1.25 MB
66.8	11130	24587	11816	384.1 kB	377.7 kB	3.85 MB
33.4	33965	75999	36403	1.18 MB	1.16 MB	11.91 MB
16.7	102728	233413	112402	3.62 MB	3.58 MB	36.65 MB
8.35	302573	687529	345815	10.67 MB	10.67 MB	109.53 MB
4.17	814040	1808993	1033344	28.22 MB	28.94 MB	301.28 MB
2.08	1927234	4166979	2842337	65.42 MB	69.9 MB	742.98 MB
1.04	4031140	7783889	7009316	125.65 MB	142.47 MB	1.568 GB
0.52	5592151	10142923	14793205	166.45 MB	225.26 MB	2.643 GB

Tabelle 3 | Statistik für den Bunker Valentin Datensatz (vgl. *Tabelle 1*)

Blatt größe cm	# Knoten	# Blätter	# Knoten 7 Bytes	Speicher- bedarf	Speicher- bedarf 7 Bytes	Speicher- bedarf referenz
5589	1	8	1	104 B	103 B	954 B
2794	9	22	9	336 B	327 B	3.28 kB
1397	30	59	31	948 B	925 B	9.54 kB
698	88	209	90	3.21 kB	3.13 kB	31.69 kB
349	294	755	299	11.41 kB	11.15 kB	111.72 kB
174	1038	2611	1054	39.63 kB	38.71 kB	388.49 kB
87.3	3602	8761	3665	133.64 kB	130.78 kB	1.31 MB
43.6	12106	30330	12426	460.8 kB	450.94 kB	4.53 MB
21.8	41098	101743	42756	1.54 MB	1.52 MB	15.31 MB
10.9	134366	324058	144499	4.96 MB	4.9 MB	49.667 MB
5.45	412174	969512	468557	14.93 MB	14.91 MB	152.43 MB
2.72	1158782	2679693	1438069	41.42 MB	42.22 MB	436.48 MB
1.36	2977454	6617215	4117762	103.22 MB	108.23 MB	1.137 GB
0.68	6356651	13130561	10734977	208.41 MB	232.71 MB	2.529 GB

Tabelle 4 | Komprimierungsergebnisse für die Datensätze. Die Kompressionsrate zwischen den binären Originaldaten und der Octreedarstellung sowie der durchschnittliche absolute Fehler sind angegeben.

Datensatz	Dateigröße .txt-Datei in MB	Dateigröße binär 64 (32 bit) in MB	Dateigröße Octree in MB	Rate %	Fehler μm
Kurt3D	1.907	1.862 (0.931)	0.472	50.73	4.165
Bremer Marktplatz	477.0	424.4 (242.5)	121.6	50.14	5.102
Bunker Valentin	665.7	601.8 (343.9)	172.1	50.04	6.677

Der erste Datensatz enthält nur eine kleine Anzahl von 3D-Punkten. Dies hat zur Folge, dass viele Punkte große Abstände zu ihren Nachbarn besitzen. Dies schlägt sich besonders in dem direkten Vergleich zwischen den hier vorgestellten Octree-Varianten nieder. Bereits bei einer Tiefe von 5 ist der Octree mit dynamischer Tiefe und größeren Knoten effizienter als der ein Byte kleinere Octree konstanter Tiefe. In größeren Datensätzen (vgl. *Tabelle 2* und *Tabelle 3*) wie sie typischerweise beim terrestrischen Laserscanning auftreten, erreicht der Octree die Tiefe 10, bis die Forderung nach konstanter Tiefe in mehr Speicherplatz resultiert. Sogar im dritten Datensatz mit den dichtesten Daten spricht etliches gegen die Eigenschaft, konstante Tiefe zu verwenden. Wir sparen im Bunker Valentin Datensatz im Idealfall zirka 40 kB. Verglichen mit der Größe der Punktwolke ist dies vernachlässigbar. Da die Eigenschaft, konstante Tiefe zu verwenden, in weniger idealen Fällen explosionsartig in mehr Speicherplatz resultiert, macht es wenig Sinn, die „kleinere" Repräsentation des Octreeknotens zu verwenden.

3 EIN DATEIFORMAT ZUM AUSTAUSCH VON 3D-PUNKTWOLKEN

Durch die vielen existierenden 3D-Laserscanner Hersteller und diversen Anwendungen existieren etliche Softwareprodukte, die jeweils für eine spezielle Anwendung optimiert sind. Im Bereich des luft- und fahrzeuggestützten Laserscannings existiert ein allgemeines Dateiformat, das .las Format [25]. „This binary file format is an alternative to proprietary systems or a generic ASCII file interchange system used by many companies. A problem with proprietary systems is that data cannot be easily taken from one system or process flow to another. In addition, processing performance is degraded because the reading and interpretation of ASCII elevation data can be very slow and the file size can be extremely large, even for small amounts of data."[2]. Dies trifft auch auf Abläufe im terrestrischen Laserscanning zu, jedoch sind dort (noch) keine Standardfileformate etabliert. [16] definiert ein binäres Format, um Verarbeitungszeit durch den Austausch von Textdateien zu reduzieren.

Der Octree, wie er in Abschnitt [2] präsentiert wurde, ist bestens geeignet, große 3D-Punktwolken zu speichern, da diese verlustfrei um einen Faktor von ungefähr 2 komprimiert werden. Somit verkürzen sich auch die Zeiten zum Laden und Speichern der Daten. *Tabelle 4* präsentiert die Komprimierungsergebnisse für die drei Datensätze. Die Serialisierung unserer Datenstruktur ergibt also eine Möglichkeit zum Speichern von großen Punktwolken.

4 EFFIZIENTE ALGORITHMEN AUF OCTREES

4.1 Adaptive Visualisierung mit Frustum Culling

Frustum Culling ist eine Optimierungsmethode, bei der alle 3D-Punkte vom Zeichnen ausgeschlossen werden, die außerhalb des Sichtbereichs liegen. Es sorgt dafür, dass akzeptable Bildwiederholraten auch bei großen Szenen möglich sind, denn durch Frustum Culling werden viele der Punkte der Szene nicht gezeichnet, die man ohnehin nicht sehen kann. Im Folgenden werden wir aufzeigen, wie

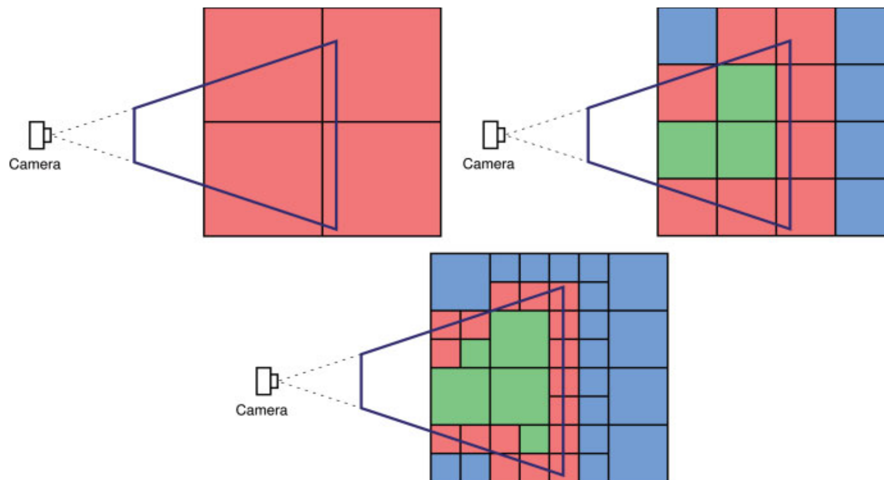

Abb. 6 | Das Prinzip des Frustum Cullings mit einem Octree. Knoten in rot schneiden das Frustum partiell, so dass hier weitere Berechnungen erforderlich sind. Von grünen Knoten ist bekannt, dass sie innerhalb des Frustums liegen, so dass sie gänzlich dargestellt werden. Knoten ausserhalb des Frustum sind in blau eingezeichnet und können folglich ignoriert werden.

Algorithm 1 | display(set<plane> frustum, octree node)

```
set<plane> childfrustum;
for all p ∈ frustum do
  if PlaneAABBCollision(p, this) = within then
    childfrustum.insert(p)
  else if PlaneAABBCollision(p, this) = outside then
    return;
  end if
end for
if isLeaf(node) then
  drawPoints(node)
else
  for all child ∈ node.children do
    display(childfrustum, child)
  end for
end if
```

unser Octree als Hierarchie von Geometriebegrenzungen von Volumina dazu verwendet werden kann, effizient 3D-Punkte anzuzeigen. Ähnlich wie bei qsplat [23] kann Frustum Culling mit einem dynamischen Detailierungsgrad kombiniert werden, so dass Anzeigeprogramme hohe Bildwiederholungsraten erzielen.

Unser Algorithmus (vgl. *Algorithmus 1*) bestimmt zunächst, welche Ebenen des Frustums (engl.: *Frustum*, deutsch: *Kegelstumpf*) relevant für die Nachfolgerknoten im Octree sind. Gleichzeitig wird überprüft, ob der aktuelle Würfel sichtbar ist und entsprechend terminiert. Im zweiten Schritt werden entweder die Punkte im Volumen angezeigt oder es wird Rekursion ausgeführt. Unnötiges Überprüfen lässt sich somit vermeiden. *Abb. 6* verdeutlicht dieses Prinzip.

Das dynamische Einstellen des Detailgrades der angezeigten Szene kann mit zwei Strategien geschehen. Wie in Abb. 6 dargestellt, lassen sich verschiedene Octree-Tiefen visualisieren. Dabei muss ein Kompromiss zwischen Auflösung und Anzeigegeschwindigkeit gefunden werden. Eine Alternative ist, alle sichtbaren Knoten anzuzeigen, aber mit einer reduzierten Anzahl von 3D-Punkten. Dies erscheint besonders sinnvoll, da der Engpass während der Anzeige die Übertragung der Punkte zur Grafikkarte ist (vgl. *Abb. 12*).

Um die Performanz unseres Frustum Cullings zu testen, haben wir ein Video mit einem Durchflug durch eine 3D-Punktwolke gerendert. Zu Beginn sind alle 3D-Punkte sichtbar und es entstehen, wie in *Abb. 8* ersichtlich, nur bei einem sehr feinen Octree zusätzliche Kosten. Während der Inspektion von 3D-Punktwolken wird in der Regel nur ein kleiner Teil der 3D-Punkte angezeigt. Hier bietet das Frustum Culling den höchsten Geschwindigkeitsvorteil (vgl. rechter Teil der Kurve in *Abb. 8*).

4.2 Raycasting

Raycasting ist eine Methode für das Finden von Schnittpunkten eines Strahls mit einer Oberfläche. Dies ist ein Problem in der Computergrafik, wo es dazu verwendet wird, Bilder zu rendern [22] und in der Robotik, um Stichproben aus Verteilungen zu ziehen [28, 29]. Für das Rendern von Bildern wird jedes Pixel vom Standpunkt zurückverfolgt und der erste getroffene Objektpunkt bestimmt. Dies ist eine Variation von Raytracing, wo zusätzlich die Strahlen an Objekten reflektiert werden. Daher ist Raycasting vereinfachtes Raytracing.

Zunächst zeigen wir auf einem Octree, der keinerlei Zeiger auf Nachbarn enthält, schnellen indizierten Zugriff auf Knoten zu implementieren und Nachbarn zu bestimmen. Indizierter Zugriff auf Knoten ist ein Nachschlagen mit den (x,y,z) Integerkoordinaten auf der tiefsten Ebene des Octrees. Dies bedeutet, ein Octree mit Tiefe d besitzt die Integerkoordinaten 0 bis 2^d-1 in jeder der drei Dimensionen. Statt eine naive Implementierung ähnlich dem Nachschlagen bei Kollisionschecks mit Octreeebenen zu geben, benutzen wir Integerkoordinaten und Bitoperationen für ein effizientes Duchlaufen des Octrees. Ähnlich zu [9] nehmen wir dazu an, dass wir ein vorab berechnetes Feld childBitDepth mit childBitDepth[n] $= 1 << (\text{maxDepth} - d - 1)$ haben. Dann kann das Nachschlagen wie in *Algorithmus 2* geschehen. Eine wichtige Komponente ist Zeile 2.

Algorithm 2 | lookup(Vector3i index, octree node, parentTrace, depth)

```
loop
  int childBit = childBitDepth[depth]
  int childIndex = (index.x & childBit ≠ 0) << 2 || (index.y & childBit ≠
  0) << 1 || (index.z
  & childBit ≠ 0)
  octree *parentTrace[depth] = &node
  depth++
  node = node.children[childIndex]
  if isLeaf(node) then
    return
  end if
end loop
```

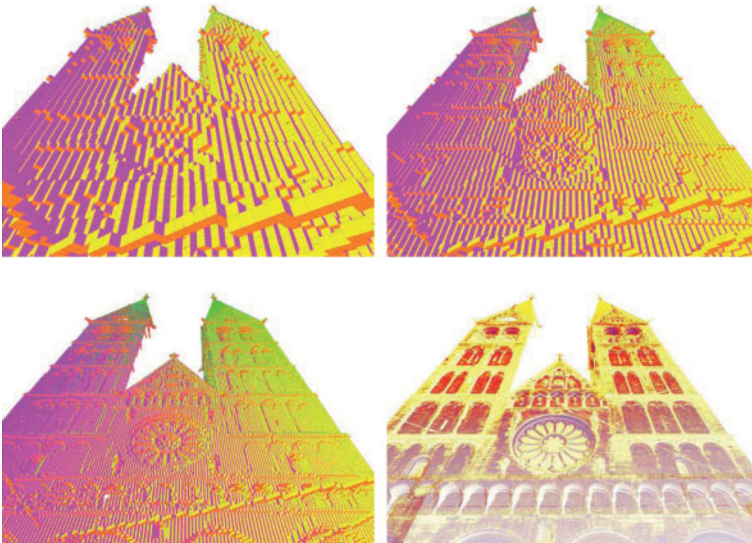

Abb. 7 | Drei verschiedene Detailgrade, die durch den Octree vorgegeben sind. Unten rechts ist die zugehörige 3D-Punktwolke dargestellt.

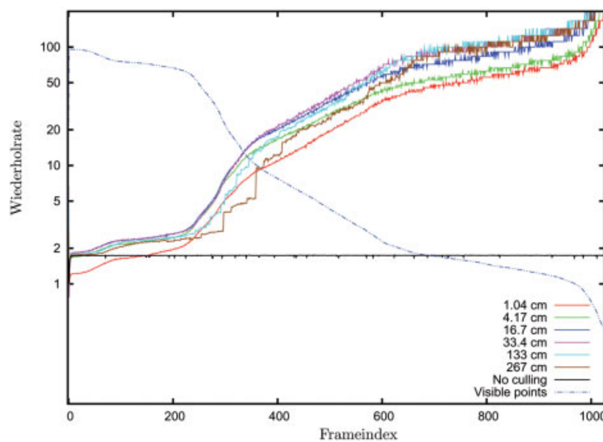

Abb. 8 | Bildwiederholrate des Frustum Cullings für verschiedene Würfelgrößen als Funktion über den Index während der Erstellung eines Videos. Des Weiteren sind die prozentualen Anteile der sichbaren Punkte in Abhängigkeit von allen Punkten der Szene eingetragen. Die *y*-Achse ist logarithmisch aufgetragen.

Hier werden die Integerkoordinaten einem Index des Nachfolgerknoten zugeordnet, der die gegebenen Koordinaten enthält. Der Algorithmus zeigt auch, wie Zeiger auf Vorgängerknoten simuliert werden können, indem diese während des Durchlaufens des Baumes verfolgt werden. Um einen Nachbarknoten zu finden, ist nur ein erweitertes Nachschlagen durchzuführen. Zunächst muss die Liste der Vorgängerknoten zurückverfolgt werden, bis der gewünschte Nachbarindex in einem Vorgängerknoten liegt. Anschließend kann der Knoten einfach nachgeschlagen werden, und der Nachbar ist gefunden.

Es ist uns somit gelungen, Zeiger auf Nachbarknoten zu eliminieren. Dies geschieht auf Kosten der Rechenzeit, da im schlechtesten Fall bis zur Wurzel zurückgegangen wird, um danach den gewünschten Knoten zu finden. Diese Situation tritt zwar äußerst selten auf, jedoch ist das Bestimmen des Nachbarn noch immer in $O(\log n)$. Mit Hilfe dieser Funktionalität können wir nun einen Raycasting Algorithmus angeben (vgl. *Algorithmus 3*). Dies ist eine Implementation der drei-dimensionalen Version des bekannten Bresenham Linienalgorithmus, der jeweils 3D-Integerkoordinaten ausgibt, die den Strahl des Raycastings beschreiben.

Algorithm 3 | castRay(Ray ray, octree root)

```
octree *parentTrace[maxDepth]
int depth = 0
octree node = lookup(ray.origin, root, parentTrace, depth)
Vector3i ci = Bresenham(ray);
loop
  while contains(node, ci) do
    ci = Bresenham(ray)
  end while
  node = findNeighbor(ci, node, parentTrace, depth)
  if isLeaf(node) then
    drawPoints(node)
    return
  end if
end loop
```

Dieses naive Raycasting kann verbessert werden, indem Strahlen zu Paketen zusammengefasst werden (vgl. Knoll et al. [18]). Weiterhin lassen sich durch eine Technik, die coherent octree traversal genannt wird, die Zugriffsoperationen auf die Datenstruktur weiter reduzieren.

4.3 Nächste Nachbarn Suche für das Scanmatching

Schnelle Suche nach nächsten Punkten (NNS, engl.: *Nearest Neighbor Search*) ist eine Grundvoraussetzung für effizientes Scanmatching mit dem ICP-Algorithmus (engl.: *Iterative Closest Point*, bzw. seiner global konsistenten Variante [6, 7] für das automatische Registrieren von 3D-Scans. Naiv implementiert ist dieser Schritt für jeden Anfragepunkt in $O(n)$, wobei n die Anzahl der Punkte im 3D-Scan ist. Diese schlechte Laufzeit lässt sich durch den Einsatz einer metrischen Datenstruktur verbessern. Der *k*D-Baum [11] mit $k = 3$ wird dazu am häufigsten verwendet, da er eine effiziente Implementierung der NNS erlaubt. Bei der Erstellung des *k*D-Baumes wird der Raum in jedem Schritt senkrecht zu einer der Koordinatenachsen unterteilt. In den Blättern werden die Punkte gespeichert. Im Gegensatz dazu unterteilt der Octree den Raum gleichmäßig und ist daher noch besser als der *k*D-Baum zur nächsten Nachbar Suche geeignet [4]. Die Schwierigkeit besteht jedoch darin, die NNS auf einem Octree zu implementieren. Der Baum muss effizient durchlaufen werden, um das Blatt zu finden, das den nächsten Punkt enthält. Dabei spielt die Reihenfolge, in der die Kindknoten durchlaufen werden, eine große Rolle. Die Anzahl der Knoten, die betrachtet werden müssen, wird am besten durch die Anwendung der „Bester-Knoten-zuerst"-Strategie (engl.: *best bin first*) reduziert. Das bedeutet, die Reihenfolge wird abhängig von der Distanz zum Anfragepunkt gewählt. Dies lässt sich bei einem *k*D-Baum einfach realisieren, benötigt jedoch etwas Aufwand in einem Octree, der bis zu 8 Nachfolgerknoten haben kann.

Für jeden Octreeknoten mit 8 Nachfolgern gibt es 48 mögliche Sequenzen, in denen Nachfolgerknoten durchlaufen werden können. Jeder Nachfolgerknoten repräsentiert einen 18er Würfel des Koordinatenraums und der Anfragepunkt kann in jeden dieser 8 Würfel fallen. Zur Suche nach dem nächsten Nachbarpunkt müssen aber auch die anderen Würfel betrachtet werden. Abhängig von der Lage des Anfragepunktes zu den drei Splitebenen des Würfels lassen sich 6 Durchlaufmöglichkeiten bestimmen. Daher muss die NNS auf einem Octree Abstände des Anfragepunktes zu allen drei Ebenen berechnen, diese sortieren und die nächste Ebene auswählen. Das Auswahlkriterium ist, das nächste Blatt auszuwählen. In einem *k*D-Baum ist hingegen nur eine einzige Abfrage notwendig, und es werden daher keine unnötigen Operationen ausgeführt für Knoten, die nicht besucht werden.

Durch die gleichmäßigen Unterteilungen in einem Octree ist die NNS schneller erfolgreich als für einen *k*D-Baum. Dies könnte den beschriebenen Mehraufwand ausgleichen. Den größten Vorteil erhalten wir, wenn wir die Indizierung, wie in *Abschnitt 4.2* beschrieben, anwenden. Dies erlaubt uns, direkt zu dem tiefsten Octree-Knoten abzusteigen, der den Anfragepunkt und den darum liegenden Ball aller in Betracht kommenden Punkte enthält. Dies geschieht mit einer konstanten Anzahl von Gleitkommaoperationen. Anschließend führen wir die vollständige NNS mit der „Bester-Knoten-zuerst"-Strategie aus. Der erste Schritt dieses Algorithmus ist wesentlich schneller als die korrespondierende Operation auf einem *k*D-Baum. Die Geschwindigkeit hängt stark von der maximal erlaubten Distanz vom Anfragepunkt ab. Je kleiner diese Distanz ist, desto tiefer kann man initial in den Baum absteigen und desto weniger Operationen sind für die vollständige NNS notwendig.

Um die Performanz dieser NNS für verschiedene, typische Maximaldistanzen zu bestimmen, setzen wir ICP basiertes Scanmatching mit unserem Octree und einem *k*D-Baum als Referenz ein. Die Ergebnisse befinden sich in *Tabelle 5* und *Abb. 9*. Ungeachtet unserer bisherigen Argumentation sieht man, dass die Octree-basierte NNS auch bei großen maximal zulässigen Abständen *nicht* wesentlich schlechter ist als der *k*D-Baum, jedoch nehmen wir eine erhöhte Varianz wahr. Die Varianz der NNS in einem *k*D-Baum ist sehr stabil für alle Distanzen. Dies hat zur Folge, dass ICP mit Octree-basierter NNS in Kombination mit großen Maximaldistanzen und schlechten initialen Poseschätzungen ineffizienter funktioniert.

Um die Anzahl der Gleitkommaoperationen zu reduzieren und zwecks vereinfachter Implementation, verringern wir die Anzahl der möglichen Kombinationen, in denen Blätter durchlaufen werden, von 48 auf 8. Dies geschieht, indem a priori für jeden der 8 Teilräume, in die ein Punkt fallen mag, genau eine zu durchlaufende Sequenz definiert wird. Da die Reihenfolge nur von der Position des Anfragepunktes abhängt, führen wir keine Abstandsberechnungen und Sortierungen mehr durch. Daher sind keine Gleitkommaoperationen

Tabelle 5 | Durchschnittliche Rechenzeiten für den ICP-Algorithmus mit NNS (Octree und *k*D-Baum). Der Durchschnitt wurde über 100 Registrierungen für den Datensatz Kurt3D und 20 für den Riegl Datensatz berechnet. Für jeden Test wurden Standardparameter verwendet, d.h. ein maximaler Punkt-Punktabstand von [25]cm und 50 ICP-Iterationen. Die initalen Posen wurden zufällig variiert. Dabei haben wir sie linear zwischen -25 und [25]cm verschoben und zwischen -10° and 10° rotiert. Die Zeit für die Konstruktion des Baumes ist nicht berücksichtigt.

Datensatz	*k*D-Baum	Octree
Kurt3D	3043.099	2386.881
Bremer Marktplatz	355848.476	314506.905
Bunker Valentin	837675.381	784241.905
Kurt3D reduziert	757.514	625.683
Bremer Marktplatz reduziert	91735.667	74706.85
Bunker Valentin reduziert	91153.0	74821.238

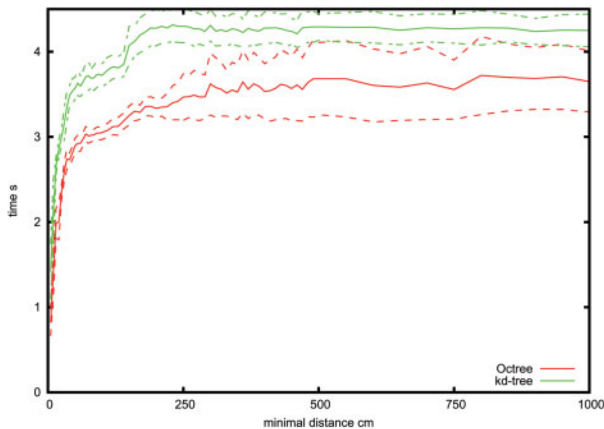

Abb. 9 | Durchschnitt und Standardabweichung für die Rechenzeiten des ICP-Algorithmus für den Octree und für den *k*D-Baum in Abhängigkeit der maximalen Punktdistanz. Für jeden Datenpunkt wurden 100 ICP-Registrierungen mit 50 Iterationen ausgewertet. Die Initalposen wurden analog *Tabelle 5* verrauscht.

in unserer NNS mehr notwendig mit Ausnahme der Operationen, die in einem Blatt ausgeführt werden. Hierbei muss die Liste der gespeicherten Punkte mit dem Anfragepunkt verglichen werden. Eine Zusammenfassung unseres Algorithmus ist in *Algorithmus 4* gegeben. Wir verwenden die Funktion FindClosestInLeaf(), um die Punkte im Blatt zu überprüfen. Das initiale Nachschlagen für des Finden des tiefsten Knotens, der die Begrenzung des Anfragevolumens enthält, ist ein modifiziertes Nachschlagen des Index. Dazu werden die beiden diagonal gegenüberliegenden Ecken in Integerkoordinaten konvertiert. Der Baum wird dann wie in *Algorithmus 2* traversiert, bis beide Indizes, die angeben, welcher Kindknoten als nächstes inspiziert werden soll, nicht mehr übereinstimmen.

Algorithm 4 | FindClosest

Input: query point *q*, maximal allowed distance *d*
lookup deepest node *N* containing bounding box of *q*
convert *q* to octree coordinate *i*
return FindClosestInNode(*N*, *q*, *i*, *d*)

4.4 RANSAC zum effizienten Schätzen von Parametern

Der Random Sample Consensus (RANSAC) Algorithmus ist ein Ansatz zum Bestimmen von Parametern eines Modells mittels Stichproben [10]. Dieser Algorithmus wird häufig zur Detektion von Linien und

Algorithm 5 | FindClosestInNode

Input: query point *q* and its coordinate *i*
Input: maximal allowed distance *d* and the current node *N*
1: compute child index *c* from *i*
2: **for** $j = 0$ to 8 **do**
3: get next closest child $C = N$.children[sequence[*c*][*j*]];
4: **if** *C* is outside of bounding ball **then**
5: **return** currently closest point *p*
6: **else**
7: **if** *C* is a leaf **then**
8: FindClosestInLeaf(*C*, *q*, *d*)
9: update currently closest point *p*
10: **else**
11: FindClosestInNode(*C*, *q*, *i*, *d*)
12: update currently closest point *p*
13: **end if**
14: **end if**
15: **end for**
16: **return** currently closest point *p*

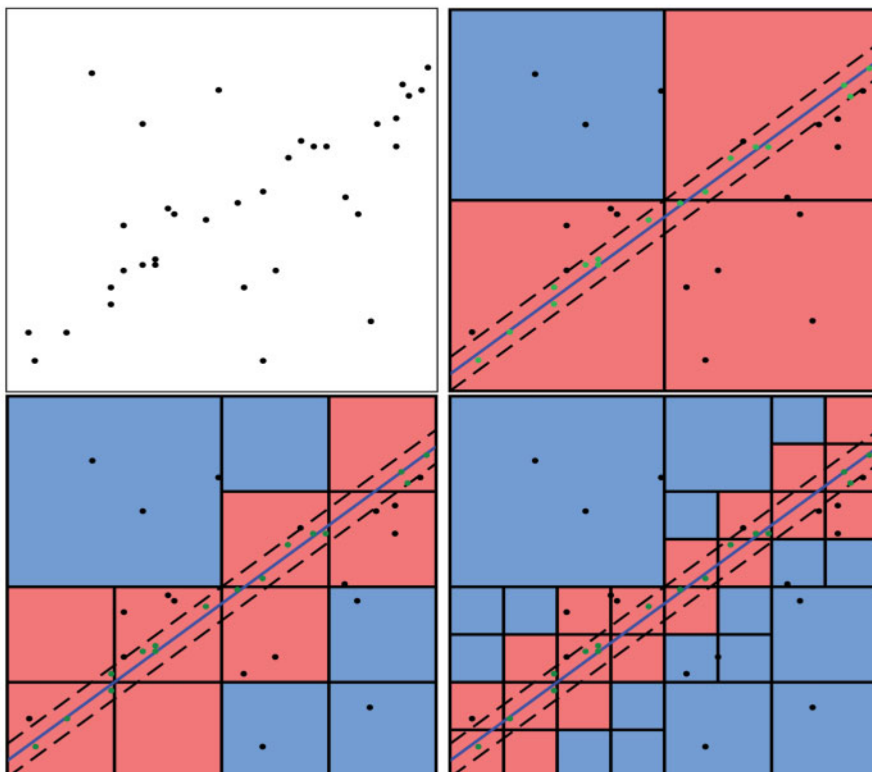

Abb. 10 | Geschwindigkeitsgewinn von RANSAC unter Verwendung eines Octrees. Oben links: Initiale Stichprobenmenge, in der eine Linie detektiert werden soll. Oben rechts: Eine Line wurde durch eine Teilmenge geschätzt. Die gestrichelte Linie gibt den maximalen Distanzschwellwert an. Unten: Mit Hilfe des Überschneidungstests werden alle mit dem Modell überlappenden Blätter ermittelt.

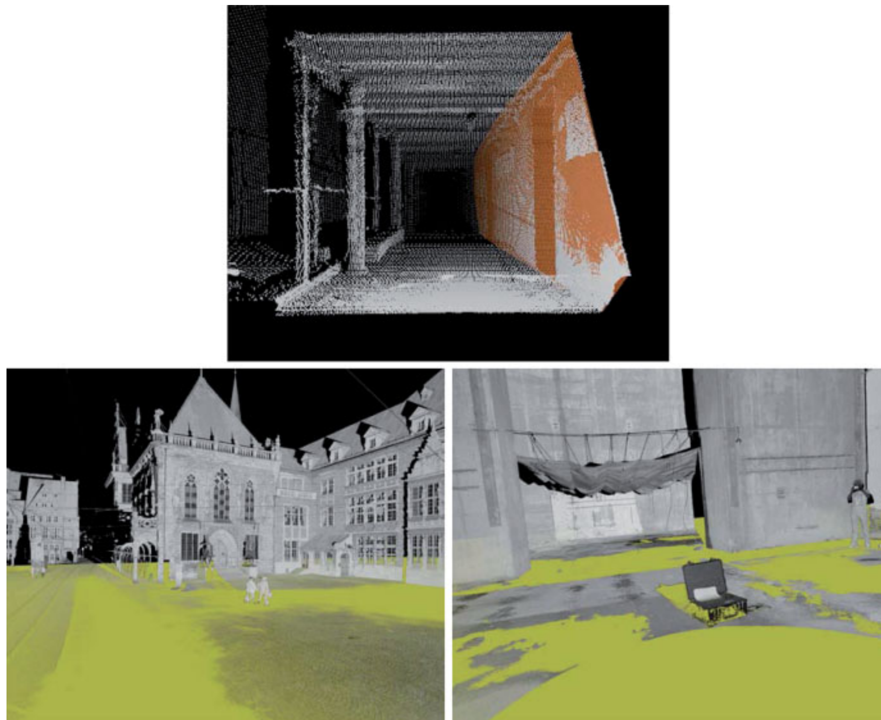

Abb. 11 | Mit RANSAC gefundene Ebenen in den drei Datensätzen aus *Abbildung 5*.

Ebenen eingesetzt, jedoch lässt er sich auf jedes parametrisierbare Modell anwenden. RANSAC ist ein iterativer Algorithmus, der wiederholt eine kleine Anzahl von Stichproben aus dem Datensatz, in dem das Modell vermutet wird, auswählt. Basierend auf dieser Auswahl werden die Modellparameter geschätzt, und anschließend wird im gesamten Datensatz überprüft, wie viele Daten mit der Schätzung übereinstimmen. Dieser Prozess wird wiederholt und das Modell mit den meisten Übereinstimungen ist das Ergebnis.

Der aufwändigste Schritt des Algorithmus ist die Bestimmung der Punkte, die mit dem Kandidatenmodell übereinstimmen. In einer unorganisierten 3D-Punktwolke muss man dazu über alle Punkte iterieren. Wir setzen dazu jedoch unseren Octree ein und erhalten eine signifikante Beschleunigung. Ähnlich zu *Abschnitt 4.2* überlagern wir das Kandidatenmodell rekursiv mit dem Octree und bestimmen dabei die Knoten, die Modellpunkte enthalten könnten. Dieser Prozess wird exemplarisch für die Liniendetektion im 2D-Fall in *Abb. 10* dargestellt. Nachdem die Kandidatenlinie generiert worden ist, wird sie dem Octree überlagert. Alle Knoten, die von der Linie berührt werden, werden rekursiv nach jeder Unterteilung bestimmt. Dadurch wird eine große Anzahl von Punkten ausgeschlossen, die nicht gegen das Modell überprüft werden müssen. Ein Vergleich der Rechenzeiten

Tabelle 6 | Durchschnittliche Laufzeit des RANSAC Algorithmus zur Bestimmung von Ebenen mit und ohne Octree. Ergebnisse wurden über 100 Durchläufe gemittelt. Für jeden Datensatz wurden 5000 Iterationen des RANSAC durchgeführt. Die Zeiten für den Octree beinhalten die Zeit für dessen Erstellung ([$\approx$ 8]s für den Bremer Marktplatz und [$\approx$ 10]s für den Bunker Datensatz). Alle Zeiten sind in Millisekunden angegeben.

Datensatz	ohne Octree	Octree	Beschleunigung
Kurt3D	1666.57	176.69	9.43
Bremer Marktplatz	388551.55	11084.81	35.05
Bunker Valentin	539536.13	26399.15	20.43

eines Standard-RANSAC zur Ebenendetektion mit unserer Octree-basierten Methode findet sich in *Tabelle 6*. Selbst unter Beachtung der Zeit, die für den Aufbau des Baumes notwendig ist, ergibt sich ein signifikanter Geschwindigkeitsvorteil. Beispiele für gefundene Ebenen für die 3 Datensätze finden sich in *Abb. 11*.

Des Weiteren lassen sich Octrees für die Verbesserung des RANSAC einsetzen. Schnabel et al. [27] haben gezeigt, dass durch sorgfältiges Auswählen der Stichprobenpunkte in enger Nachbarschaft, die Anzahl der notwendigen Iterationen des RANSAC-Algorithmus zum zuverlässigen Detektieren eines Modells um etliche Größenordnungen reduziert werden kann. Dies gelingt durch die Auswahl eines Stichprobenpunktes in einem geeigneten Blatt *l*. Anschließend werden weitere Stichproben nur von zufällig ausgewählten Vorgängerknoten von *l* ermittelt.

5 STAND VON WISSENSCHAFT UND TECHNIK

Seit Beginn der 1980er Jahre ist der Octree [20] neben dem *k*D-Baum [5, 8, 14] eine der am häufigsten verwendeten Datenstrukturen, um dreidimensionale Daten zu speichern. Ursprünglich wurde die Datenstruktur in der Computergrafik eingesetzt [17, 18, 13, 19], mittlerweile findet sie Anwendung in der theoretischen Physik [1] und Robotik [29].

In der Computergrafik und Visualisierung wird der Octree derzeit unter dem Namen *sparse voxel octree* verwendet. Dies ist ein Octreetyp, der sowohl große Datenmengen, als auch schnellen Zugriff darauf ermöglicht. Raycasting wird in $O(\log n)$ zur Verfügung gestellt, wobei n die Anzahl der Objekte in der Szene ist. In diesem Anwendungsszenario enthält der Octree nur Voxel und assoziierte Informationen, wie Farbe und Normale, also keine 3D-Punkte. Knoll et al. haben einen Raytracing-Algorithmus auf Octrees entwickelt

[17, 18]. Dazu verwenden sie die schnelle Indizierung von Frisken und Perry [9] und verbessern die Strahlverfolgung durch kohärentes Durchlaufen des Octrees.

QSplat [23] ist ein nicht auf Octrees basierendes Visualisierungsprogramm für 3D-Punktwolken. Die Punkte, ihre Normalen und Farben werden in einer Hierarchie von Kugelhüllen gespeichert. Interaktive Anzeige von großen Datenmengen wird erreicht, indem die Komplexität der Szene variiert wird und die Szenenhierarchie nur bis zu einer bestimmten Tiefe durchlaufen wird. Die zugrundeliegende Datenstruktur ist sehr kompakt, jedoch schwierig zu implementieren und nur sinnvoll für Visualisierungszwecke. Ein Octree ist dennoch kompakter und auch für andere Anwendungen geeignet.

Die in diesem Artikel vorgeschlagene Datemkomprimierung mit Hilfe des Octrees unterscheidet sich wesentlich von der in [26]. Dort wird die ursprüngliche Punktwolke quantisiert, d.h. sie wird durch die Mittelpunkte der Octree-Blätter ersetzt. Dieser Vorgang wird als „virtually lossless“ beschrieben. Ein ähnlicher Ansatz für *k*D-Bäume findet sich in [14]. Wir nutzen hingegen die hierarchische Struktur aus, um die ursprüngliche Punktwolke beinahe identisch beizubehalten. Dies ist ähnlich zu der Anwendung des Octrees im Bereich der Algorithmen für Farbquantisierung, um dort speichereffizent zu arbeiten [12]. Jede Stufe des Octrees repräsentiert ein Bit

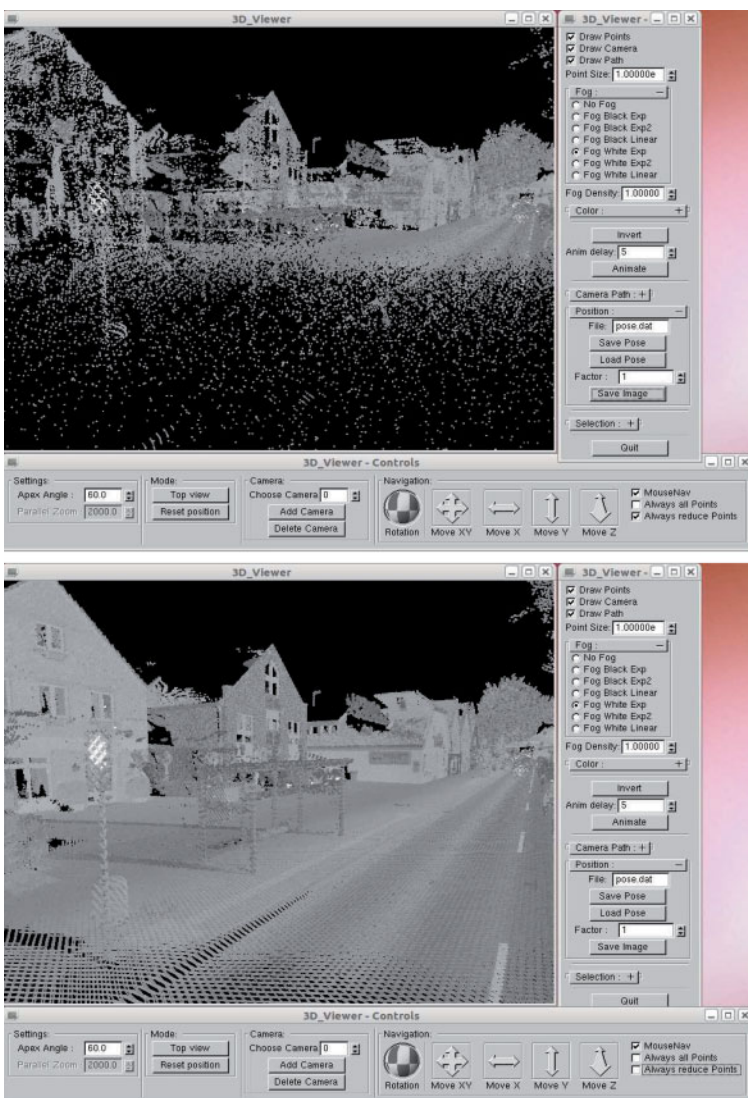

Abb. 12 | Dynamische Punktreduktion im *3DTK – The 3D Toolkit* (http://threedtk.de). Oben: Ausgedünnte Punktwolke, um einer Bildwiederholrate von 20 fps zu erreichen. Unten: Alle Messpunkte.

der Farbe, die enthalten ist, wobei mit dem Bit höchster Wertigkeit begonnen wird. Dies ist sehr ähnlich zu unserer Komprimierungsmethode, bei der die höchstwertigen Bits implizit gegeben sind und benutzt werden.

6 ZUSAMMENFASSUNG UND AUSBLICK

Um 1 Millarde Punkte mit Standardhardware zu verarbeiten, ist eine effiziente Datenstruktur für Punktwolken implementiert worden. Die Implementation steht im *3DTK – The 3D Toolkit* unter http://threedtk.de unter der GPL Lizenz zur Verfügung. Die Werkzeugbox enthält ein kleines Anzeigeprogramm mit Frustum Culling zum Verarbeiten von 1 Millarde Punkte. Liegen mehr Punkte im Frustum als die Grafikkarte bei gegebener Bildwiederholrate anzeigen kann, wird die Punktdichte dynamisch reduziert, so dass der Benutzer weiterhin in der Lage ist, in der Szene flüssig zu navigieren. *Abb. 12* zeigt eine Szene mit dynamischer Punktreduktion, die während der Oldenburger 3D-Tage 2011 mit dem kinematischen Messsystem VMX-250 der Firma Riegl aufgenommen wurde. Der Datensatz enthält 307 Millionen 3D-Punkte und kann problemlos in einem Stück verarbeitet werden.

In zukünftigen Arbeiten werden wir die Verwendung der vorgeschlagenen Datenstruktur in weiteren Algorithmen untersuchen. Derzeit finden Arbeiten bezüglich des Beschleunigens des RANSAC-Algorithmus [10] für das Finden von Ebenen sowie für schnelle Suche nach nächsten Punkten statt. Letzteres ist eine Grundvoraussetzung für effizientes Scanmatching mit dem ICP-Algorithmus, bzw. seiner global konsistenten Variante [6, 7].

DANKSAGUNG

An dieser Stelle gilt unser herzlicher Dank Nikolaus Studnicka für seine Idee, den Octree als Basis für ein Dateiformat zu verwenden. Des Weiteren bedanken wir uns bei der Firma RIEGL Laser Measurement Systems (www.riegl.com) für die Bereitstellung der Testdaten auf den Oldenburger 3D-Tagen. Ein ausführbare Windows Version des Anzeigeprogramms zum Testen unter http://plum.eecs.jacobs-university.de/exchange/Oldenburger3DTage.zip bezogen werden. Für die Unterstützung während der Oldenburger 3D-Tage danken wir ebenfalls Jochen Sprickerhof (Universität Osnabrück).

LITERATUR

[1] J. Bielack, O. Ghattas and E. J. Kim. Parallel octree-based finite element method for large-scale earthquake ground motion simulation. *Computer Modeling in Engineering and Sciences*, 10(2):99-112, 2005.

[2] American Society for Photogrammetry and Remote Sensing. Common Lidar Data Exchange Format. http://www.asprs.org/society/committees/lidar/lidarformat.html, February 2011.

[3] Andreas Nüchter et al. 3DTK – The 3D Toolkit. http://slam6d.sourceforge.net/, May 2011.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman and A. Y. Wu. An Optimal Algorithms for Approximate Nearest Neighbor Searching in Fixed Dimensions. *Journal of the ACM*, 45:891-923, 1998.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509-517, September 1975.

[6] P. Besl and N. McKay. A method for Registration of 3-D Shapes. *IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI)*, 14(2):239-256, 1992.

[7] D. Borrmann, J. Elseberg, K. Lingemann, A. Nüchter and J. Hertzberg. Globally consistent 3d mapping with scan matching. *Journal Robotics and Autonomous Systems (JRAS)*, 56(2):130-142, February 2008.

[8] M. Botsch, A. Wiratanaya and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proceedings of the 13th Eurographics workshop on Rendering (EGRW '02)*, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[9] S. F-Frisken and R. N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools*, 7(3), 2002.

[10] M. A. Fischler and R. C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24:381-395, 1981.

[11] J. H. Friedman, J. L. Bentley and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transaction on Mathematical Software*, 3(3):209-226, September 1977.

[12] M. Gervauts and W. Purgathofer. A simple method for color quantization: octree quantization. *Graphics Gems I*, pages 287-293, 1990.

[13] E. Gobbetti, F. Marton, G. Iglesias and A. Jose. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Visual Computing*, 24:797-806, July 2008.

[14] E. Hubo, T. Mertens, T. Haber and P. Bekaert. The quantized kd-tree: Efficient ray tracing of compressed point clouds. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, September 2006.

[15] K. M. Wurm et al. Octomap. http://octomap.sourceforge.net/, May 2011.

[16] F. Kern, M. Pospis and O. Pümm. Das Datenaustauschformat Binary Pointcloud (BPC) für TLS-Punktwolken. In Luhmann/Müller, editors, *Photogrammetrie Laserscanning Optische 3D-Messtechnik, Beiträge der Oldenburger 3D-Tage*, pages 20-30, Oldenburg, Germany, February 2009. Wichmann.

[17] A. Knoll, I. Wald, S. Parker and C. Hansen. Interactive isosurface ray tracing of large octree volumes. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pages 115-124, September 2006.

[18] A. M. Knoll, I. Wald and C. D. Hansen. Coherent multiresolution isosurface ray tracing. *Visual Computing*, 25:209-225, February 2009.

[19] S. Laine and T. Karras. Efficient sparse voxel octrees. In *Proceedings of the ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D '10)*, pages 55-63, New York, NY, USA, 2010. ACM.

[20] D. Meagher. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2):129-147, 1982.

[21] Radu Bogdan Rusu et al. Point cloud library. http://pointclouds.org/, May 2011.

[22] S. D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18:109-144, February 1982.

[23] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proceedings of the ACM SIGGRAPH*, 2000.

[24] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '11)*, Shanghai, China, May 2011.

[25] A. Samberg. An Implementation of the ASPRS LAS Standard. *ISPRS*, Volume XXXVI, Part 3/W52:363-372, 2007.

[26] R. Schnabel and R. Klein. Octree-based point-cloud compression. In *Eurographics Symposium on Point-Based Graphics*, 2006.

[27] R. Schnabel, R. Wahl, and R. Klein. Efficient RANSAC for Point-Cloud Shape Detection. *Computer Graphics Forum*, 2007.

[28] S. Thrun, D. Fox and W. Burgard. A real-time algorithm for mobile robot mapping with application to multi robot and 3D mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '00)*, San Francisco, CA, USA, April 2000.

[29] K. M. Wurm, A. Hornung, M. Bennewitz, C. Stachniss and W. Burgard. OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems. In *Proceedings of the IEEE ICRA Workshop on Best Practice in 3D Perception and Modeling for Mobile Manipulation, Anchorage*, AK, USA, 2010.

PhD Student Jan Elseberg

JACOBS UNIVERSITY BREMEN GGMBH
AUTOMATION GROUP, SCHOOL OF ENGINEERING AND SCIENCE

Campus Ring 1 | 28759 Bremen
E-Mail: j.elseberg@jacobs-university.de

PhD Student Dorit Borrmann

JACOBS UNIVERSITY BREMEN GGMBH
AUTOMATION GROUP, SCHOOL OF ENGINEERING AND SCIENCE

Campus Ring 1 | 28759 Bremen
E-Mail: d.borrmann@jacobs-university.de

Prof. Dr. Andreas Nüchter

JACOBS UNIVERSITY BREMEN GGMBH
AUTOMATION GROUP, SCHOOL OF ENGINEERING AND SCIENCE

Campus Ring 12 | 28759 Bremen
E-Mail: a.nuechter@jacobs-university.de

Manuskript eingereicht: 05.03.2011 Im Peer-Review-Verfahren bequtachtet